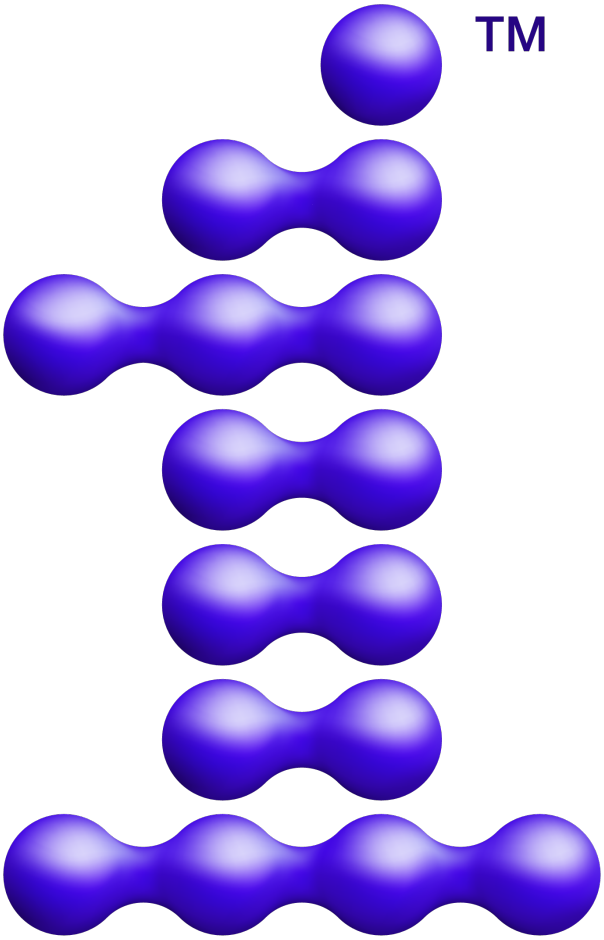


oneAPI

oneAPI Specification

Release 1.0-rev-2

Intel

Feb 09, 2021

CONTENTS

oneAPI is an open, free, and standards-based programming system that provides portability and performance across accelerators and generations of hardware. oneAPI consists of a language and libraries for creating parallel applications:

- *DPC++*: oneAPI's core language for programming accelerators and multiprocessors. DPCPP allows developers to reuse code across hardware targets (CPUs and accelerators such as GPUs and FPGAs) and tune for a specific architecture
- *oneDPL*: A companion to the DPC++ Compiler for programming oneAPI devices with APIs from C++ standard library, Parallel STL, and extensions.
- *oneDNN*: High performance implementations of primitives for deep learning frameworks
- *oneCCL*: Communication primitives for scaling deep learning frameworks across multiple devices
- *Level Zero*: System interface for oneAPI languages and libraries
- *oneDAL*: Algorithms for accelerated data science
- *oneTBB*: Library for adding thread-based parallelism to complex applications on multiprocessors
- *oneVPL*: Algorithms for accelerated video processing
- *oneMKL*: High performance math routines for science, engineering, and financial applications
- *Advanced Rendering*: Rendering

CHAPTER

ONE

INTRODUCTION

oneAPI simplifies software development by providing the same languages and programming models across accelerator architectures. In this section, we introduce the programming model.

Parallel application development is a combination of *API programming*, where the parallel algorithm is hidden behind an API provided by the system, and *direct programming*, where the application programmer writes the parallel algorithm.

When using API programming, a developer implements performance critical sections of the program with library calls. Well-defined and mature problem domains have high-performance solutions packaged as libraries. oneAPI defines a set of APIs for the most used data parallel domains, and oneAPI platforms provide library implementations across a variety of accelerators. Where possible, the API is based on established standards like BLAS. API programming enables a programmer to attain high performance across a diverse set of accelerators with minimal coding & tuning.

Some problem domains are not well suited to API programming because no standard solution exists or because solutions require a level of customization that cannot be easily implemented in a library. In this case, a developer uses direct programming and must explicitly code the parallel algorithm. oneAPI's programming model is based on data parallelism, where the same computation is performed on each data element, and parallelism of the application scales as the data scales. By allowing the programmer to directly express parallelism, data parallel algorithms make it possible to productively create highly efficient algorithms for parallel architectures.

Data parallel algorithms are used for many of the most computationally vdemanding problems including scientific computing, artificial intelligence, and visualization. Data parallel algorithms can be efficiently mapped to a diverse set of architectures: multi-core CPUs, GPUs, systolic arrays, and FPGAs.

1.1 Target Audience

The expected audience for this specification includes: application developers, middleware developers, system software providers, and hardware providers. As a *contributor* to this specification, you will shape the accelerator software ecosystem. A productive and high performing system must take into account the constraints at all levels of the software stack. As a *user* of this document, you can ensure that your components will inter-operate with applications and system software for the oneAPI platform.

1.2 Goals of the Specification

oneAPI seeks to provide:

- *Source-level compatibility*: oneAPI applications and middleware port to a conformant oneAPI platform through recompilation and re-tuning.
- *Performance transparency*: API's and language construct allow the programmer enough control over the mapping to hardware to create an efficient solution
- *Software stack portability*: Platform providers can port a oneAPI software stack by implementing the oneAPI Level Zero interface.

1.3 Definitions

This specification uses the definition of must, must not, required, and so on specified in RFC 2119.

1.4 Contribution Guidelines

This specification is a continuation of Intel's decades-long history of working with standards groups and industry/academia initiatives such as The Khronos Group, to create and define specifications in an open and fair process to achieve interoperability and interchangeability. oneAPI is intended to be an open specification and we encourage you to help us make it better. Your feedback is optional, but to enable Intel to incorporate any feedback you may provide to this specification, and to further upstream your feedback to other standards bodies, including The Khronos Group SYCL specification, please submit your feedback under the terms and conditions below. Any contribution of your feedback to the oneAPI Specification does not prohibit you from also contributing your feedback directly to other standard bodies, including The Khronos Group under their respective submission policies.

Contribute to the oneAPI Specification by opening issues in the oneAPI Specification GitHub repository.

1.4.1 Sign your work

Please include a signed-off-by tag in every contribution of your feedback. By including a signed-off-by tag, you agree that: (a) you have a right to license your feedback to Intel; (b) Intel will be free to use, disclose, reproduce, modify, license, or otherwise distribute your feedback at its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations; and (c) your feedback will be public and that a record of your feedback may be maintained indefinitely.

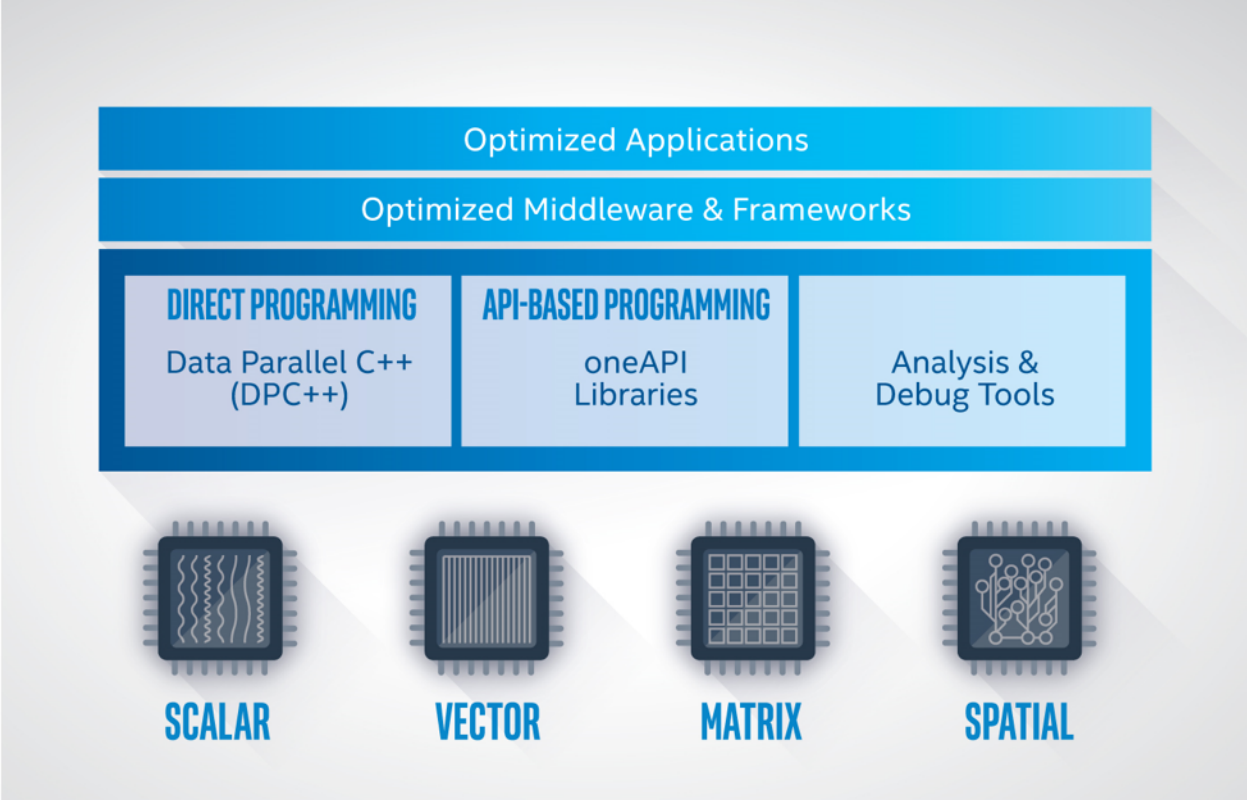
If you agree to the above, every contribution of your feedback must include the following line using your real name and email address: Signed-off-by: Joe Smith joe.smith@email.com

CHAPTER

TWO

SOFTWARE ARCHITECTURE

oneAPI provides a common developer interface across a range of data parallel accelerators (see the figure below). Programmers use DPC++ for both API programming and direct programming. The capabilities of a oneAPI platform are determined by the Level Zero interface, which provides system software a common abstraction for a oneAPI device.



2.1 oneAPI Platform

A oneAPI platform is comprised of a *host* and a collection of *devices*. The host is typically a multi-core CPU, and the devices are one or more GPUs, FPGAs, and other accelerators. The processor serving as the host can also be targeted as a device by the software.

Each device has an associated command *queue*. A application that employs oneAPI runs on the host, following standard C++ execution semantics. To run a *function object* on a device, the application submits a *command group* containing the function object to the device's queue. A function object contains a function definition together with associated variables. A function object submitted to a queue is also referred to as a *data parallel kernel* or simply a *kernel*.

The application running on the host and the functions running on the devices communicate through *memory*. oneAPI defines several mechanisms for sharing memory across the platform, depending on the capabilities of the devices:

Memory Sharing Mechanism	Description
Buffer objects	An application can create *buffer objects* to pass data to devices. A buffer is an array of data. A command group will define *accessor objects* to identify which buffers are accessed in this call to the device. The oneAPI runtime will ensure the data in the buffer is accessible to the function running on the device. The buffer-accessor mechanism is available on all oneAPI platforms
Unified addressing	Unified addressing guarantees that the host and all devices will share a unified address space. Pointer values in the unified address space will always refer to the same location in memory.
Unified shared memory	Unified shared memory enables data to be shared through pointers without using buffers and accessors. There are several levels of support for this feature, depending on the capabilities of the underlying device.

The *scheduler* determines when a command group is run on a device. The following mechanisms are used to determine when a command group is ready to run.

- If the buffer-accessor method is used, the command group is ready when the buffers are defined and copied to the device as necessary.
- If an ordered queue is used for a device, the command group is ready as soon as the prior command groups in the queue are finished.

- If unified shared memory is used, you must specify a set of event objects which the command group depends on, and the command group is ready when all of the events are completed.

The application on the host and the functions on the devices can *synchronize* through *events*, which are objects that can coordinate execution. If the buffer-accessor mechanism is used, the application and device can also synchronize through a *host accessor*, through the destruction of a buffer object, or through other more advanced mechanisms.

2.2 API Programming Example

API programming requires the programmer to specify the target device and the memory communication strategy. In the following example, we call the oneMKL matrix multiply routine, GEMM. We are writing in DPC++ and omitting irrelevant details.

We create a queue initialized with a *gpu_selector* to specify that we want the computation performed on a GPU, and we define buffers to hold the arrays allocated on the host. Compared to a standard C++ GEMM call, we add a parameter to specify the queue, and we replace the references to the arrays with references to the buffers that contain the arrays. Otherwise this is the standard GEMM C++ interface.

```
using namespace cl::sycl;

// declare host arrays
double *A = new double[M*N];
double *B = new double[N*P];
double *C = new double[M*P];

{
    // Initializing the devices queue with a gpu_selector
    queue q{gpu_selector()};

    // Creating 1D buffers for matrices which are bound to host arrays
    buffer<double, 1> a{A, range<1>{M*N}};
    buffer<double, 1> b{B, range<1>{N*P}};
    buffer<double, 1> c{C, range<1>{M*P}};

    mkl::transpose nT = mkl::transpose::nontrans;
    // Syntax
    //   void gemm(queue &exec_queue, transpose transa, transpose transb,
    //             int64_t m, int64_t n, int64_t k, T alpha,
    //             buffer<T,1> &a, int64_t lda,
    //             buffer<T,1> &b, int64_t ldb, T beta,
    //             buffer<T,1> &c, int64_t ldc);
    // call gemm
    mkl::blas::gemm(q, nT, nT, M, P, N, 1.0, a, M, b, N, 0.0, c, M);
}
// when we exit the block, the buffer destructor will write result back to C.
```

2.3 Direct Programming Example

With direct programming, we specify the target device and the memory communication strategy, as we do for API programming. In addition, we must define and submit a command group to perform the computation. In the following example, we write a simple data parallel matrix multiply. We are writing in DPC++ and omitting irrelevant details.

We create a queue initialized with a *gpu_selector* to specify that the command group should run on the GPU, and we define buffers to hold the arrays allocated on the host. We then submit the command group to the queue to perform the computation. The command group defines accessors to specify we are reading arrays A and B and writing to C. We then write a C++ lambda to create a function object that computes one element of the resulting matrix multiply. We specify this function object as a parameter to a `parallel_for` which maps the function across the arrays `A` and `B` in parallel. When we leave the scope, the destructor for the buffer object holding `C` writes the data back to the host array.

```
#include <CL/sycl.hpp>
using namespace sycl;

int main() {
  // declare host arrays
  double *Ahost = new double[M*N];
  double *Bhost = new double[N*P];
  double *Chost = new double[M*P];

  {
    // Initializing the devices queue with a gpu_selector
    queue q{gpu_selector()};

    // Creating 2D buffers for matrices which are bound to host arrays
    buffer<double, 2> a{Ahost, range<2>{M,N}};
    buffer<double, 2> b{Bhost, range<2>{N,P}};
    buffer<double, 2> c{Chost, range<2>{M,P}};

    // Submitting command group to queue to compute matrix c=a*b
    q.submit([&](handler &h){
                // Read from a and b, write to c
                auto A = a.get_access<access::mode::read>(h);
                auto B = b.get_access<access::mode::read>(h);
                auto C = c.get_access<access::mode::write>(h);

                int WidthA = a.get_range()[1];

                // Executing kernel
                h.parallel_for(range<2>{M, P},
                               [=](id<2> index){
                                 int row = index[0];
                                 int col = index[1];

                                 // Compute the result of one element in c
                                 double sum = 0.0;
                                 for (int i = 0; i < WidthA; i++) {
                                   sum += A[row][i] * B[i][col];
                                 }
                                 C[index] = sum;
                               });
            });
  }
  // when we exit the block, the buffer destructor will write result back to C.
```

(continues on next page)

(continued from previous page)

```
}
```

CHAPTER

THREE

DPC++

3.1 Overview

oneAPI Data Parallel C++ (DPC++) is the direct programming language and associated direct programming APIs of oneAPI. It provides the features needed to define data parallel functions and to launch them on devices. The language is comprised of the following components:

- C++. Every DPC++ program is also a C++ program. A compliant DPC++ implementation must support the C++17 Core Language (as specified in Sections 1-19 of ISO/IEC 14882:2017) or newer. See the C++ Standard.
- SYCL. DPC++ builds on the SYCL specification from The Khronos Group. The SYCL language enables the definition of data parallel functions that can be offloaded to devices and defines runtime APIs and classes that are used to orchestrate the offloaded functions.
- DPC++ Language extensions. A compliant DPC++ implementation must support the specified language features. These include unified shared memory (USM), ordered queues, and reductions. Some extensions are required only when the DPC++ implementation supports a specific class of device, as summarized in the *Extensions Table*. An implementation supports a class of device if it can target hardware that responds "true" for a DPC++ device type query, either through explicit support built into the implementation, or by using a lower layer that can support those device classes such as the oneAPI Level Zero (Level Zero). A DPC++ implementation must pass the conformance tests for all extensions that are required (*Extensions Table*) for the classes of devices that the implementation can support. (See SYCL Extensions.)

This specification requires a minimum of C++17 Core Language support and DPC++ extensions. These version and feature coverage requirements will evolve over time, with specific versions of C++ and SYCL being required, some additional extensions being required, and some DPC++ extensions no longer required if covered by newer C++ or SYCL versions directly.

Table 1: DPC++ Extensions Table: Support requirements for DPC++ implementations above SYCL 1.2.1

Feature	Where defined	CPU	GPU	FPGA	Test[1]
Accessor simplifications	SYCL 2020 provisional	Required	Required	Required	NA[3]
bit_cast	SYCL 2020 provisional	Required	Required	Required	NA[3]
Deduction guides	SYCL 2020 provisional	Required	Required	Required	NA[3]
Device specific queries	SYCL 2020 provisional	Not required[4]	Not required[4]	Not required[4]	NA[3]
Extended atomics	SYCL 2020 provisional	Required[7]	Required[7]	Not required[4]	NA[3]
Kernel func type attributes	SYCL 2020 provisional	Required	Required	Required	NA[3]
In-order queues	SYCL 2020 provisional	Required	Required	Required	NA[3]
Math array	SYCL 2020 provisional	Not required[4]	Not required[4]	Not required[4]	NA[3]
Optional lambda name	SYCL 2020 provisional	Required	Required	Required	NA[3]
Queue shortcuts	SYCL 2020 provisional	Required	Required	Required	NA[3]
Required work-group size	SYCL 2020 provisional	Required	Required	Required	NA[3]
Standard layout relaxed	SYCL 2020 provisional	Required	Required	Required	NA[3]
Unified Shared Memory	SYCL 2020 provisional	Required[2]	Required[2]	Required[2]	usm
Accessor properties	DPC++ extension	Required[8]	Required[8]	Required[8]	NA[3]
CXX standard library	DPC++ extension	Required	Required	Not required[4]	NA[3]
Data flow pipes	DPC++ extension	Not required	Not required	Required	fpga_tests
Enqueued barriers	DPC++ extension	Required	Required	Required	NA[3]
Group algorithms	DPC++ extension	Required	Required	Not required[4]	NA[3]
Group mask	DPC++ extension	Not required[4]	Not required[4]	Not required[4]	NA[3]
Parallel for shortcuts	DPC++ extension	Required	Required	Required	NA[3]
Pinned memory property	DPC++ extension	Required	Required	Required	NA[3]
Reductions	DPC++ extension	Required[5]	Required[5]	Not required[4]	NA[3]
Restrict all arguments	DPC++ extension	Required	Required	Required	NA[3]
Static local mem query	DPC++ extension	Not required[4]	Not required[4]	Not required[4]	NA[3]
Sub-groups	DPC++ extension	Required	Required	Not required	sub_group
Sub-group algorithms	DPC++ extension	Required[6]	Required[6]	Not required	sub_group

3.2 Detailed API and Language Descriptions

The SYCL 1.2.1 Specification describes the SYCL APIs and language. DPC++ extensions on top of SYCL are described in the SYCL Extensions repository. Some features defined in the SYCL 2020 Provisional Specification but not in the SYCL 1.2.1 Specification are required in DPC++, as summarized in *Extensions Table*, and most replace DPC++ extensions that were required in previous versions of this specification.

A brief summary of the required features from SYCL 2020 Provisional Specification (above SYCL 1.2.1 Specification) follows:

- Accessor simplifications - simplification of the accessor interface, reduction of verbosity in common code, and removal of need to specify template arguments in common cases. Section 4.7.6 of the SYCL 2020 Provisional Specification.
- bit_cast - inclusion of C++20 (p0476r2) `std::bit_cast` as `sycl::bit_cast`. Section 3.8.2 of the SYCL 2020 Provisional Specification.
- Deduction guides - simplifies common code patterns and reduces code length and verbosity by enabling Class Template Argument Deduction (CTAD) from modern C++. Distributed throughout the SYCL 2020 Provisional Specification.
- Device specific queries - kernel property queries associated with a specific device. Section 4.12 of the SYCL 2020 Provisional Specification.
- Extended atomics - alignment with C++20 `std::atomic_ref`, including some tweaks for memory models in SYCL. Support for floating-point types and shorthand operators. Section 4.17.3 of the SYCL 2020 Provisional Specification. Additional atomic-related queries are defined in Table 4.19, and some changes to fences and barriers are reflected in Section 4.17.1 (both in the SYCL 2020 Provisional Specification).
- Kernel function type attributes - definition of kernel attributes as function type attributes that allows them to be applied to lambdas. Definition of some core attributes. Section 5.7 of the SYCL 2020 Provisional Specification.
- In-order queues - defines simple in-order semantics for queues, to simplify common coding patterns. Section 4.6.5 of the SYCL 2020 Provisional Specification.
- Math array - contiguous fixed-size portable container. Section 4.16.3 of the SYCL 2020 Provisional Specification.
- Optional lambda name - removes requirement to manually name lambdas that define kernels. Simplifies coding and enables composability with libraries. Lambdas can still be manually named, if desired, such as when debugging or interfacing with a `sycl::program` object. Section 4.14.2 of the SYCL 2020 Provisional Specification.
- Queue shortcuts - defines kernel invocation functions directly on the queue classes, to simplify code patterns where dependencies and/or accessors do not need to be created within the additional command group scope. Reduces code verbosity in some common patterns. Section 4.6.5 of the SYCL 2020 Provisional Specification.
- Required work-group size - defines an attribute that can be applied to kernels (including lambda definitions of kernels) which signals that the kernel will only be invoked with a specific work-group size. This is an optimization attribute that enables optimizations based on additional user-driven information. Section 5.7 of the SYCL 2020 Provisional Specification.

[1] Test directory within extension tests
[3] Not yet available.
[4] Likely to be required in the future
[7] DPC++ requirement does not include support for atomics in the generic address space
[2] Minimum of explicit USM support
[8] DPC++ requirement is for the general property mechanism, and not specific properties within it
[5] DPC++ requirement is for one dimensional reductions, single reduction variable support
[6] DPC++ requirement is for sub-group algorithms that have equivalent group algorithms

- Standard layout relaxed - removes the requirement that data shared by a host and device(s) must be C++ standard layout types. Requires device compilers to validate layout compatibility. Section 4.14.4 of the SYCL 2020 Provisional Specification.
- Unified Shared Memory (USM) - defines pointer based memory accesses and management interfaces. Provides the ability to create allocations that are visible and have consistent pointer values across both host and device(s). Different USM capability levels are defined, corresponding to different levels of device and implementation support. Section 4.8 of the SYCL 2020 Provisional Specification.

A brief summary of the extensions is as follows:

- Accessor properties - compile-time accessor properties that are visible to the compiler.
- CXX standard library - enable subset of the C and C++ standard libraries in device code.
- Data flow pipes - enable efficient First-In, First-Out (FIFO) communication in DPC++, a mechanism commonly used when describing algorithms for spatial architectures such as FPGAs.
- Enqueued barriers - simplifies dependence creation and tracking for some common programming patterns by allowing coarser grained synchronization within a queue without manual creation of fine grained dependencies.
- Group algorithms - defines collective operations that operate across groups of work-items, including broadcast, reduce, and scan. Improves productivity by providing common algorithms without explicit coding, and enables optimized implementations to exist for combinations of device and runtime.
- Group mask - defines a type that can represent a set of work-items from a group, and collective operations that create or operate on that type such as ballot and count.
- Parallel for shortcuts - simplification of common patterns such as invoking a kernel with a scalar range.
- Pinned memory property - optimization indicating that a buffer should use a specific memory resource if possible, to accelerate movement of data between host and devices in some implementations.
- Reductions - provides a reduction abstraction to the ND-range form of *parallel_for*. Improves productivity by providing the common reduction pattern without explicit coding, and enables optimized implementations to exist for combinations of device, runtime, and reduction properties.
- Restrict all arguments - defines an attribute that can be applied to kernels (including lambda definitions of kernels) which signals that there will be no memory aliasing between any pointer arguments that are passed to or captured by a kernel. This is an optimization attribute that can have large impact when the developer knows more about the kernel arguments than a compiler can infer or safely assume.
- Static local memory query - query for the amount of local memory used by a compiler and unavailable for dynamic use.
- Subgroups - defines a grouping of work-items within a work-group. Synchronization of work-items in a sub-group can occur independently of work-items in other sub-groups, and sub-groups expose communication operations across work-items in the group. Subgroups commonly map to SIMD hardware where it exists.
- Subgroup algorithms - defines collective operations across work-items in a sub-group that are available only for sub-groups. Also enables algorithms from the more generic "group algorithms" extension as sub-group collective operations.

3.3 Open Source Implementation

An open source implementation is available under an LLVM license. Details on incomplete features and known issues are available in the Release Notes (and the Getting Started Guide until the release notes are available).

3.4 Testing

A DPC++ implementation must pass:

1. The extension tests for any extension implemented from the *Extensions Table*. Each extension in the *Extensions Table* lists the name of the directory that contains corresponding tests, within the extension tests tree.

3.5 Acknowledgment

We thank the DPC++ and oneDPL Technical Advisory Board for their valuable feedback, and the Khronos SYCL working group for their efforts defining and evolving the SYCL specification.

CHAPTER

FOUR

ONEDPL

The oneAPI DPC++ Library (oneDPL) provides the functionality specified in the C++ standard, with extensions to support data parallelism and offloading to devices, and with extensions to simplify its usage for implementing data parallel algorithms.

The library is comprised of the following components:

- A subset of the C++ standard library which you can use with buffers and data parallel kernels.
- Parallel STL algorithms, complemented with execution policies and companion APIs for running on oneAPI devices. (See *Extensions to Parallel STL*.)
- Extensions: an additional set of library classes and functions that are known to be useful in practice but are not yet included into C++ or SYCL specifications. (See *Specific API of oneDPL*.)

4.1 Namespaces

oneDPL uses `namespace oneapi::std` for the subset of the standard C++ library for kernels, and uses `namespace oneapi::dpl` for other functionality including Parallel STL algorithms, oneDPL execution policies, and extensions.

4.2 Supported C++ Standard Library APIs and Algorithms

oneDPL defines a subset of the C++ standard library APIs for use in DPC++ kernels. These APIs can be employed in the kernels similarly to how they are employed in code for a typical CPU-based platform.

For all C++ algorithms accepting execution policies (as defined by C++ Standard), oneDPL provides an implementation for oneAPI devices via `oneapi::dpl::execution::device_policy`. These algorithms must be capable of processing data in SYCL buffers (passed via `oneapi::dpl::begin/end`) and in unified shared memory (USM). (See *Extensions to Parallel STL*.)

4.3 Extensions to Parallel STL

oneDPL extends Parallel STL with the following APIs.

4.3.1 DPC++ Execution Policy

A DPC++ execution policy specifies where and how an algorithm runs.

```
// Defined in <oneapi/dpl/execution>

namespace oneapi {
  namespace dpl {
    namespace execution {

      template <typename KernelName = /*unspecified*/>
      class device_policy;

      device_policy<> dpcpp_default;

      template <typename KernelName = /*unspecified*/>
      device_policy<KernelName>
      make_device_policy( sycl::queue );

      template <typename KernelName = /*unspecified*/>
      device_policy<KernelName>
      make_device_policy( sycl::device );

      template <typename NewKernelName, typename OldKernelName>
      device_policy<NewKernelName>
      make_device_policy( const device_policy<OldKernelName>& = dpcpp_default );
    }
  }
}
```

`dpcpp_default` is a predefined execution policy object to run algorithms on the default DPC++ device.

device_policy class

```
template <typename KernelName = /*unspecified*/>
class device_policy
{
public:
    using kernel_name = KernelName;

    device_policy();
    template <typename OtherName>
    device_policy( const device_policy<OtherName>& );
    explicit device_policy( sycl::queue );
    explicit device_policy( sycl::device );

    sycl::queue queue() const;
    operator sycl::queue() const;
};
```

An object of the `device_policy` type is associated with a `sycl::queue` that is used to run algorithms on a DPC++ compliant device.

The `KernelName` template parameter, also aliased as `kernel_name` within the class template, is to explicitly provide a name for DPC++ kernels executed by an algorithm the policy is passed to.

```
device_policy()
```

Construct a policy object associated with a queue created with the default device selector.

```
template <typename OtherName>
device_policy( const device_policy<OtherName>& policy )
```

Construct a policy object associated with the same queue as `policy`, by changing the kernel name of the given policy to `kernel_name` defined for the new policy.

```
explicit device_policy( sycl::queue queue )
```

Construct a policy object associated with the given queue.

```
explicit device_policy( sycl::device device )
```

Construct a policy object associated with a queue created for the given device.

```
sycl::queue queue() const
```

Return the queue the policy is associated with.

```
operator sycl::queue() const
```

Allow implicit conversion of the policy to a `sycl::queue` object.

make_device_policy function

The `make_device_policy` function templates simplify `device_policy` creation.

```
template <typename KernelName = /*unspecified*/>
device_policy<KernelName>
make_device_policy( sycl::queue queue )
```

Return a policy object associated with `queue`, with a kernel name possibly provided as the template argument, otherwise unspecified.

```
template <typename KernelName = /*unspecified*/>
device_policy<KernelName>
make_device_policy( sycl::device device )
```

Return a policy object to run algorithms on `device`, with a kernel name possibly provided as the template argument, otherwise unspecified.

```
template <typename NewKernelName, typename OldKernelName>
device_policy<NewKernelName>
make_device_policy( const device_policy<OldKernelName>& policy = dpcpp_default )
```

Return a policy object constructed from `policy`, with a new kernel name provided as the template argument. If no policy object is provided, the new policy is constructed from `dpcpp_default`.

4.3.2 Buffer wrappers

```
// Defined in <oneapi/dpl/iterator>

namespace oneapi {
  namespace dpl {

    template < typename T, typename AllocatorT, sycl::access::mode Mode >
    /*unspecified*/ begin( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                           sycl::mode_tag_t<Mode> tag = sycl::read_write );

    template < typename T, typename AllocatorT, sycl::access::mode Mode >
    /*unspecified*/ begin( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                           sycl::mode_tag_t<Mode> tag, sycl::property::noinit );

    template < typename T, typename AllocatorT >
    /*unspecified*/ begin( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                           sycl::property::noinit );


    template < typename T, typename AllocatorT, sycl::access::mode Mode >
    /*unspecified*/ end( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                         sycl::mode_tag_t<Mode> tag = sycl::read_write );

    template < typename T, typename AllocatorT, sycl::access::mode Mode >
    /*unspecified*/ end( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                         sycl::mode_tag_t<Mode> tag, sycl::property::noinit );

    template < typename T, typename AllocatorT >
    /*unspecified*/ end( sycl::buffer<T, /*dim=*/1, AllocatorT> buf,
                         sycl::property::noinit );

  }
}
```

`oneapi::dpl::begin` and `oneapi::dpl::end` are helper functions for passing DPC++ buffers to oneDPL algorithms. These functions accept a buffer and return an object of an unspecified type that satisfies the following requirements:

- it is `CopyConstructible`, `CopyAssignable`, and comparable with operators == and `!=`;
- the following expressions are valid: `a + n`, `a - n`, `a - b`, where `a` and `b` are objects of the type, and `n` is an integer value;
- it provides the `get_buffer()` method that returns the buffer passed to the `begin` or `end` function.

When invoking an algorithm, the buffer passed to `begin` should be the same as the buffer passed to `end`. Otherwise, the behavior is undefined.

`sycl::mode_tag_t` and `sycl::property::noinit` parameters allow to specify an access mode to be used for accessing the buffer by algorithms. The mode serves as a hint, and can be overridden depending on semantics of the algorithm. When invoking an algorithm, the same access mode arguments should be used for `begin` and `end`. Otherwise, the behavior is undefined.

```
using namespace oneapi;
auto buf_begin = dpl::begin(buf, sycl::write_only);
auto buf_end_1 = dpl::end(buf, sycl::write_only);
auto buf_end_2 = dpl::end(buf, sycl::write_only, sycl::noinit);
```

(continues on next page)

(continued from previous page)

```
dpl::fill(dpl::dpcpp_default, buf_begin, buf_end_1, 42); // allowed
dpl::fill(dpl::dpcpp_default, buf_begin, buf_end_2, 42); // not allowed
```

4.4 Specific API of oneDPL

The oneDPL extensions include iterators, function objects, and parallel algorithms.

4.4.1 Function Objects

The oneDPL function objects are defined in the `<oneapi/dpl/functional>` header, in `namespace oneapi::dpl`.

```
namespace oneapi {
namespace dpl {
    struct identity
    {
        template <typename T>
        constexpr T&&
        operator()(T&& t) const noexcept;
    };
}
}
```

The `oneapi::dpl::identity` class implements an identity operation. Its function operator receives an instance of a type and returns the argument unchanged.

4.4.2 Iterators

The oneDPL iterators are defined in the `<oneapi/dpl/iterator>` header, in `namespace oneapi::dpl`.

```
template <typename Integral>
class counting_iterator
{
  public:
    using difference_type = /* a signed integer type of the same size as Integral */;
    using value_type = Integral;
    using reference = Integral;

    counting_iterator();
    explicit counting_iterator(Integral init);

    reference operator*() const;
    reference operator[](difference_type i) const;

    difference_type operator-(const counting_iterator& it) const;

    counting_iterator operator+(difference_type forward) const;
    counting_iterator operator-(difference_type backward) const;

    counting_iterator& operator+=(difference_type forward);
    counting_iterator& operator-=(difference_type backward);
```

(continues on next page)

(continued from previous page)

```
    counting_iterator& operator++();
    counting_iterator& operator--();
    counting_iterator& operator++(int);
    counting_iterator& operator--(int);

    bool operator==(const counting_iterator& it) const;
    bool operator!=(const counting_iterator& it) const;
    bool operator<(const counting_iterator& it) const;
    bool operator>(const counting_iterator& it) const;
    bool operator<=(const counting_iterator& it) const;
    bool operator>=(const counting_iterator& it) const;
};
```

`counting_iterator` is a random access iterator-like type that represents an integer counter. When dereferenced, `counting_iterator` provides an Integral rvalue equal to the value of the counter; dereference operations cannot be used to modify the counter. The arithmetic and comparison operators of `counting_iterator` behave as if applied to the values of Integral type representing the counters of the iterator instances passed to the operators.

```
class discard_iterator
{
  public:
    using difference_type = std::ptrdiff_t;
    using value_type = /* unspecified */;
    using reference = /* unspecified */;

    discard_iterator();
    explicit discard_iterator(difference_type init);

    reference operator*() const;
    reference operator[](difference_type) const;

    difference_type operator-(const discard_iterator& it) const;

    discard_iterator operator+(difference_type forward) const;
    discard_iterator operator-(difference_type backward) const;

    discard_iterator& operator+=(difference_type forward);
    discard_iterator& operator-=(difference_type backward);

    discard_iterator& operator++();
    discard_iterator& operator--();
    discard_iterator operator++(int);
    discard_iterator operator--(int);

    bool operator==(const discard_iterator& it) const;
    bool operator!=(const discard_iterator& it) const;
    bool operator<(const discard_iterator& it) const;
    bool operator>(const discard_iterator& it) const;
};
```

`discard_iterator` is a random access iterator-like type that, when dereferenced, provides an lvalue that may be assigned an arbitrary value. The assignment has no effect on the `discard_iterator` instance; the write is discarded. The arithmetic and comparison operators of `discard_iterator` behave as if applied to integer counter values maintained by the iterator instances to determine their position relative to each other.

```
template <typename SourceIterator, typename IndexMap>
class permutation_iterator
{
  public:
    using difference_type =
        typename std::iterator_traits<SourceIterator>::difference_type;
    using value_type = typename std::iterator_traits<SourceIterator>::value_type;
    using pointer = typename std::iterator_traits<SourceIterator>::pointer;
    using reference = typename std::iterator_traits<SourceIterator>::reference;

    permutation_iterator(const SourceIterator& input1, const IndexMap& input2,
                         std::size_t index = 0);

    reference operator*() const;
    reference operator[](difference_type i) const;

    difference_type operator-(const permutation_iterator& it) const;

    permutation_iterator operator+(difference_type forward) const;
    permutation_iterator operator-(difference_type backward) const;

    permutation_iterator& operator+=(difference_type forward);
    permutation_iterator& operator-=(difference_type forward);

    permutation_iterator& operator++();
    permutation_iterator& operator--();
    permutation_iterator operator++(int);
    permutation_iterator operator--(int);

    bool operator==(const permutation_iterator& it) const;
    bool operator!=(const permutation_iterator& it) const;
    bool operator<(const permutation_iterator& it) const;
    bool operator>(const permutation_iterator& it) const;
    bool operator<=(const permutation_iterator& it) const;
    bool operator>=(const permutation_iterator& it) const;
};
```

`permutation_iterator` is a random access iterator-like type whose dereferenced value set is defined by the source iterator provided, and whose iteration order over the dereferenced value set is defined by either another iterator or a functor that maps the `permutation_iterator` index to the index of the source iterator. The arithmetic and comparison operators of `permutation_iterator` behave as if applied to integer counter values maintained by the iterator instances to determine their position in the index map.

`permutation_iterator::operator*` uses the counter value of the instance on which it is invoked to index into the index map. The corresponding value in the map is then used to index into the value set defined by the source iterator. The resulting lvalue is returned as the result of the operator.

`permutation_iterator::operator[]` uses the parameter `i` to index into the index map. The corresponding value in the map is then used to index into the value set defined by the source iterator. The resulting lvalue is returned as the result of the operator.

```
template <typename SourceIterator, typename IndexMap>
permutation_iterator<SourceIterator, IndexMap>
make_permutation_iterator(SourceIterator source, IndexMap map);
```

`make_permutation_iterator` constructs and returns an instance of `permutation_iterator` using the source iterator and index map provided.

```
template <typename Iterator, typename UnaryFunc>
class transform_iterator
{
  public:
    using difference_type = typename std::iterator_traits<Iterator>::difference_type;
    using reference = typename std::invoke_result<UnaryFunc,
                          typename std::iterator_traits<Iterator>::reference>::type;
    using value_type = typename std::remove_reference<reference>::type;
    using pointer = typename std::iterator_traits<Iterator>::pointer;

    transform_iterator(Iterator it, UnaryFunc unary_func);
    transform_iterator(const transform_iterator& input);
    transform_iterator& operator=(const transform_iterator& input);

    reference operator*() const;
    reference operator[](difference_type i) const;

    difference_type operator-(const transform_iterator& it) const

    transform_iterator operator+(difference_type forward) const;
    transform_iterator operator-(difference_type backward) const;

    transform_iterator& operator+=(difference_type forward);
    transform_iterator& operator-=(difference_type backward);

    transform_iterator& operator++();
    transform_iterator& operator--();
    transform_iterator operator++(int);
    transform_iterator operator--(int);

    bool operator==(const transform_iterator& it) const;
    bool operator!=(const transform_iterator& it) const;
    bool operator<(const transform_iterator& it) const;
    bool operator>(const transform_iterator& it) const;
    bool operator<=(const transform_iterator& it) const;
    bool operator>=(const transform_iterator& it) const;
};
```

`transform_iterator` is a random access iterator-like type whose dereferenced value set is defined by the unary function and source iterator provided. When dereferenced, `transform_iterator` provides the result of the unary function applied to the corresponding element of the source iterator; dereference operations cannot be used to modify the elements of the source iterator unless the unary function result includes a reference to the element. The arithmetic and comparison operators of `transform_iterator` behave as if applied to the source iterator itself.

```
template <typename UnaryFunc, typename Iterator>
transform_iterator<UnaryFunc, Iterator>
make_transform_iterator(Iterator, UnaryFunc);
```

`make_transform_iterator` constructs and returns an instance of `transform_iterator` using the source iterator and unary function object provided.

```
template <typename... Iterators>
class zip_iterator
{
  public:
    using difference_type = typename std::make_signed<std::size_t>::type;
    using value_type =
```

(continues on next page)

(continued from previous page)

```
        std::tuple<typename std::iterator_traits<Iterators>::value_type...>;
    using reference = /* unspecified tuple of reference types */;
    using pointer =
        std::tuple<typename std::iterator_traits<Iterators>::pointer...>;

    zip_iterator();
    explicit zip_iterator(Iterators... args);
    zip_iterator(const zip_iterator& input);
    zip_iterator& operator=(const zip_iterator& input);

    reference operator*() const;
    reference operator[](difference_type i) const;

    difference_type operator-(const zip_iterator& it) const;
    zip_iterator operator-(difference_type backward) const;
    zip_iterator operator+(difference_type forward) const;

    zip_iterator& operator+=(difference_type forward);
    zip_iterator& operator-=(difference_type backward);

    zip_iterator& operator++();
    zip_iterator& operator--();
    zip_iterator operator++(int);
    zip_iterator operator--(int);

    bool operator==(const zip_iterator& it) const;
    bool operator!=(const zip_iterator& it) const;
    bool operator<(const zip_iterator& it) const;
    bool operator>(const zip_iterator& it) const;
    bool operator<=(const zip_iterator& it) const;
    bool operator>=(const zip_iterator& it) const;
};
```

`zip_iterator` is an iterator-like type defined over one or more iterators. When dereferenced, the value returned from `zip_iterator` is a tuple of the values returned by dereferencing the source iterators over which the `zip_iterator` is defined. The arithmetic operators of `zip_iterator` update the source iterators of a `zip_iterator` instance as though the operation were applied to each of these iterators.

```
template <typename... Iterators>
zip_iterator<Iterators...>
make_zip_iterator(Iterators...);
```

`make_zip_iterator` constructs and returns an instance of `zip_iterator` using the set of source iterators provided.

4.4.3 Parallel Algorithms

The parallel algorithms are defined in the `<oneapi/dpl/algorithm>` header, in `namespace oneapi::dpl`.

```
template<typename Policy, typename InputKeyIt, typename InputValueIt,
    typename OutputValueIt,
    typename T = typename std::iterator_traits<InputValueIt>::value_type,
    typename BinaryPred =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type>,
    typename BinaryOp =
```

(continues on next page)

(continued from previous page)

```
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>>
OutputValueIt
exclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first,
    InputKeyIt keys_last, InputValueIt values_first, OutputValueIt values_result,
    T initial_value = 0,
    BinaryPred binary_pred =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type>(),
    BinaryOp binary_op =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>());
```

`oneapi::dpl::exclusive_scan_by_segment` performs partial prefix scans by applying the `binary_op` operation to a sequence of values. Each partial scan applies to a contiguous subsequence determined by the keys associated with the values being equal according to the `binary_pred` predicate, and the first element of each scan is the initial value provided. The return value is an iterator targeting the end of the result sequence.

The initial value used if one is not provided is an instance of the `value_type` of the `InputValueIt` iterator type initialized to 0. If no binary predicate is provided for the comparison of keys an instance of `std::equal_to` with the `value_type` of the `InputKeyIt` iterator type is used. Finally, an instance of `std::plus` with the `value_type` of the `InputValueIt` iterator type is used if no binary operator is provided to combine the elements of the value subsequences.

```
template<typename Policy, typename InputKeyIt, typename InputValueIt,
    typename OutputValueIt,
    typename BinaryPredcate =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type,
    typename BinaryOp =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>>
OutputValueIt
inclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first,
    InputKeyIt keys_last, InputValueIt values_first, OutputValueIt values_result
    BinaryPred binary_pred =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type>(),
    BinaryOp binary_op =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>());
```

`oneapi::dpl::inclusive_scan_by_segment` performs partial prefix scans by applying the `binary_op` operation to a sequence of values. Each partial scan applies to a contiguous subsequence determined by the keys associated with the values being equal according to the `binary_pred` predicate. The return value is an iterator targeting the end of the result sequence.

If no binary predicate is provided for the comparison of keys an instance of `std::equal_to` with the `value_type` of the `InputKeyIt` iterator type is used. An instance of `std::plus` with the `value_type` of the `InputValueIt` iterator type is used if no binary operator is provided to combine the elements of the value subsequences.

```
template<typename Policy, typename InputKeyIt, typename InputValueIt,
    typename OutputKeyIt, typename OutputValueIt,
    typename BinaryPredcate =
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type,
    typename BinaryOp =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>>
std::pair<OutputKeyIt,OutputValueIt>
reduce_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_last,
    InputValueIt values_first, OutputKeyIt keys_result,
    OutputValueIt values_result,
    BinaryPred binary_pred =
```

(continues on next page)

(continued from previous page)

```
        std::equal_to<typename std::iterator_traits<InputKeyIt>::value_type>(),
    BinaryOp binary_op =
        std::plus<typename std::iterator_traits<InputValueIt>::value_type>());
```

`oneapi::dpl::reduce_by_segment` performs partial reductions on a sequence of values. Each reduction is computed with the `binary_op` operation for a contiguous subsequence of values determined by the associated keys being equal according to the `binary_pred` predicate. For each subsequence the first of the equal keys is stored into `keys_result` and the computed reduction is stored into `values_result`. The return value is a pair of iterators holding the end of the resulting sequences.

If no binary predicate is provided for the comparison of keys an instance of `std::equal_to` with the `value_type` of the `InputKeyIt` iterator type is used. An instance of `std::plus` with the `value_type` of the `InputValueIt` iterator type is used to combine the values in each subsequence identified if a binary operator is not provided.

```
template<typename Policy, typename InputIt1, typename InputIt2, typename OutputIt,
    typename Comparator =
        std::less<typename std::iterator_traits<InputIt>::value_type>>
OutputIt
binary_search(Policy&& policy, InputIt1 start, InputIt1 end,
    InputIt2 value_first, InputIt2 value_last, OutputIterator result,
    Comparator comp =
        std::less<typename std::iterator_traits<InputIt1>::value_type>());
```

`oneapi::dpl::binary_search` performs a binary search over the data in `[start, end)` for each value in `[value_first, value_last)`. If the value exists in the data searched then the corresponding element in `[result, result + distance(value_first, value_last))` is set to true, otherwise it is set to false.

If no comparator is provided, `operator<` is used to determine when the search value is less than an element in the range being searched.

```
template<typename Policy, typename InputIt1, typename InputIt2, typename OutputIt,
    typename Comparator =
        std::less<typename std::iterator_traits<InputIt>::value_type>>
OutputIt
lower_bound(Policy&& policy, InputIt1 start, InputIt1 end,
    InputIt2 value_first, InputIt2 value_last, OutputIterator result,
    Comparator comp =
        std::less<typename std::iterator_traits<InputIt1>::value_type>());
```

`oneapi::dpl::lower_bound` performs a binary search over the data in `[start, end)` for each value in `[value_first, value_last)` to find the lowest index at which the search value could be inserted in `[start, end)` without violating the ordering defined by the comparator provided. That lowest index is then assigned to the corresponding element in `[result, result + distance(value_first, value_last))`.

If no comparator is provided, `operator<` is used to determine when the search value is less than an element in the range being searched.

```
template<typename Policy, typename InputIt1, typename InputIt2, typename OutputIt,
    typename Comparator =
        std::less<typename std::iterator_traits<InputIt>::value_type>>
OutputIt
upper_bound(Policy&& policy, InputIt1 start, InputIt1 end,
    InputIt2 value_first, InputIt2 value_last, OutputIterator result,
    Comparator comp =
        std::less<typename std::iterator_traits<InputIt1>::value_type>());
```

`oneapi::dpl::upper_bound` performs a binary search over the data in `[start, end)` for each value in `[value_first, value_last)` to find the highest index at which the search value could be inserted in `[start, end)` without violating the ordering defined by the comparator provided. That highest index is then assigned to the corresponding element in `[result, result + distance(value_first, value_last))`.

If no comparator is provided, `operator<` is used to determine when the search value is less than an element in the range being searched.

CHAPTER

FIVE

ONEDNN

oneAPI Deep Neural Network Library (oneDNN) is a performance library containing building blocks for for deep learning applications and frameworks. oneDNN supports:

- CNN primitives (Convolutions, Inner product, Pooling, etc.)
- RNN primitives (LSTM, Vanilla, RNN, GRU)
- Normalizations (LRN, Batch, Layer)
- Elementwise operations (ReLU, Tanh, ELU, Abs, etc.)
- Softmax, Sum, Concat, Shuffle
- Reorders from/to optimized data layouts
- 8-bit integer, 16-, 32-bit, and bfloat16 floating point data types

```
// Tensor dimensions
int N, C, H, W;

// User-owned DPC++ objects
sycl::device dev {sycl::gpu_selector {}}; // Device
sycl::context ctx {dev}; // Context
sycl::queue queue {dev}; // Queue
std::vector<sycl::event> dependencies; // Input events dependencies
// Source
float *buf_src = static_cast<float *>(
        sycl::malloc_device((N * C * H * W) * sizeof(float), dev, ctx));
// Results
float *buf_dst = static_cast<float *>(
        sycl::malloc_device((N * C * H * W) * sizeof(float), dev, ctx));

// Create an engine encapsulating users' DPC++ GPU device and context
dnnl::engine engine = dnnl::sycl_interop::make_engine(dev, ctx);
// Create a stream encapsulating users' DPC++ GPU queue
dnnl::stream stream = dnnl::sycl_interop::make_stream(engine, queue);
// Create memory objects that use buf_src and buf_dst as the underlying storage
dnnl::memory mem_src({{N, C, H, W}, dnnl::memory::data_type::f32,
                              dnnl::memory::format_tag::nhwc},
        engine, buf_src);
dnnl::memory mem_dst({{N, C, H, W}, dnnl::memory::data_type::f32,
                              dnnl::memory::format_tag::nhwc},
        engine, buf_dst);
// Create a ReLU elementwise primitive
dnnl::eltwise_forward relu {
        {{dnnl::prop_kind::forward_inference, dnnl::algorithm::eltwise_relu,
```

(continues on next page)

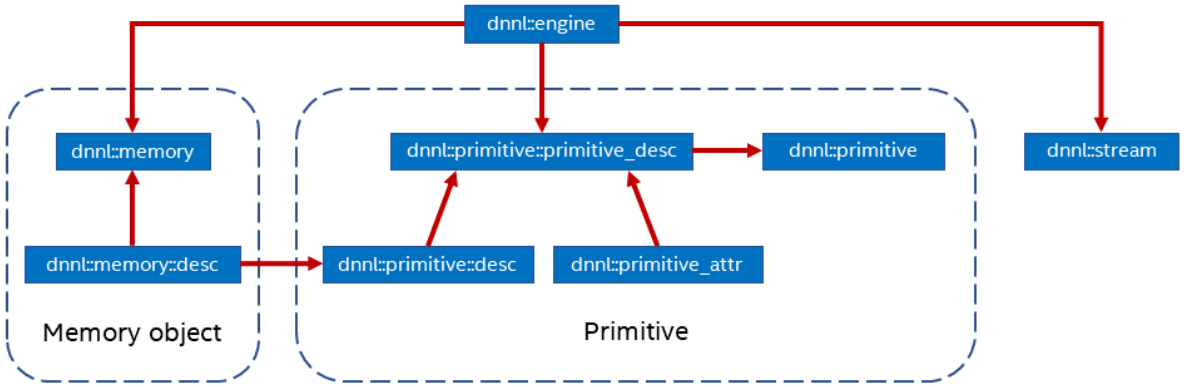
(continued from previous page)

```
                mem_src.get_desc(), 0.f, 0.f},
               engine}};
// Execute the ReLU primitive in the stream passing input dependencies and
// retrieving the output dependency
sycl::event event = dnnl::sycl_interop::execute(relu, stream,
        {{DNNL_ARG_SRC, mem_src}, {DNNL_ARG_DST, mem_dst}}, dependencies);
```

5.1 Introduction

Although the origins of this specification are in the existing open source implementation, its goal is to define a *portable* set of APIs. To this end, for example, it intentionally omits implementation-specific details like tiled or blocked memory formats (layouts), and instead describes plain multi-dimensional memory formats and defines opaque *optimized* memory format that can be implementation specific.

oneDNN main concepts are *primitives*, *engines*, *streams*, and *memory objects*.



A *primitive* (`dnnl::primitive`) is a functor object that encapsulates a particular computation such as forward convolution, backward LSTM computations, or a data transformation operation. A single primitive can sometimes represent more complex *fused* computations such as a forward convolution followed by a ReLU. Fusion, among other things, is controlled via the *primitive attributes* mechanism.

The most important difference between a primitive and a pure function is that a primitive can be specialized for a subset of input parameters.

For example, a convolution primitive stores parameters like tensor shapes and can pre-compute other dependent parameters like cache blocking. This approach allows oneDNN primitives to pre-generate code specifically tailored for the requested operation to be performed. The oneDNN programming model assumes that the time it takes to perform the pre-computations is amortized by reusing the same primitive to perform computations multiple times.

A primitive may also need a mutable memory buffer that it may use for temporary storage only during computations. Such buffer is called a scratchpad. It can either be owned by a primitive object (which makes that object non-thread safe) or be an execution-time parameter.

Primitive creation is a potentially expensive operation. Users are expected to create primitives once and reuse them multiple times. Alternatively, implementations may reduce the primitive creation cost by caching primitives that have the same parameters. This optimization falls outside of the scope of this specification.

Engines (`dnnl::engine`) are an abstraction of a computational device: a CPU, a specific GPU card in the system, etc. Most primitives are created to execute computations on one specific engine. The only exceptions are reorder primitives that may transfer data between two different engines.

Streams (`dnnl::stream`) encapsulate execution context tied to a particular engine. For example, they can correspond to DPC++ command queues.

Memory objects (`dnnl::memory`) encapsulate handles to memory allocated on a specific engine, tensor dimensions, data type, and memory format – the way tensor indices map to offsets in linear memory space. Memory objects are passed to primitives during execution.

Levels of Abstraction

oneDNN has multiple levels of abstractions for primitives and memory objects in order to expose maximum flexibility to its users.

On the logical level, the library provides the following abstractions:

- Memory descriptors (`dnnl::memory::desc`) define the logical dimensions of a tensor, data type, and the format in which the data is laid out in memory. The special format any (`dnnl::memory::format_tag::any`) indicates that the actual format will be defined later.
- Operation descriptors (one for each supported primitive) describe the most basic properties of an operation without specifying, for example, which engine will be used to compute them. For example, convolution descriptor describes shapes of source, destination, and weights tensors, propagation kind (forward, backward with respect to data or weights), and other implementation-independent parameters.
- Primitive descriptors (`dnnl::primitive_desc_base` is the base class and each of the supported primitives have their own version) are at an abstraction level in between operation descriptors and primitives and can be used to inspect details of a specific primitive implementation like expected memory formats via queries to implement memory format propagation (see *Memory format propagation*) without having to fully instantiate a primitive.

Abstraction level	Memory object	Primitive objects
Logical description	Memory descriptor	Operation descriptor
Intermediate description	N/A	Primitive descriptor
Implementation	Memory object	Primitive

5.1.1 General API notes

There are certain assumptions on how oneDNN objects behave:

- Memory and operation descriptors behave similarly to trivial types.
- All other objects behave like shared pointers. Copying is always shallow.

oneDNN objects can be *empty* in which case they are not valid for any use. Memory descriptors are special in this regard, as their empty versions are regarded as *zero* memory descriptors that can be used to indicate absence of a memory descriptor. Empty objects are usually created using default constructors, but also may be a result of an error during object construction (see the next section).

5.1.2 Error Handling

All oneDNN functions throw the following exception in case of error.

struct error : **public** exception
The exception class.

Additionally, many oneDNN functions that construct or return oneDNN objects have a boolean `allow_empty` parameter that defaults to `false` and that makes the library to return an empty object (a zero object in case of memory descriptors) when an object cannot be constructed instead of throwing an error.

5.1.3 Namespaces

All oneDNN functions and classes reside in `::dnnl` namespace. The functions that accept or return DPC++ objects such as command queues or buffers reside in `::dnnl::sycl_interop` namespace.

Furthermore, oneDNN defines `::oneapi::dnnl` namespace, that is an alias for the `::dnnl` namespace.

5.2 Conventions

oneDNN specification relies on a set of standard naming conventions for variables. This section describes these conventions.

5.2.1 Variable (Tensor) Names

Neural network models consist of operations of the following form:

$$\mathrm{dst} = f(\mathrm{src}, \mathrm{weights}),$$

where dst and src are activation tensors, and $\mathrm{weights}$ are learnable tensors.

The backward propagation therefore consists in computing the gradients with respect to the srcweights` respectively:

$$\mathrm{diff\_src} = \mathrm{d}f_{\mathrm{src}}(\mathrm{diff\_dst}, \mathrm{src}, \mathrm{weights}, \mathrm{dst}),$$

and

$$\mathrm{diff\_weights} = \mathrm{d}f_{\mathrm{weights}}(\mathrm{diff\_dst}, \mathrm{src}, \mathrm{weights}, \mathrm{dst}).$$

While oneDNN uses *src*, *dst*, and *weights* as generic names for the activations and learnable tensors, for a specific operation there might be commonly used and widely known specific names for these tensors. For instance, the *convolution* operation has a learnable tensor called *bias*. For usability reasons, oneDNN primitives use such names in initialization and other functions.

oneDNN uses the following commonly used notations for tensors:

Name	Meaning
`src`	Source tensor
`dst`	Destination tensor
`weights`	Weights tensor
`bias`	Bias tensor (used in *convolution*, *inner product* and other primitives)
`scale_shift`	Scale and shift tensors (used in *Batch Normalization* and *Layer normalization* primitives)
`workspace`	Workspace tensor that carries additional information from the forward propagation to the backward propagation
`scratchpad`	Temporary tensor that is required to store the intermediate results
`diff_src`	Gradient tensor with respect to the source
`diff_dst`	Gradient tensor with respect to the destination
`diff_weights`	Gradient tensor with respect to the weights
`diff_bias`	Gradient tensor with respect to the bias
`diff_scale_shift`	Gradient tensor with respect to the scale and shift
`*_layer`	RNN layer data or weights tensors
`*_iter`	RNN recurrent data or weights tensors

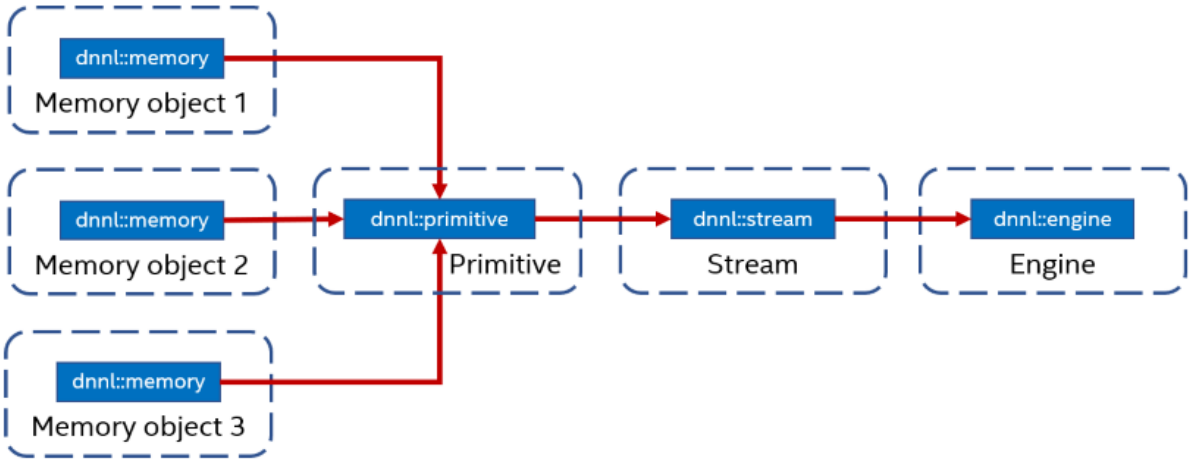
5.2.2 RNN-Specific Notation

The following notations are used when describing RNN primitives.

Name	Semantics
$\cdot$	matrix multiply operator
$*$	elementwise multiplication operator
W	input weights
U	recurrent weights
$\square^T$	transposition
B	bias
h	hidden state
a	intermediate value
x	input
$\square_t$	timestamp index
$\square_l$	layer index
activation	tanh, relu, logistic
c	cell state
$\tilde{c}$	candidate state
i	input gate
f	forget gate
o	output gate
u	update gate
r	reset gate

5.3 Execution Model

To execute a primitive, a user needs to pass memory arguments and a stream to the *dnnl::primitive::execute()* member function.



The primitive's computations are executed on the computational device corresponding to the engine on which the primitive (and memory arguments) were created and happens within the context of the stream.

5.3.1 Engine

Engine is abstraction of a computational device: a CPU, a specific GPU card in the system, etc. Most primitives are created to execute computations on one specific engine. The only exceptions are reorder primitives that transfer data between two different engines.

Engines correspond to and can be constructed from pairs of the DPC++ `sycl::device` and `sycl::context` objects. Alternatively, oneDNN itself can create and own the corresponding objects.

struct `dnnl::engine`
An execution engine.

Public Types

enum kind
Kinds of engines.

Values:

enumerator any
An unspecified engine.

enumerator cpu
CPU engine.

enumerator gpu
GPU engine.

Public Functions

engine()
Constructs an empty engine. An empty engine cannot be used in any operations.

engine(*kind akind*, size_t *index*)
Constructs an engine.

Parameters

- `akind`: The kind of engine to construct.
- `index`: The index of the engine. Must be less than the value returned by *get_count()* for this particular kind of engine.

kind **get_kind**() **const**
Returns the kind of the engine.

Return The kind of the engine.

Public Static Functions

size_t **get_count**(*kind akind*)
Returns the number of engines of a certain kind.

Return The number of engines of the specified kind.

Parameters

- `akind`: The kind of engines to count.

engine dnnl::sycl_interop::**make_engine**(**const** cl::sycl::device &*adevice*, **const** cl::sycl::context &*acontext*)
Creates an engine object using a specified SYCL device and SYCL context objects.

Return Engine object for the `adevice` SYCL device, within the specified `acontext` SYCL context.

Parameters

- `adevice`: SYCL device.
- `acontext`: SYCL context.

cl::sycl::device dnnl::sycl_interop::**get_device**(**const** *engine* &*aengine*)
Returns the SYCL device underlying a specified engine object.

Return SYCL device object underlying the `aengine` engine object.

Parameters

- `aengine`: Engine object.

cl::sycl::context dnnl::sycl_interop::**get_context**(**const** *engine* &*aengine*)
Returns the SYCL context underlying a specified engine object.

Return SYCL context object underlying the `aengine` engine object.

Parameters

- `aengine`: Engine object.

5.3.2 Stream

A *stream* is an encapsulation of execution context tied to a particular engine. They are passed to *dnnl::primitive::execute()* when executing a primitive.

Streams correspond to and can be constructed from DPC++ `sycl::queue` objects. Alternatively, oneDNN itself can create and own the corresponding objects. Streams are considered to be ephemeral and can be created / destroyed as long these operation do not violate DPC++ synchronization requirements.

Similar to DPC++ queues, streams can be in-order and out-of-order (see the relevant portion of the DPC++ specification for the explanation). The desired behavior can be specified using *dnnl::stream::flags* value. A stream created from a DPC++ queue inherits its behavior.

struct dnnl::**stream**

An execution stream.

Public Types

enum flags

Stream flags. Can be combined using the bitwise OR operator.

Values:

enumerator in_order

In-order execution.

enumerator out_of_order

Out-of-order execution.

enumerator default_flags

Default stream configuration.

Public Functions

stream()

Constructs an empty stream. An empty stream cannot be used in any operations.

stream(**const** *engine* &*aengine*, *flags* *aflags* = *flags*::*default_flags*)

Constructs a stream for the specified engine and with behavior controlled by the specified flags.

Parameters

- `aengine`: Engine to create the stream on.
- `aflags`: Flags controlling stream behavior.

stream &**wait**()

Waits for all primitives executing in the stream to finish.

Return The stream itself.

stream dnnl::sycl_interop::**make_stream**(**const** *engine* &*aengine*, cl::sycl::queue &*aqueue*)

Creates a stream for a specified engine and SYCL queue objects.

Return Stream object for the `aengine` engine object, which holds the `aqueue` SYCL queue object.

Parameters

- `aengine`: Engine object to use for the stream.

- `aqueue`: SYCL queue to use for the stream.

cl::sycl::queue `dnnl::sycl_interop::`**`get_queue`**`(`**`const`** *stream* *&astream*`)`
Returns the SYCL queue underlying a specified stream object.

Return SYCL queue underlying the `astream` stream object.

Parameters

- `astream`: Stream object.

5.4 Data model

Data in oneDNN is stored in *memory objects* that both store and describe data that can be of various types and be stored in different formats (layouts).

5.4.1 Data types

oneDNN supports multiple data types. However, the 32-bit IEEE single-precision floating-point data type is the fundamental type in oneDNN. It is the only data type that must be supported by an implementation. All the other types discussed below are optional.

Primitives operating on the single-precision floating-point data type consume data, produce, and store intermediate results using the same data type.

Moreover, single-precision floating-point data type is often used for intermediate results in the mixed precision computations because it provides better accuracy. For example, the elementwise primitive and elementwise post-ops always use it internally.

oneDNN uses the following enumeration to refer to data types it supports:

`enum` `dnnl::memory::`**`data_type`**
Data type specification.

Values:

`enumerator undef`
Undefined data type (used for empty memory descriptors).

`enumerator f16`
16-bit/half-precision floating point.

`enumerator bf16`
non-standard 16-bit floating point with 7-bit mantissa.

`enumerator f32`
32-bit/single-precision floating point.

`enumerator s32`
32-bit signed integer.

`enumerator s8`
8-bit signed integer.

`enumerator u8`
8-bit unsigned integer.

oneDNN supports training and inference with the following data types:

Usage mode	Data types
inference	*dnnl::memory::data_type::f32*, *dnnl::memory::data_type::bf16*, *dnnl::memory::data_type::f16*, *dnnl::memory::data_type::s8*/*dnnl::memory::data_type::u*
training	*dnnl::memory::data_type::f32*, *dnnl::memory::data_type::bf16*

Note: Using lower precision arithmetic may require changes in the deep learning model implementation.

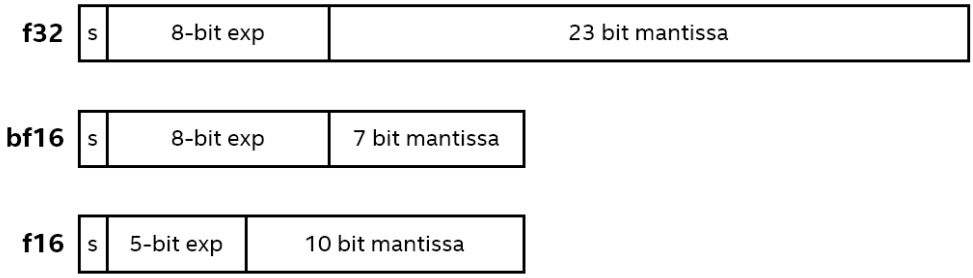
Individual primitives may have additional limitations with respect to data type support based on the precision requirements. The list of data types supported by each primitive is included in the corresponding sections of the specification guide.

Bfloat16

Note: In this section we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Bfloat16 (*bf16*) is a 16-bit floating point data type based on the IEEE 32-bit single-precision floating point data type (*f32*).

Both *bf16* and *f32* have an 8-bit exponent. However, while *f32* has a 23-bit mantissa, *bf16* has only a 7-bit one, keeping only the most significant bits. As a result, while these data types support a very close numerical range of values, *bf16* has a significantly reduced precision. Therefore, *bf16* occupies a spot between *f32* and the IEEE 16-bit half-precision floating point data type, *f16*. Compared directly to *f16*, which has a 5-bit exponent and a 10-bit mantissa, *bf16* trades increased range for reduced precision.



More details of the bfloat16 data type can be found here.

One of the advantages of using *bf16* versus *f32* is reduced memory footprint and, hence, increased memory access throughput.

Workflow

The main difference between implementing training with the *f32* data type and with the *bf16* data type is the way the weights updates are treated. With the *f32* data type, the weights gradients have the same data type as the weights themselves. This is not necessarily the case with the *bf16* data type as oneDNN allows some flexibility here. For example, one could maintain a master copy of all the weights, computing weights gradients in *f32* and converting the result to *bf16* afterwards.

Support

Most of the primitives can support the *bf16* data type for source and weights tensors. Destination tensors can be specified to have either the *bf16* or *f32* data type. The latter is intended for cases in which the output is to be fed to operations that do not support bfloat16 or require higher precision.

Int8

To push higher performance during inference computations, recent work has focused on computations that use activations and weights stored at lower precision to achieve higher throughput. Int8 computations offer improved performance over higher-precision types because they enable packing more computations into a single instruction, at the cost of reduced (but acceptable) accuracy.

Workflow

The *Quantization* describes what kind of quantization model oneDNN supports.

Support

oneDNN supports int8 computations for inference by allowing to specify that primitives input and output memory objects use int8 data types.

5.4.2 Memory

There are two levels of abstraction for memory in oneDNN.

1. *Memory descriptor* – engine-agnostic logical description of data (number of dimensions, dimension sizes, *data type*, and *format*.
2. *Memory object* – an engine-specific object combines memory descriptor with storage.

oneDNN defines the following convenience aliases to denote tensor dimensions

using dnnl::*memory*::**dim** = int64_t
Integer type for representing dimension sizes and indices.

using dnnl::*memory*::**dims** = std::vector<*dim*>
Vector of dimensions. Implementations are free to force a limit on the vector's length.

Memory Formats

In oneDNN memory format is how a multidimensional tensor is stored in 1-dimensional linear memory address space. oneDNN specifies two kinds of memory formats: *plain* which correspond to traditional multidimensional arrays, and *optimized* which are completely opaque.

Plain Memory Formats

Plain memory formats describe how multidimensional tensors are laid out in memory using an array of $\mathrm{dimensions}$ and an array of $\mathrm{strides}$ both of which have length equal to the rank of the tensor. In oneDNN the order of dimensions is fixed and different dimensions can have certain canonical interpretation depending on the primitive. For example, for CNN primitives the order for activation tensors is $\{N, C, ..., D, H, W\}$, where N stands for minibatch (or batch size), C stands for channels, and D, H, and W stand for image spatial dimensions: depth, height and width respectively. Spatial dimensions may be omitted in the order from outermost to innermost; for example, it is not possible to omit H when D is present and it is never possible to omit W. Canonical interpretation is documented for each primitive. However, this means that the $\mathrm{strides}$ array plays an important role defining the order in which different dimension are laid out in memory. Moreover, the $\mathrm{strides}$ need to agree with $\mathrm{dimensions}$.

More precisely, let T be a tensor of rank n and let σ be the permutation of the $\mathrm{strides}$ array that sorts it, i.e. $\mathrm{strides}[i] \geq \mathrm{strides}[j]$ if $\sigma(i) < \sigma(j)$ for all $0 \leq i, j < n$. Then the following must hold:

$$\mathrm{stides}[i] \geq \mathrm{strides}[j] * \mathrm{dimensions}[j] \text{ if } \sigma(i) < \sigma(j) \text{ for all } 0 \leq i, j < n.$$

For an element with coordinates $(i_0, \ldots, i_{n-1})$ such that $0 \leq i_j < \mathrm{dimensions}[j]$ for $0 \leq j < n$, its offset in memory is computed as:

$$\mathrm{offset}(i_0, \ldots, i_{n-1}) = \mathrm{offset}_0 + \sum_{j=0}^{n-1} i_j * \mathrm{strides}[j].$$

Here offset_0 is the offset from the *parent* memory and is non-zero only for *submemory* memory descriptors created using `dnnl::memory::desc::submemory_desc()`. Submemory memory descriptors inherit strides from the parent memory descriptor. Their main purpose is to express in-place concat operations.

As an example, consider an $M \times N$ matrix A (M rows times N columns). Regardless of whether A is stored transposed or not, $\mathrm{dimensions}_A = \{M, N\}$. However, $\mathrm{strides}_A = \{LDA, 1\}$ if it is not transposed and $\mathrm{strides}_A = \{1, LDA\}$ if it is, where LDA is such that $LDA \geq N$ if A is not transposed, and $LDA \geq M$ if it is. This also shows that A does not have to be stored *densly* in memory.

Note: The example above shows that oneDNN assumes data to be stored in row-major order.

Code example:

```
int M, N;
dnnl::memory::dims dims {M, N}; // Dimensions always stay the same

// Non-transposed matrix
dnnl::memory::dims strides_non_transposed {N, 1};
dnnl::memory::desc A_non_transposed {dims, dnnl::memory::data_type::f32,
        strides_non_transposed};

// Transposed matrix
dnnl::memory::dims strides_transposed {1, M};
dnnl::memory::desc A_transposed {dims, dnnl::memory::data_type::f32,
        strides_transposed};
```

Format Tags

In addition to strides, oneDNN provides named *format tags* via the *dnnl::memory::format_tag* enum type. The enumerators of this type can be used instead of strides for dense plain layouts.

The format tag names for N-dimensional memory formats use first N letters of the English alphabet which can be arbitrarily permuted. This permutation is used to compute strides for tensors with up to 6 dimensions. The resulting strides specify dense storage, in other words, using the nomenclature from the previous section, the following equality holds:

$$\text{stides}[i] = \text{strides}[j] * \text{dimensions}[j] \text{ if } \sigma(i) + 1 = \sigma(j) \text{ for all } 0 \leq i, j < n - 1.$$

In the matrix example, we could have used *dnnl::memory::format_tag::ab* for the non-transposed matrix above, and *dnnl::memory::format_tag::ba* for the transposed:

```
int M, N;
dnnl::memory::dims dims {M, N}; // Dimensions always stay the same

// Non-transposed matrix
dnnl::memory::desc A_non_transposed {dims, dnnl::memory::data_type::f32,
        dnnl::memory::format_tag::ab};

// Transposed matrix
dnnl::memory::desc A_transposed {dims, dnnl::memory::data_type::f32,
        dnnl::memory::format_tag::ba};
```

Note: In what follows in this section we abbreviate memory format tag names for readability. For example, *dnnl::memory::format_tag::abcd* is abbreviated to *abcd*.

In addition to abstract format tag names, oneDNN also provides convenience aliases. Some examples for CNNs and RNNs:

- *nchw* is an alias for *abcd* (see the canonical order order of dimensions for CNNs discussed above).
- *oihw* is an alias for *abcd*.
- *nhwc* is an alias for *acdb*.
- *tnc* is an alias for *abc*.
- *ldio* is an alias for *abcd*.
- *ldoi* is an alias for *abdc*.

Optimized Format 'any'

Another kind of format that oneDNN supports is an opaque *optimized* memory format that cannot be created directly from strides and dimensions arrays. A memory descriptor for an optimized memory format can only be created by passing *any* when creating certain operation descriptors, using them to create corresponding primitive descriptors and then querying them for memory descriptors. Data in plain memory format should then be reordered into the data in optimized data format before computations. Since reorders are expensive, the optimized memory format needs to be propagated through computations graph.

Optimized formats can employ padding, blocking and other data transformations to keep data in layout optimal for a certain architecture. This means that it in general operations like *dnnl::memory::desc::permute_axes()* or *dnnl::memory::desc::submemory_desc()* may fail. It is in general incorrect to use product of dimension

sizes to calculate amount of memory required to store data: *dnnl::memory::desc::get_size()* must be used instead.

Memory Format Propagation

Memory format propagation is one of the central notions that needs to be well-understood to use oneDNN correctly.

Convolution and inner product primitives choose the memory format when you create them with the placeholder memory format *any* for input or output. The memory format chosen is based on different circumstances such as hardware and convolution parameters. Using the placeholder *any* memory format is the recommended practice for convolutions, since they are the most compute-intensive operations in most topologies where they are present.

Other primitives, such as Elementwise, LRN, batch normalization and other, on forward propagation should use the same memory format as the preceding layer thus propagating the memory format through multiple oneDNN primitives. This avoids unnecessary reorders which may be expensive and should be avoided unless a compute-intensive primitive requires a different format. For performance reasons, backward computations of such primitives requires consistent memory format with the corresponding forward computations. Hence, when initializing there primitives for backward computations you should use *dnnl::memory::format_tag::any* memory format tag as well.

Below is the short summary when to use and not to use memory format *any* during operation description initialization:

Primitive Kinds	Forward Propagation	Backward Propagation	No Propagation
Compute intensive: (De-)convolution, Inner product, RNN	Use *any*	Use *any*	N/A
Memory-bandwidth limited: Pooling, Layer and Batch Normalization, Local Response Normalization, Elementwise, Shuffle, Softmax	Use memory format from preceding layer for source tensors, and *any* for destination tensors	Use *any* for gradient tensors, and actual memory formats for data tensors	N/A
Memory-bandwidth limited: Reorder, Concat, Sum, Binary	N/A	N/A	Use memory format from preceding layer for source tensors, and *any* for destination tensors

Additional format synchronization is required between forward and backward propagation when running training workloads. This is achieved via the `hint_pd` arguments of primitive descriptor constructors for primitives that implement backward propagation.

API

enum dnnl::*memory*::**format_tag**
Memory format tag specification.

Memory format tags can be further divided into two categories:

- Domain-agnostic names, i.e. names that do not depend on the tensor usage in the specific primitive. These names use letters from `a` to `f` to denote logical dimensions and form the order in which the dimensions are laid in memory. For example, *dnnl::memory::format_tag::ab* is used to denote a 2D tensor where the second logical dimension (denoted as `b`) is the innermost, i.e. has stride = 1, and the first logical dimension (`a`) is laid out in memory with stride equal to the size of the second dimension. On the other hand, *dnnl::memory::format_tag::ba* is the transposed version of the same tensor: the outermost dimension (`a`) becomes the innermost one.

- Domain-specific names, i.e. names that make sense only in the context of a certain domain, such as CNN. These names are aliases to the corresponding domain-agnostic tags and used mostly for convenience. For example, *dnnl::memory::format_tag::nc* is used to denote 2D CNN activations tensor memory format, where the channels dimension is the innermost one and the batch dimension is the outermost one. Moreover, *dnnl::memory::format_tag::nc* is an alias for *dnnl::memory::format_tag::ab*, because for CNN primitives the logical dimensions of activations tensors come in order: batch, channels, spatial. In other words, batch corresponds to the first logical dimension (`a`), and channels correspond to the second one (`b`).

The following domain-specific notation applies to memory format tags:

- `'n'` denotes the mini-batch dimension
- `'c'` denotes a channels dimension
- When there are multiple channel dimensions (for example, in convolution weights tensor), `'i'` and `'o'` denote dimensions of input and output channels
- `'g'` denotes a groups dimension for convolution weights
- `'d'`, `'h'`, and `'w'` denote spatial depth, height, and width respectively

Values:

enumerator undef
Undefined memory format tag.

enumerator any
Placeholder memory format tag. Used to instruct the primitive to select a format automatically.

enumerator a
plain 1D tensor

enumerator ab
plain 2D tensor

enumerator ba
permuted 2D tensor

enumerator abc
plain 3D tensor

enumerator acb
permuted 3D tensor

enumerator bac
permuted 3D tensor

enumerator bca
permuted 3D tensor

enumerator cba
permuted 3D tensor

enumerator abcd
plain 4D tensor

enumerator abdc
permuted 4D tensor

enumerator acdb
permuted 4D tensor

enumerator bacd
permuted 4D tensor

enumerator bcda
permuted 4D tensor

enumerator cdba
permuted 4D tensor

enumerator dcab
permuted 4D tensor

enumerator abcde
plain 5D tensor

enumerator abdec
permuted 5D tensor

enumerator acbde
permuted 5D tensor

enumerator acdeb
permuted 5D tensor

enumerator bacde
permuted 5D tensor

enumerator bcdea
permuted 5D tensor

enumerator cdeba
permuted 5D tensor

enumerator decab
permuted 5D tensor

enumerator abcdef
plain 6D tensor

enumerator acbdef
plain 6D tensor

enumerator defcab
plain 6D tensor

enumerator x
1D tensor; an alias for *dnnl::memory::format_tag::a*

enumerator nc
2D CNN activations tensor; an alias for *dnnl::memory::format_tag::ab*

enumerator cn
2D CNN activations tensor; an alias for *dnnl::memory::format_tag::ba*

enumerator tn
2D RNN statistics tensor; an alias for *dnnl::memory::format_tag::ab*

enumerator nt
2D RNN statistics tensor; an alias for *dnnl::memory::format_tag::ba*

enumerator ncw
3D CNN activations tensor; an alias for *dnnl::memory::format_tag::abc*

enumerator nwc
3D CNN activations tensor; an alias for *dnnl::memory::format_tag::acb*

enumerator nchw
4D CNN activations tensor; an alias for *dnnl::memory::format_tag::abcd*

enumerator nhwc
4D CNN activations tensor; an alias for *dnnl::memory::format_tag::acdb*

enumerator chwn
4D CNN activations tensor; an alias for *dnnl::memory::format_tag::bcda*

enumerator ncdhw
5D CNN activations tensor; an alias for *dnnl::memory::format_tag::abcde*

enumerator ndhwc
5D CNN activations tensor; an alias for *dnnl::memory::format_tag::acdeb*

enumerator oi
2D CNN weights tensor; an alias for *dnnl::memory::format_tag::ab*

enumerator io
2D CNN weights tensor; an alias for *dnnl::memory::format_tag::ba*

enumerator oiw
3D CNN weights tensor; an alias for *dnnl::memory::format_tag::abc*

enumerator owi
3D CNN weights tensor; an alias for *dnnl::memory::format_tag::acb*

enumerator wio
3D CNN weights tensor; an alias for *dnnl::memory::format_tag::cba*

enumerator iwo
3D CNN weights tensor; an alias for *dnnl::memory::format_tag::bca*

enumerator oihw
4D CNN weights tensor; an alias for *dnnl::memory::format_tag::abcd*

enumerator hwio
4D CNN weights tensor; an alias for *dnnl::memory::format_tag::cdba*

enumerator ohwi
4D CNN weights tensor; an alias for *dnnl::memory::format_tag::acdb*

enumerator ihwo
4D CNN weights tensor; an alias for *dnnl::memory::format_tag::bcda*

enumerator iohw
4D CNN weights tensor; an alias for *dnnl::memory::format_tag::bacd*

enumerator oidhw
5D CNN weights tensor; an alias for *dnnl::memory::format_tag::abcde*

enumerator dhwio
5D CNN weights tensor; an alias for *dnnl::memory::format_tag::cdeba*

enumerator odhwi
5D CNN weights tensor; an alias for *dnnl::memory::format_tag::acdeb*

enumerator iodhw
5D CNN weights tensor; an alias for *dnnl::memory::format_tag::bacde*

enumerator idhwo
5D CNN weights tensor; an alias for *dnnl::memory::format_tag::bcdea*

enumerator goiw
4D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::abcd*

enumerator wigo
4D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::dcab*

enumerator goihw
5D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::abcde*

enumerator hwigo
5D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::decab*

enumerator giohw
5D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::acbde*

enumerator goidhw
6D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::abcdef*

enumerator giodhw
6D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::abcdef*

enumerator dhwigo
6D CNN weights tensor with groups; an alias for *dnnl::memory::format_tag::defcab*

enumerator tnc
3D RNN data tensor in the format (seq_length, batch, input channels).

enumerator ntc
3D RNN data tensor in the format (batch, seq_length, input channels).

enumerator ldnc
4D RNN states tensor in the format (num_layers, num_directions, batch, state channels).

enumerator ldigo
5D RNN weights tensor in the format (num_layers, num_directions, input_channels, num_gates, output_channels).

- For LSTM cells, the gates order is input, forget, candidate and output gate.
- For GRU cells, the gates order is update, reset and output gate.

enumerator ldgoi
5D RNN weights tensor in the format (num_layers, num_directions, num_gates, output_channels, input_channels).

- For LSTM cells, the gates order is input, forget, candidate and output gate.
- For GRU cells, the gates order is update, reset and output gate.

enumerator ldio
4D LSTM projection tensor in the format (num_layers, num_directions, num_channels_in_hidden_state, num_channels_in_recurrent_projection).

enumerator ldoi
4D LSTM projection tensor in the format (num_layers, num_directions, num_channels_in_recurrent_projection, num_channels_in_hidden_state).

enumerator ldgo
4D RNN bias tensor in the format (num_layers, num_directions, num_gates, output_channels).

- For LSTM cells, the gates order is input, forget, candidate and output gate.

- For GRU cells, the gates order is update, reset and output gate.

Memory Descriptors and Objects

Descriptors

Memory descriptor is an engine-agnostic logical description of data (number of dimensions, dimension sizes, and data type), and, optionally, the information about the physical format of data in memory. If this information is not known yet, a memory descriptor can be created with format tag set to *dnnl::memory::format_tag::any*. This allows compute-intensive primitives to chose the most appropriate format for the computations. The user is then responsible for reordering their data into the new format if the formats do not match. See *Memory Format Propagation*.

A memory descriptor can be initialized either by specifying dimensions, and memory format tag or strides for each of them.

User can query amount of memory required by a memory descriptor using the *dnnl::memory::desc::get_size()* function. The size of data in general cannot be computed as the product of dimensions multiplied by the size of the data type. So users are required to use this function for better code portability.

Two memory descriptors can be compared using the equality and inequality operators. The comparison is especially useful when checking whether it is necessary to reorder data from the user's data format to a primitive's format.

Along with ordinary memory descriptors with all dimensions being positive, oneDNN supports *zero-volume* memory descriptors with one or more dimensions set to zero. This is used to support the NumPy* convention. If a zero-volume memory is passed to a primitive, the primitive typically does not perform any computations with this memory. For example:

- The concatenation primitive would ignore all memory object with zeroes in the concatenation dimension / axis.
- A forward convolution with a source memory object with zero in the minibatch dimension would always produce a destination memory object with a zero in the minibatch dimension and perform no computations.
- However, a forward convolution with a zero in one of the weights dimensions is ill-defined and is considered to be an error by the library because there is no clear definition on what the output values should be.

Data handle of a zero-volume memory is never accessed.

API

struct dnnl::*memory*::**desc**
A memory descriptor.

Public Functions

desc()
Constructs a zero (empty) memory descriptor. Such a memory descriptor can be used to indicate absence of an argument.

desc(**const** *dims* &*adims*, *data_type* *adata_type*, *format_tag* *aformat_tag*, bool *allow_empty* = false)
Constructs a memory descriptor.

Note The logical order of dimensions corresponds to the `abc...` format tag, and the physical meaning of the dimensions depends both on the primitive that would operate on this memory and the operation context.

Parameters

- `adims`: Tensor dimensions.
- `adata_type`: Data precision/type.
- `aformat_tag`: Memory format tag.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be constructed. This flag is optional and defaults to false.

desc(**const** *dims* &*adims*, *data_type* *adata_type*, **const** *dims* &*strides*, bool *allow_empty* = false)

Constructs a memory descriptor by strides.

Note The logical order of dimensions corresponds to the `abc...` format tag, and the physical meaning of the dimensions depends both on the primitive that would operate on this memory and the operation context.

Parameters

- `adims`: Tensor dimensions.
- `adata_type`: Data precision/type.
- `strides`: Strides for each dimension.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be constructed. This flag is optional and defaults to false.

desc **submemory_desc**(**const** *dims* &*adims*, **const** *dims* &*offsets*, bool *allow_empty* = false) **const**

Constructs a memory descriptor for a region inside an area described by this memory descriptor.

Return A memory descriptor for the region.

Parameters

- `adims`: Sizes of the region.
- `offsets`: Offsets to the region from the encompassing memory object in each dimension.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be returned. This flag is optional and defaults to false.

desc **reshape**(**const** *dims* &*adims*, bool *allow_empty* = false) **const**

Constructs a memory descriptor by reshaping an existing one. The new memory descriptor inherits the data type.

The operation ensures that the transformation of the physical memory format corresponds to the transformation of the logical dimensions. If such transformation is impossible, the function either throws an exception (default) or returns a zero memory descriptor depending on the `allow_empty` flag.

The reshape operation can be described as a combination of the following basic operations:

i. Add a dimension of size `1`. This is always possible.

ii. Remove a dimension of size `1`.

iii. Split a dimension into multiple ones. This is possible only if the product of all tensor dimensions stays constant.

iv. Join multiple consecutive dimensions into a single one. This requires that the dimensions are dense in memory and have the same order as their logical counterparts.

- Here, 'dense' means: `stride for dim[i] == (stride for dim[i + 1]) * dim[i + 1]`;
- And 'same order' means: `i < j` if and only if `stride for dim[j] <= stride for dim[i]`.

Note Reshape may fail for optimized memory formats.

Return A new memory descriptor with new dimensions.

Parameters

- `adims`: New dimensions. The product of dimensions must remain constant.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be returned. This flag is optional and defaults to false.

desc **permute_axes**(**const** std::vector<int> &*permutation*, bool *allow_empty* = false) **const**

Constructs a memory descriptor by permuting axes in an existing one.

The physical memory layout representation is adjusted accordingly to maintain the consistency between the logical and physical parts of the memory descriptor. The new memory descriptor inherits the data type.

The logical axes will be permuted in the following manner:

```
for (i = 0; i < ndims(); i++)
    new_desc.dims()[permutation[i]] = dims()[i];
```

Example:

```
std::vector<int> permutation = {1, 0}; // swap the first and
                                       // the second axes
dnnl::memory::desc in_md(
        {2, 3}, data_type, memory::format_tag::ab);
dnnl::memory::desc expect_out_md(
        {3, 2}, data_type, memory::format_tag::ba);

assert(in_md.permute_axes(permutation) == expect_out_md);
```

Return A new memory descriptor with new dimensions.

Parameters

- `permutation`: Axes permutation.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be returned. This flag is optional and defaults to false.

memory::*dims* **dims**() **const**

Returns dimensions of the memory descriptor.

Potentially expensive due to the data copy involved.

Return A copy of the dimensions vector.

memory::*data_type* **data_type**() **const**

Returns the data type of the memory descriptor.

Return The data type.

size_t **get_size**() **const**
Returns size of the memory descriptor in bytes.

Return The number of bytes required to allocate a memory buffer for the memory object described by this memory descriptor.

bool **is_zero**() **const**
Checks whether the memory descriptor is zero (empty).

Return `true` if the memory descriptor describes an empty memory and `false` otherwise.

bool **operator==**(**const** *desc* &*other*) **const**
An equality operator.

Return Whether this and the other memory descriptors have the same format tag, dimensions, strides, etc.

Parameters

- `other`: Another memory descriptor.

bool **operator!=**(**const** *desc* &*other*) **const**
An inequality operator.

Return Whether this and the other memory descriptors describe different memory.

Parameters

- `other`: Another memory descriptor.

Objects

Memory objects combine memory descriptors with storage for data (a data handle). With USM, the data handle is simply a pointer to `void`. The data handle can be queried using *dnnl::memory::get_data_handle()* and set using *dnnl::memory::set_data_handle()*. The underlying SYCL buffer, when used, can be queried using *dnnl::sycl_interop::get_buffer()* and set using *dnnl::sycl_interop::set_buffer()*. In addition, the memory descriptor and the engine underlying a memory object can be queried using *dnnl::memory::get_desc()* and *dnnl::memory::get_engine()* respectively.

API

struct dnnl::**memory**
Memory object.

A memory object encapsulates a handle to a memory buffer allocated on a specific engine, tensor dimensions, data type, and memory format, which is the way tensor indices map to offsets in linear memory space. Memory objects are passed to primitives during execution.

Public Functions

memory()
Default constructor.

Constructs an empty memory object, which can be used to indicate absence of a parameter.

memory(**const** *desc* &*md*, **const** *engine* &*aengine*, void **handle*)
Constructs a memory object.

Unless `handle` is equal to *DNNL_MEMORY_NONE*, the constructed memory object will have the underlying buffer set. In this case, the buffer will be initialized as if *dnnl::memory::set_data_handle()* has been called.

See *memory::set_data_handle()*

Parameters

- `md`: Memory descriptor.
- `aengine`: Engine to store the data on.
- `handle`: Handle of the memory buffer to use.
  - A pointer to the user-allocated buffer. In this case the library doesn't own the buffer.
  - The *DNNL_MEMORY_ALLOCATE* special value. Instructs the library to allocate the buffer for the memory object. In this case the library owns the buffer and the memory allocation kind of the underlying buffer is *dnnl::sycl_interop::memory_kind::usm*.
  - *DNNL_MEMORY_NONE* to create *dnnl::memory* without an underlying buffer.

memory(**const** *desc* &*md*, **const** *engine* &*aengine*)

Constructs a memory object.

The underlying buffer for the memory will be allocated by the library. The memory allocation kind of the underlying buffer is *dnnl::sycl_interop::memory_kind::usm*.

Parameters

- `md`: Memory descriptor.
- `aengine`: Engine to store the data on.

desc **get_desc**() **const**

Returns the associated memory descriptor.

engine **get_engine**() **const**

Returns the associated engine.

void ***get_data_handle**() **const**

Returns the underlying memory buffer.

On the CPU engine, or when using USM, this is a pointer to the allocated memory.

void **set_data_handle**(void **handle*, **const** *stream* &*astream*) **const**

Sets the underlying memory buffer.

This function may write zero values to the memory specified by the `handle` if the memory object has a zero padding area. This may be time consuming and happens each time this function is called. The operation is always blocking and the stream parameter is a hint.

Note Even when the memory object is used to hold values that stay constant during the execution of the program (pre-packed weights during inference, for example), the function will still write zeroes to the padding area if it exists. Hence, the `handle` parameter cannot and does not have a const qualifier.

Parameters

- `handle`: Memory buffer to use. On the CPU engine or when USM is used, the memory buffer is a pointer to the actual data. It must have at least *dnnl::memory::desc::get_size()* bytes allocated.
- `astream`: Stream to use to execute padding in.

void **set_data_handle**(void **handle*) **const**
Sets the underlying memory buffer.

See documentation for *dnnl::memory::set_data_handle(void *, const stream &) const* for more information.

Parameters

- `handle`: Memory buffer to use. For the CPU engine, the memory buffer is a pointer to the actual data. It must have at least *dnnl::memory::desc::get_size()* bytes allocated.

template<typename **T** = void>
T ***map_data**() **const**
Maps a memory object and returns a host-side pointer to a memory buffer with a copy of its contents.

Mapping enables read/write directly from/to the memory contents for engines that do not support direct memory access.

Mapping is an exclusive operation - a memory object cannot be used in other operations until it is unmapped via *dnnl::memory::unmap_data()* call.

Note Any primitives working with the memory should be completed before the memory is mapped. Use *dnnl::stream::wait()* to synchronize the corresponding execution stream.

Note The map_data and unmap_data functions are provided mainly for debug and testing purposes and their performance may be suboptimal.

Return Pointer to the mapped memory.

Template Parameters

- `T`: Data type to return a pointer to.

void **unmap_data**(void **mapped_ptr*) **const**
Unmaps a memory object and writes back any changes made to the previously mapped memory buffer.

Note The map_data and unmap_data functions are provided mainly for debug and testing purposes and their performance may be suboptimal.

Parameters

- `mapped_ptr`: A pointer previously returned by *dnnl::memory::map_data()*.

enum dnnl::sycl_interop::**memory_kind**
Memory allocation kinds.

Values:

enumerator usm
USM memory allocation kind.

enumerator buffer
Buffer memory allocation kind.

memory dnnl::sycl_interop::**make_memory**(**const** *memory*::*desc* &*adesc*, **const** *engine* &*aengine*, *memory_kind akind*, void **ahandle* = DNNL_MEMORY_ALLOCATE)
Creates a memory object of a specified description and of a specified memory allocation kind, for a specified engine.

Note If `akind` is *dnnl::sycl_interop::memory_kind::buffer*, and `ahandle` is not *DNNL_MEMORY_ALLOCATE* or *DNNL_MEMORY_NONE*, an exception is thrown.

Return Memory object described by `adesc` memory descriptor, which has `akind` memory allocation kind, and is attached to the `aengine` engine.

Parameters

- `adesc`: Memory descriptor that describes the data.
- `aengine`: Engine to store the data on.
- `akind`: Memory allocation kind.
- `ahandle`: Handle of the memory data to use. This parameter is optional. By default, the underlying memory buffer is allocated internally, its memory allocation kind is *dnnl::sycl_interop::memory_kind::usm*, and the library owns the buffer. If `handle` is provided, the library does not own the buffer.

memory `dnnl::sycl_interop::`**`make_memory`**`(`**`const`** *memory*::*desc* *&adesc*, **`const`** *stream* *&astream*, *memory_kind akind*, void **ahandle* = DNNL_MEMORY_ALLOCATE)

Creates a memory object of a specified description and of a specified memory allocation kind, for a specified stream.

Note If `akind` is *dnnl::sycl_interop::memory_kind::buffer*, and `ahandle` is not *DNNL_MEMORY_ALLOCATE* or *DNNL_MEMORY_NONE*, an exception is thrown.

Return Memory object described by `adesc` memory descriptor, which has `akind` memory allocation kind, and used withing the `astream` stream.

Parameters

- `adesc`: Memory descriptor that describes the data.
- `astream`: Stream object where the data is used.
- `akind`: Memory allocation kind.
- `ahandle`: Handle of the memory data to use. This parameter is optional. By default, the underlying memory buffer is allocated internally, its memory allocation kind is *dnnl::sycl_interop::memory_kind::usm*, and the library owns the buffer. If `handle` is provided, the library does not own the buffer.

template<typename **`T`**, int **`ndims`**>
memory `dnnl::sycl_interop::`**`make_memory`**`(`**`const`** *memory*::*desc* *&adesc*, **`const`** *engine* *&aengine*, cl::sycl::buffer<*T*, *ndims*> *abuffer*)

Creates a memory object using a specified SYCL buffer.

Note When such memory object is created, it is implied that its memory allocation kind is *dnnl::sycl_interop::memory_kind::buffer*.

Return Memory object which holds a `abuffer` SYCL buffer described by the `adesc` memory descriptor and attached to the `aengine` engine.

Template Parameters

- `T`: Data type of the specified SYCL buffer.
- `ndims`: Number of dimensions of the specified SYCL buffer.

Parameters

- `adesc`: Memory descriptor that describes the data within the specified buffer.
- `aengine`: Engine to store the data on.
- `abuffer`: SYCL buffer.

template<typename **T**, int **ndims**>
memory `dnnl::sycl_interop::` **`make_memory`**`(` **`const`** *memory*::*desc* *&adesc*, **`const`** *stream* *&astream*, cl::sycl::buffer<*T*, *ndims*> *abuffer*`)`

Creates a memory object using a specified SYCL buffer.

Note When such memory object is created, it is implied that its memory allocation kind is *dnnl::sycl_interop::memory_kind::buffer*.

Return Memory object which holds a `abuffer` SYCL buffer described by the `adesc` memory descriptor and used within the `astream` stream.

Template Parameters

- `T`: Data type of the specified SYCL buffer.
- `ndims`: Number of dimensions of the specified SYCL buffer.

Parameters

- `adesc`: Memory descriptor that describes the data within the specified buffer.
- `astream`: Stream object where the data is used.
- `abuffer`: SYCL buffer.

memory_kind `dnnl::sycl_interop::` **`get_memory_kind`**`(` **`const`** *memory* *&amemory*`)`

Returns the memory allocation kind of a specified memory object.

Note The memory allocation kind of a memory object could be changed during its lifetime, by setting the USM handle or SYCL buffer of said memory object.

Return Memory allocation kind of the `amemory` memory object.

Parameters

- `amemory`: Memory object.

template<typename **T**, int **ndims**>
void `dnnl::sycl_interop::` **`set_buffer`**`(` *memory* *&amemory*, cl::sycl::buffer<*T*, *ndims*> *abuffer*`)`

Sets the SYCL buffer underlying a specified memory object.

Note By setting the SYCL buffer of a memory object its memory allocation kind will be changed to *dnnl::sycl_interop::memory_kind::buffer*.

Template Parameters

- `T`: Data type of the specified SYCL buffer.
- `ndims`: Number of dimensions of the specified SYCL buffer.

Parameters

- `amemory`: Memory object that will store the `abuffer` SYCL buffer.
- `abuffer`: SYCL buffer to be stored in the `amemory` memory object.

template<typename **T**, int **ndims**>

void dnnl::sycl_interop::**set_buffer**(*memory* &*amemory*, cl::sycl::buffer<*T*, *ndims*> *abuffer*, *stream* &*astream*)

Sets the SYCL buffer underlying a specified memory object in a specified stream.

Template Parameters

- `T`: Data type of the specified SYCL buffer.
- `ndims`: Number of dimensions of the specified SYCL buffer.

Parameters

- `amemory`: Memory object that will store the `abuffer` SYCL buffer.
- `abuffer`: SYCL buffer to be stored in the `amemory` memory object and used in the `astream` stream.
- `astream`: Stream object within which the `amemory` memory object is used.

template<typename **T**, int **ndims** = 1>
cl::sycl::buffer<*T*, *ndims*> dnnl::sycl_interop::**get_buffer**(**const** *memory* &*amemory*)

Returns the SYCL buffer underlying a specified memory object.

Return SYCL buffer of type `T` with `ndims` dimensions, underlying the `amemory` memory object.

Template Parameters

- `T`: Data type of the specified SYCL buffer.
- `ndims`: Number of dimensions of the specified buffer.

Parameters

- `amemory`: Memory object.

DNNL_MEMORY_NONE

Special pointer value that indicates that a memory object should not have an underlying buffer.

DNNL_MEMORY_ALLOCATE

Special pointer value that indicates that the library needs to allocate an underlying buffer for a memory object.

5.5 Primitives

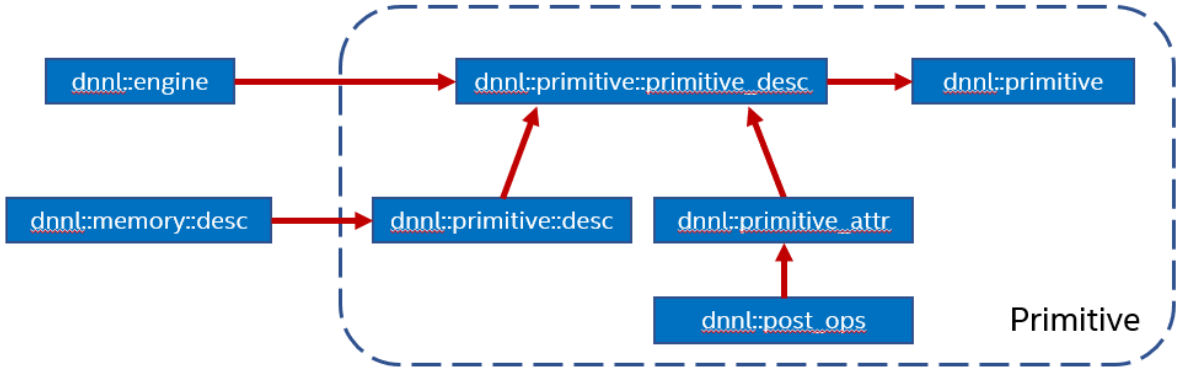
Primitives are functor objects that encapsulate a particular computation such as forward convolution, backward LSTM computations, or a data transformation operation. A single primitive can sometimes represent more complex fused computations such as a forward convolution followed by a ReLU.

The most important difference between a primitive and a pure function is that a primitive can store state.

One part of the primitive's state is immutable. For example, convolution primitives store parameters like tensor shapes and can pre-compute other dependent parameters like cache blocking. This approach allows oneDNN primitives to pre-generate code specifically tailored for the operation to be performed. The oneDNN programming model assumes that the time it takes to perform the pre-computations is amortized by reusing the same primitive to perform computations multiple times.

The mutable part of the primitive's state is referred to as a scratchpad. It is a memory buffer that a primitive may use for temporary storage only during computations. The scratchpad can either be owned by a primitive object (which makes that object non-thread safe) or be an execution-time parameter.

Conceptually, oneDNN establishes several layers of how to describe a computation from more abstract to more concrete:



- Operation descriptors (one for each supported primitive) describe an operation's most basic properties without specifying, for example, which engine will be used to compute them. For example, convolution descriptor describes shapes of source, destination, and weights tensors, propagation kind (forward, backward with respect to data or weights), and other implementation-independent parameters. The shapes are usually described as memory descriptors (*dnnl::memory::desc*).
- Primitive descriptors are at the abstraction level in between operation descriptors and primitives. They combine both an operation descriptor and primitive attributes. Primitive descriptors can be used to query various primitive implementation details and, for example, to implement *memory format propagation* by inspecting expected memory formats via queries without having to fully instantiate a primitive. oneDNN may contain multiple implementations for the same primitive that can be used to perform the same particular computation. Primitive descriptors allow one-way iteration which allows inspecting multiple implementations. The library is expected to order the implementations from most to least preferred, so it should always be safe to use the one that is chosen by default.
- Primitives, which are the most concrete, embody actual computations that can be executed.

On the API level:

- Primitives are represented as a class on the top level of the `dnnl` namespace that have *dnnl::primitive* as their base class, for example *dnnl::convolution_forward*
- Operation descriptors are represented as classes named `desc` and nested within the corresponding primitives classes, for example *dnnl::convolution_forward::desc*. The *dnnl::primitive_desc::next_impl()* member function provides a way to iterate over implementations.
- Primitive descriptors are represented as classes named `primitive_desc` and nested within the corresponding primitive classes that have *dnnl::primitive_desc_base* as their base class (except for RNN primitives that derive from *dnnl::rnn_primitive_desc_base*), for example *dnnl::convolution_forward::primitive_desc*

```
namespace dnnl {
   struct something_forward : public primitive {
      struct desc {
         // Primitive-specific constructors.
      }
      struct primitive_desc : public primitive_desc_base {
         // Constructors and primitive-specific memory descriptor queries.
      }
   };
}
```

The sequence of actions to create a primitive is:

1. Create an operation descriptor via, for example, `dnnl::convolution_forward::desc`. The operation descriptor can contain memory descriptors with placeholder `dnnl::memory::format_tag::any` memory formats if the primitive supports it.
2. Create a primitive descriptor based on the operation descriptor, engine and attributes.
3. Create a primitive based on the primitive descriptor obtained in step 2.

Note: Strictly speaking, not all the primitives follow this sequence. For example, the reorder primitive does not have an operation descriptor and thus does not require step 1 above.

5.5.1 Common Definitions

This section lists common types and definitions used by all or multiple primitives.

Base Class for Primitives

struct dnnl::**primitive**

Base class for all computational primitives.

Subclassed by *dnnl::batch_normalization_backward*, *dnnl::batch_normalization_forward*, *dnnl::binary*, *dnnl::concat*, *dnnl::convolution_backward_data*, *dnnl::convolution_backward_weights*, *dnnl::convolution_forward*, *dnnl::deconvolution_backward_data*, *dnnl::deconvolution_backward_weights*, *dnnl::deconvolution_forward*, *dnnl::eltwise_backward*, *dnnl::eltwise_forward*, *dnnl::gru_backward*, *dnnl::gru_forward*, *dnnl::inner_product_backward_data*, *dnnl::inner_product_backward_weights*, *dnnl::inner_product_forward*, *dnnl::layer_normalization_backward*, *dnnl::layer_normalization_forward*, *dnnl::lbr_gru_backward*, *dnnl::lbr_gru_forward*, *dnnl::logsoftmax_backward*, *dnnl::logsoftmax_forward*, *dnnl::lrn_backward*, *dnnl::lrn_forward*, *dnnl::lstm_backward*, *dnnl::lstm_forward*, *dnnl::matmul*, *dnnl::pooling_backward*, *dnnl::pooling_forward*, *dnnl::reorder*, *dnnl::resampling_backward*, *dnnl::resampling_forward*, *dnnl::shuffle_backward*, *dnnl::shuffle_forward*, *dnnl::softmax_backward*, *dnnl::softmax_forward*, *dnnl::sum*, *dnnl::vanilla_rnn_backward*, *dnnl::vanilla_rnn_forward*

Public Types

enum kind

Kinds of primitives supported by the library.

Values:

enumerator undef

Undefined primitive.

enumerator reorder

A reorder primitive.

enumerator shuffle

A shuffle primitive.

enumerator concat

A (out-of-place) tensor concatenation primitive.

enumerator sum

A summation primitive.

enumerator convolution
A convolution primitive.

enumerator deconvolution
A deconvolution primitive.

enumerator eltwise
An element-wise primitive.

enumerator softmax
A softmax primitive.

enumerator pooling
A pooling primitive.

enumerator lrn
An LRN primitive.

enumerator batch_normalization
A batch normalization primitive.

enumerator layer_normalization
A layer normalization primitive.

enumerator inner_product
An inner product primitive.

enumerator rnn
An RNN primitive.

enumerator binary
A binary primitive.

enumerator logsoftmax
A logsoftmax primitive.

enumerator matmul
A matmul (matrix multiplication) primitive.

enumerator resampling
A resampling primitive.

Public Functions

primitive()
Default constructor. Constructs an empty object.

primitive(**const** *primitive_desc_base* &*pd*)
Constructs a primitive from a primitive descriptor.

Parameters

- pd: Primitive descriptor.

kind **get_kind**() **const**
Returns the kind of the primitive.

Return The primitive kind.

void **execute**(**const** *stream* &*astream*, **const** std::unordered_map<int, *memory*> &*args*) **const**
Executes computations specified by the primitive in a specified stream.

Arguments are passed via an arguments map containing <index, memory object> pairs. The index must be one of the DNNL_ARG_* values such as DNNL_ARG_SRC, and the memory must have a memory descriptor matching the one returned by *dnnl::primitive_desc_base::query_md*(query::exec_arg_md, index) unless using dynamic shapes (see *DNNL_RUNTIME_DIM_VAL*).

Parameters

- astream: Stream object. The stream must belong to the same engine as the primitive.
- args: Arguments map.

primitive &**operator=**(**const** *primitive* &*rhs*)
Assignment operator.

cl::sycl::event dnnl::sycl_interop::**execute**(**const** *primitive* &*aprimitive*, **const** *stream* &*astream*, **const** std::unordered_map<int, *memory*> &*args*, **const** std::vector<cl::sycl::event> &*dependencies* = {})
Executes computations using a specified primitive object in a specified stream.

Arguments are passed via an arguments map containing <index, memory object> pairs. The index must be one of the DNNL_ARG_* values such as DNNL_ARG_SRC, and the memory must have a memory descriptor matching the one returned by *dnnl::primitive_desc_base::query_md*(query::exec_arg_md, index) unless using dynamic shapes (see *DNNL_RUNTIME_DIM_VAL*).

Return SYCL event object for the specified primitive execution.

Parameters

- aprimitive: Primitive to be executed.
- astream: Stream object. The stream must belong to the same engine as the primitive.
- args: Arguments map.
- dependencies: Vector of SYCL events that the execution depends on.

Base Class for Primitives Descriptors

There is no common base class for operation descriptors because they are very different between different primitives. However, there is a common base class for primitive descriptors.

struct dnnl::**primitive_desc_base**
Base class for all primitive descriptors.

Subclassed by *dnnl::concat::primitive_desc*, *dnnl::primitive_desc*, *dnnl::reorder::primitive_desc*, *dnnl::sum::primitive_desc*

Public Functions

primitive_desc_base()
Default constructor. Produces an empty object.

engine **get_engine**() **const**
Returns the engine of the primitive descriptor.

Return The engine of the primitive descriptor.

const char ***impl_info_str**() **const**
Returns implementation name.

Return The implementation name.

memory::*dim* **query_s64**(query *what*) **const**
Returns a *memory::dim* value (same as int64_t).

Return The result of the query.

Parameters

- `what`: The value to query.

memory::*desc* **query_md**(query *what*, int *idx* = 0) **const**
Returns a memory descriptor.

Note There are also convenience methods *dnnl::primitive_desc_base::src_desc()*, *dnnl::primitive_desc_base::dst_desc()*, and others.

Return The requested memory descriptor.

Return A zero memory descriptor if the primitive does not have a parameter of the specified kind or index.

Parameters

- `what`: The kind of parameter to query; can be dnnl::query::src_md, dnnl::query::dst_md, etc.
- `idx`: Index of the parameter. For example, convolution bias can be queried with what = dnnl::query::weights_md and idx = 1.

memory::*desc* **src_desc**(int *idx*) **const**
Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter with index `pdx`.

Parameters

- `idx`: Source index.

memory::*desc* **dst_desc**(int *idx*) **const**
Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter with index `pdx`.

Parameters

- `idx`: Destination index.

memory::*desc* **weights_desc**(int *idx*) **const**
Returns a weights memory descriptor.

Return Weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a weights parameter with index `pdx`.

Parameters

- `idx`: Weights index.

memory::*desc* **diff_src_desc**(int *idx*) **const**
Returns a diff source memory descriptor.

Return Diff source memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff source parameter with index `pdx`.

Parameters

- `idx`: Diff source index.

memory::*desc* **diff_dst_desc**(int *idx*) **const**
Returns a diff destination memory descriptor.

Return Diff destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination parameter with index `pdx`.

Parameters

- `idx`: Diff destination index.

memory::*desc* **diff_weights_desc**(int *idx*) **const**
Returns a diff weights memory descriptor.

Return Diff weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff weights parameter with index `pdx`.

Parameters

- `idx`: Diff weights index.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **weights_desc**() **const**
Returns a weights memory descriptor.

Return Weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **diff_src_desc**() **const**
Returns a diff source memory descriptor.

Return Diff source memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() **const**
Returns a diff destination memory descriptor.

Return Diff destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_weights_desc**() **const**
Returns a diff weights memory descriptor.

Return Diff weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.

Return Workspace memory descriptor.

Return A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **scratchpad_desc**() **const**
Returns the scratchpad memory descriptor.

Return scratchpad memory descriptor.

Return A zero memory descriptor if the primitive does not require scratchpad parameter.

engine **scratchpad_engine**() **const**
Returns the engine on which the scratchpad memory is located.

Return The engine on which the scratchpad memory is located.

primitive_attr **get_primitive_attr**() **const**
Returns the primitive attributes.

Return The primitive attributes.

dnnl::*primitive*::*kind* **get_kind**() **const**
Returns the kind of the primitive descriptor.

Return The kind of the primitive descriptor.

It is further derived from to provide base class for all primitives that have operation descriptors.

struct dnnl::**primitive_desc** : **public** dnnl::*primitive_desc_base*
A base class for descriptors of all primitives that have an operation descriptor and that support iteration over multiple implementations.

Subclassed by *dnnl::batch_normalization_backward::primitive_desc*, *dnnl::batch_normalization_forward::primitive_desc*, *dnnl::binary::primitive_desc*, *dnnl::convolution_backward_data::primitive_desc*, *dnnl::convolution_backward_weights::primitive_desc*, *dnnl::convolution_forward::primitive_desc*, *dnnl::deconvolution_backward_data::primitive_desc*, *dnnl::deconvolution_backward_weights::primitive_desc*, *dnnl::deconvolution_forward::primitive_desc*, *dnnl::eltwise_backward::primitive_desc*, *dnnl::eltwise_forward::primitive_desc*, *dnnl::inner_product_backward_data::primitive_desc*, *dnnl::inner_product_backward_weights::primitive_desc*, *dnnl::inner_product_forward::primitive_desc*, *dnnl::layer_normalization_backward::primitive_desc*, *dnnl::layer_normalization_forward::primitive_desc*, *dnnl::logsoftmax_backward::primitive_desc*, *dnnl::logsoftmax_forward::primitive_desc*, *dnnl::lrn_backward::primitive_desc*, *dnnl::lrn_forward::primitive_desc*, *dnnl::matmul::primitive_desc*, *dnnl::pooling_backward::primitive_desc*, *dnnl::pooling_forward::primitive_desc*, *dnnl::resampling_backward::primitive_desc*, *dnnl::resampling_forward::primitive_desc*, *dnnl::rnn_primitive_desc_base*, *dnnl::shuffle_backward::primitive_desc*, *dnnl::shuffle_forward::primitive_desc*, *dnnl::softmax_backward::primitive_desc*, *dnnl::softmax_forward::primitive_desc*

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

bool **next_impl**()
Advances the primitive descriptor iterator to the next implementation.

Return `true` on success, and `false` if the last implementation reached, in which case primitive descriptor is not modified.

The *dnnl::reorder*, *dnnl::sum* and *dnnl::concat* primitives also subclass *dnnl::primitive_desc* to implement their primitive descriptors.

RNN primitives further subclass the *dnnl::primitive_desc_base* to provide utility functions for frequently queried memory descriptors.

struct dnnl::**rnn_primitive_desc_base** : **public** dnnl::*primitive_desc*
Base class for primitive descriptors for RNN primitives.

Subclassed by *dnnl::gru_backward::primitive_desc*, *dnnl::gru_forward::primitive_desc*, *dnnl::lbr_gru_backward::primitive_desc*, *dnnl::lbr_gru_forward::primitive_desc*, *dnnl::lstm_backward::primitive_desc*, *dnnl::lstm_forward::primitive_desc*, *dnnl::vanilla_rnn_backward::primitive_desc*, *dnnl::vanilla_rnn_forward::primitive_desc*

Public Functions

rnn_primitive_desc_base()
Default constructor. Produces an empty object.

memory::*desc* **src_layer_desc**() **const**
Returns source layer memory descriptor.

Return Source layer memory descriptor.

memory::*desc* **src_iter_desc**() **const**
Returns source iteration memory descriptor.

Return Source iteration memory descriptor.

Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **src_iter_c_desc**() **const**
Returns source recurrent cell state memory descriptor.

Return Source recurrent cell state memory descriptor.

memory::*desc* **weights_layer_desc**() **const**
Returns weights layer memory descriptor.

Return Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() **const**
Returns weights iteration memory descriptor.

Return Weights iteration memory descriptor.

memory::*desc* **weights_peephole_desc**() **const**
Returns weights peephole memory descriptor.

Return Weights peephole memory descriptor.

memory::*desc* **weights_projection_desc**() **const**
Returns weights projection memory descriptor.

Return Weights projection memory descriptor.

memory::*desc* **bias_desc**() **const**
Returns bias memory descriptor.

Return Bias memory descriptor.

Return A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() **const**
Returns destination layer memory descriptor.

Return Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() **const**
Returns destination iteration memory descriptor.

Return Destination iteration memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **dst_iter_c_desc**() **const**
Returns destination recurrent cell state memory descriptor.

Return Destination recurrent cell state memory descriptor.

memory::*desc* **diff_src_layer_desc**() **const**
Returns diff source layer memory descriptor.

Return Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() **const**
Returns diff source iteration memory descriptor.

Return Diff source iteration memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_src_iter_c_desc**() **const**
Returns diff source recurrent cell state memory descriptor.

Return Diff source recurrent cell state memory descriptor.

memory::*desc* **diff_weights_layer_desc**() **const**
Returns diff weights layer memory descriptor.

Return Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() **const**
Returns diff weights iteration memory descriptor.

Return Diff weights iteration memory descriptor.

memory::*desc* **diff_weights_peephole_desc**() **const**
Returns diff weights peephole memory descriptor.

Return Diff weights peephole memory descriptor.

memory::*desc* **diff_weights_projection_desc**() **const**
Returns diff weights projection memory descriptor.

Return Diff weights projection memory descriptor.

memory::*desc* **diff_bias_desc**() **const**
Returns diff bias memory descriptor.

Return Diff bias memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() **const**
Returns diff destination layer memory descriptor.

Return Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() **const**
Returns diff destination iteration memory descriptor.

Return Diff destination iteration memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

memory::*desc* **diff_dst_iter_c_desc**() **const**
Returns diff destination recurrent cell state memory descriptor.

Return Diff destination recurrent cell state memory descriptor.

Common Enumerations

enum dnnl::**prop_kind**
Propagation kind.

Values:

enumerator undef
Undefined propagation kind.

enumerator forward_training
Forward data propagation (training mode). In this mode, primitives perform computations necessary for subsequent backward propagation.

enumerator forward_inference
Forward data propagation (inference mode). In this mode, primitives perform only computations that are necessary for inference and omit computations that are necessary only for backward propagation.

enumerator forward_scoring
Forward data propagation, alias for *dnnl::prop_kind::forward_inference*.

enumerator forward
Forward data propagation, alias for *dnnl::prop_kind::forward_training*.

enumerator backward
Backward propagation (with respect to all parameters).

enumerator backward_data
Backward data propagation.

enumerator backward_weights
Backward weights propagation.

enumerator backward_bias
Backward bias propagation.

enum dnnl::**algorithm**
Kinds of algorithms.

Values:

enumerator undef
Undefined algorithm.

enumerator convolution_auto
Convolution algorithm that is chosen to be either direct or Winograd automatically

enumerator convolution_direct
Direct convolution.

enumerator convolution_winograd
Winograd convolution.

enumerator deconvolution_direct
Direct deconvolution.

enumerator deconvolution_winograd
Winograd deconvolution.

enumerator eltwise_relu
Elementwise: rectified linear unit (ReLU)

enumerator eltwise_tanh
Elementwise: hyperbolic tangent non-linearity (tanh)

enumerator eltwise_elu
Elementwise: exponential linear unit (ELU)

enumerator eltwise_square
Elementwise: square.

enumerator eltwise_abs
Elementwise: abs.

enumerator eltwise_sqrt
Elementwise: square root.

enumerator eltwise_swish
Elementwise: swish ($x \cdot sigmoid(a \cdot x)$)

enumerator eltwise_linear
Elementwise: linear.

enumerator eltwise_bounded_relu
Elementwise: bounded_relu.

enumerator eltwise_soft_relu
Elementwise: soft_relu.

enumerator eltwise_logistic
Elementwise: logistic.

enumerator eltwise_exp
Elementwise: exponent.

enumerator eltwise_gelu
Elementwise: gelu alias for *dnnl::algorithm::eltwise_gelu_tanh*

enumerator eltwise_gelu_tanh
Elementwise: tanh-based gelu.

enumerator eltwise_gelu_erf
Elementwise: erf-based gelu.

enumerator eltwise_log
Elementwise: natural logarithm.

enumerator eltwise_clip
Elementwise: clip.

enumerator eltwise_pow
Elementwise: pow.

enumerator eltwise_round
Elementwise: round.

enumerator eltwise_relu_use_dst_for_bwd
Elementwise: rectified linear unit (ReLU) (dst for backward)

enumerator eltwise_tanh_use_dst_for_bwd
Elementwise: hyperbolic tangent non-linearity (tanh) (dst for backward)

enumerator eltwise_elu_use_dst_for_bwd
Elementwise: exponential linear unit (ELU) (dst for backward)

enumerator eltwise_sqrt_use_dst_for_bwd
Elementwise: square root (dst for backward)

enumerator eltwise_logistic_use_dst_for_bwd
Elementwise: logistic (dst for backward)

enumerator eltwise_exp_use_dst_for_bwd
Elementwise: exponent (dst for backward)

enumerator lrn_across_channels
Local response normalization (LRN) across multiple channels.

enumerator lrn_within_channel
LRN within a single channel.

enumerator pooling_max
Max pooling.

enumerator pooling_avg
Average pooling exclude padding, alias for *dnnl::algorithm::pooling_avg_include_padding*

enumerator pooling_avg_include_padding
Average pooling include padding.

enumerator pooling_avg_exclude_padding
Average pooling exclude padding.

enumerator vanilla_rnn
RNN cell.

enumerator vanilla_lstm
LSTM cell.

enumerator vanilla_gru
GRU cell.

enumerator lbr_gru
GRU cell with linear before reset. Differs from original GRU in how the new memory gate is calculated: $c_t = tanh(W_c * x_t + b_{c_x} + r_t * (U_c * h_{t-1} + b_{c_h}))$ LRB GRU expects 4 bias tensors on input: $[b_u, b_r, b_{c_x}, b_{c_h}]$

enumerator binary_add
Binary add.

enumerator binary_mul
Binary mul.

enumerator binary_max
Binary max.

enumerator binary_min
Binary min.

enumerator resampling_nearest
Nearest Neighbor resampling method.

enumerator resampling_linear
Linear (Bilinear, Trilinear) resampling method.

Normalization Primitives Flags

enum dnnl::**normalization_flags**
Flags for normalization primitives (can be combined via '|')

Values:

enumerator none
Use no normalization flags. If specified, the library computes mean and variance on forward propagation for training and inference, outputs them on forward propagation for training, and computes the respective derivatives on backward propagation.

enumerator use_global_stats
Use global statistics. If specified, the library uses mean and variance provided by the user as an input on forward propagation and does not compute their derivatives on backward propagation. Otherwise, the library computes mean and variance on forward propagation for training and inference, outputs them on forward propagation for training, and computes the respective derivatives on backward propagation.

enumerator use_scale_shift
Use scale and shift parameters. If specified, the user is expected to pass scale and shift as inputs on forward propagation. On backward propagation of type *dnnl::prop_kind::backward*, the library computes their derivatives. If not specified, the scale and shift parameters are not used by the library in any way.

enumerator fuse_norm_relu
Fuse normalization with ReLU. On training, normalization will require the workspace to implement backward propagation. On inference, the workspace is not required and behavior is the same as when normalization is fused with ReLU using the post-ops API.

Execution argument indices

DNNL_ARG_SRC_0
Source argument #0.

DNNL_ARG_SRC
A special mnemonic for source argument for primitives that have a single source. An alias for *DNNL_ARG_SRC_0*.

DNNL_ARG_SRC_LAYER
A special mnemonic for RNN input vector. An alias for *DNNL_ARG_SRC_0*.

DNNL_ARG_FROM
A special mnemonic for reorder source argument. An alias for *DNNL_ARG_SRC_0*.

DNNL_ARG_SRC_1
Source argument #1.

DNNL_ARG_SRC_ITER
A special mnemonic for RNN input recurrent hidden state vector. An alias for *DNNL_ARG_SRC_1*.

DNNL_ARG_SRC_2
Source argument #2.

DNNL_ARG_SRC_ITER_C
A special mnemonic for RNN input recurrent cell state vector. An alias for *DNNL_ARG_SRC_2*.

DNNL_ARG_DST_0
Destination argument #0.

DNNL_ARG_DST
A special mnemonic for destination argument for primitives that have a single destination. An alias for *DNNL_ARG_DST_0*.

DNNL_ARG_TO
A special mnemonic for reorder destination argument. An alias for *DNNL_ARG_DST_0*.

DNNL_ARG_DST_LAYER
A special mnemonic for RNN output vector. An alias for *DNNL_ARG_DST_0*.

DNNL_ARG_DST_1
Destination argument #1.

DNNL_ARG_DST_ITER
A special mnemonic for RNN input recurrent hidden state vector. An alias for *DNNL_ARG_DST_1*.

DNNL_ARG_DST_2
Destination argument #2.

DNNL_ARG_DST_ITER_C
A special mnemonic for LSTM output recurrent cell state vector. An alias for *DNNL_ARG_DST_2*.

DNNL_ARG_WEIGHTS_0
Weights argument #0.

DNNL_ARG_WEIGHTS
A special mnemonic for primitives that have a single weights argument. Alias for *DNNL_ARG_WEIGHTS_0*.

DNNL_ARG_SCALE_SHIFT
A special mnemonic for scale and shift argument of normalization primitives. Alias for *DNNL_ARG_WEIGHTS_0*.

DNNL_ARG_WEIGHTS_LAYER
A special mnemonic for RNN weights applied to the layer input. An alias for *DNNL_ARG_WEIGHTS_0*.

DNNL_ARG_WEIGHTS_1
Weights argument #1.

DNNL_ARG_WEIGHTS_ITER
A special mnemonic for RNN weights applied to the recurrent input. An alias for *DNNL_ARG_WEIGHTS_1*.

DNNL_ARG_BIAS
Bias tensor argument.

DNNL_ARG_MEAN
Mean values tensor argument.

DNNL_ARG_VARIANCE
Variance values tensor argument.

DNNL_ARG_WORKSPACE
Workspace tensor argument. Workspace is used to pass information from forward propagation to backward propagation computations.

DNNL_ARG_SCRATCHPAD
Scratchpad (temporary storage) tensor argument.

DNNL_ARG_DIFF_SRC_0
Gradient (diff) of the source argument #0.

DNNL_ARG_DIFF_SRC
A special mnemonic for primitives that have a single diff source argument. An alias for *DNNL_ARG_DIFF_SRC_0*.

DNNL_ARG_DIFF_SRC_LAYER
A special mnemonic for gradient (diff) of RNN input vector. An alias for *DNNL_ARG_DIFF_SRC_0*.

DNNL_ARG_DIFF_SRC_1
Gradient (diff) of the source argument #1.

DNNL_ARG_DIFF_SRC_ITER
A special mnemonic for gradient (diff) of RNN input recurrent hidden state vector. An alias for *DNNL_ARG_DIFF_SRC_1*.

DNNL_ARG_DIFF_SRC_2
Gradient (diff) of the source argument #2.

DNNL_ARG_DIFF_SRC_ITER_C
A special mnemonic for gradient (diff) of RNN input recurrent cell state vector. An alias for *DNNL_ARG_DIFF_SRC_1*.

DNNL_ARG_DIFF_DST_0
Gradient (diff) of the destination argument #0.

DNNL_ARG_DIFF_DST
A special mnemonic for primitives that have a single diff destination argument. An alias for *DNNL_ARG_DIFF_DST_0*.

DNNL_ARG_DIFF_DST_LAYER
A special mnemonic for gradient (diff) of RNN output vector. An alias for *DNNL_ARG_DIFF_DST_0*.

DNNL_ARG_DIFF_DST_1
Gradient (diff) of the destination argument #1.

DNNL_ARG_DIFF_DST_ITER
A special mnemonic for gradient (diff) of RNN input recurrent hidden state vector. An alias for *DNNL_ARG_DIFF_DST_1*.

DNNL_ARG_DIFF_DST_2
Gradient (diff) of the destination argument #2.

DNNL_ARG_DIFF_DST_ITER_C
A special mnemonic for gradient (diff) of RNN input recurrent cell state vector. An alias for *DNNL_ARG_DIFF_DST_2*.

DNNL_ARG_DIFF_WEIGHTS_0
Gradient (diff) of the weights argument #0.

DNNL_ARG_DIFF_WEIGHTS
A special mnemonic for primitives that have a single diff weights argument. Alias for *DNNL_ARG_DIFF_WEIGHTS_0*.

DNNL_ARG_DIFF_SCALE_SHIFT
A special mnemonic for diff of scale and shift argument of normalization primitives. Alias for *DNNL_ARG_DIFF_WEIGHTS_0*.

DNNL_ARG_DIFF_WEIGHTS_LAYER
A special mnemonic for diff of RNN weights applied to the layer input. An alias for *DNNL_ARG_DIFF_WEIGHTS_0*.

DNNL_ARG_DIFF_WEIGHTS_1
Gradient (diff) of the weights argument #1.

DNNL_ARG_DIFF_WEIGHTS_ITER
A special mnemonic for diff of RNN weights applied to the recurrent input. An alias for *DNNL_ARG_DIFF_WEIGHTS_1*.

DNNL_ARG_DIFF_BIAS
Gradient (diff) of the bias tensor argument.

DNNL_ARG_ATTR_OUTPUT_SCALES
Output scaling factors provided at execution time.

DNNL_ARG_MULTIPLE_SRC
Starting index for source arguments for primitives that take a variable number of source arguments.

DNNL_ARG_MULTIPLE_DST
Starting index for destination arguments for primitives that produce a variable number of destination arguments.

DNNL_ARG_ATTR_ZERO_POINTS
Zero points provided at execution time.

DNNL_RUNTIME_DIM_VAL
A wildcard value for dimensions that are unknown at a primitive creation time.

DNNL_RUNTIME_SIZE_VAL
A `size_t` counterpart of the *DNNL_RUNTIME_DIM_VAL*. For instance, this value is returned by *dnnl::memory::desc::get_size()* if either of the dimensions or strides equal to *DNNL_RUNTIME_DIM_VAL*.

DNNL_RUNTIME_F32_VAL
A wildcard value for floating point values that are unknown at a primitive creation time.

DNNL_RUNTIME_S32_VAL
A wildcard value for int32_t values that are unknown at a primitive creation time.

5.5.2 Attributes

The parameters passed to create a primitive descriptor specify the problem. An engine specifies where the primitive will be executed. An operation descriptor specifies the basics: the operation kind; the propagation kind; the source, destination, and other tensors; the strides (if applicable); and so on.

Attributes specify some extra properties of the primitive. Users must create them before use and must set required specifics using the corresponding setters. The attributes are copied during primitive descriptor creation, so users can change or destroy attributes right after that.

If not modified, attributes can stay empty, which is equivalent to the default attributes. Primitive descriptors' constructors have empty attributes as default parameters, so, unless required, users can simply omit them.

Attributes can also contain *post-ops*, which are computations executed after the primitive.

Post-ops

Post-ops are operations that are appended after a primitive. They are implemented using the *Attributes* mechanism. If there are multiple post-ops, the are executed in the order they have been appended.

The post-ops are represented by `dnnl::post_ops` which is copied once it is attached to the attributes using `dnnl::primitive_attr::set_post_ops()` function. The attributes then need to be passed to a primitive descriptor creation function to take effect. Below is a simple sketch:

```
dnnl::post_ops po; // default empty post-ops
assert(po.len() == 0); // no post-ops attached

po.append_SOMETHING(params); // append some particular post-op
po.append_SOMETHING_ELSE(other_params); // append one more post-op

// (!) Note that the order in which post-ops are appended matters!
assert(po.len() == 2);

dnnl::primitive_attr attr; // default attributes
attr.set_post_ops(po); // attach the post-ops to the attr
// any changes to po after this point don't affect the value stored in attr

primitive::primitive_desc op_pd(params, attr); // create a pd with the attr
```

Note: Different primitives may have different post-ops support. Moreover, the support might also depend on the actual implementation of a primitive. So robust code should be able to handle errors accordingly. See the *Attribute Related Error Handling*.

Note: Post-ops do not change memory format of the operation destination memory object.

The post-op objects can be inspected using the `dnnl::post_ops::kind()` function that takes an index of the post-op to inspect (that must be less than the value returned by `dnnl::post_ops::len()`), and returns its kind.

Supported Post-ops

Eltwise Post-op

The eltwise post-op is appended using `dnnl::post_ops::append_eltwise()` function. The `dnnl::post_ops::kind()` returns `dnnl::primitive::kind::eltwise` for such a post-op.

The eltwise post-op replaces:

$$\text{dst}[:] = \text{Op}(...)$$

with

$$\text{dst}[:] = scale \cdot \text{eltwise}(\text{Op}(...))$$

The intermediate result of the $\text{Op}(...)$ is not preserved.

The $scale$ factor is supported in *int8* inference only. For all other cases the scale must be *1.0*.

Sum Post-op

The sum post-op accumulates the result of a primitive with the existing data and is appended using *dnnl::post_ops::append_sum()* function. The *dnnl::post_ops::kind()* returns *dnnl::primitive::kind::sum* for such a post-op.

Prior to accumulating the result, the existing value us multiplied by scale. The scale parameter can be used in The $scale$ factor is supported in *int8* inference only and should be used only when the result and the existing data have different magnitudes. For all other cases the scale must be *1.0*.

Additionally, the sum post-op can reinterpret the destination values as a different data type of the same size. This may be used to, for example, reinterpret 8-bit signed data as unsigned or vice versa (which requires that values fall within a common range to work).

The sum post-op replaces

$$\text{dst}[:] = \text{Op}(...)$$

with

$$\text{dst}[:] = scale \cdot as_data_type(\text{dst}[:]) + \text{Op}(...)$$

Examples of Chained Post-ops

Post-ops can be chained together by appending one after another. Note that the order matters: the post-ops are executed in the order they have been appended.

Sum -> ReLU

This pattern is pretty common for the CNN topologies of the ResNet family.

```
dnnl::post_ops po;
po.append_sum(
        /* scale = */ 1.f);
po.append_eltwise(
        /* scale     = */ 1.f,
        /* algorithm = */ dnnl::algorithm::eltwise_relu,
        /* neg slope = */ 0.f,
        /* unused for ReLU */ 0.f);

dnnl::primitive_attr attr;
attr.set_post_ops(po);

convolution_forward::primitive_desc(conv_d, attr, engine);
```

This will lead to the following computations:

$$\text{dst}[:] = \text{ReLU}(\text{dst}[:] + \text{conv}(\text{src}[:], \text{weights}[:])$$

API

struct dnnl::**post_ops**

Post-ops.

Post-ops are computations executed after the main primitive computations and are attached to the primitive via primitive attributes.

Public Functions

post_ops()

Constructs an empty sequence of post-ops.

int **len**() **const**

Returns the number of post-ops entries.

primitive::*kind* **kind**(int *index*) **const**

Returns the primitive kind of post-op at entry with a certain index.

Return Primitive kind of the post-op at the specified index.

Parameters

- `index`: Index of the post-op to return the kind for.

void **append_sum**(float *scale* = 1.f, *memory*::*data_type* *data_type* = *memory*::*data_type*::*undef*)

Appends an accumulation (sum) post-op. Prior to accumulating the result, the previous value would be multiplied by a scaling factor `scale`.

The kind of this post-op is *dnnl::primitive::kind::sum*.

This feature may improve performance for cases like residual learning blocks, where the result of convolution is accumulated to the previously computed activations. The parameter `scale` may be used for the integer-based computations when the result and previous activations have different logical scaling factors.

In the simplest case when the accumulation is the only post-op, the computations would be `dst[:] := scale * dst[:] + op(...)` instead of `dst[:] := op(...)`.

If `data_type` is specified, the original dst tensor will be reinterpreted as a tensor with the provided data type. Because it is a reinterpretation, data_type and dst data type should have the same size. As a result, computations would be `dst[:] <- scale * as_data_type(dst[:]) + op(...)` instead of `dst[:] <- op(...)`.

Note This post-op executes in-place and does not change the destination layout.

Parameters

- `scale`: Scaling factor.
- `data_type`: Data type.

void **get_params_sum**(int *index*, float &*scale*) **const**

Returns the parameters of an accumulation (sum) post-op.

Parameters

- `index`: Index of the sum post-op.
- `scale`: Scaling factor of the sum post-op.

void **get_params_sum**(int *index*, float &*scale*, *memory*::*data_type* &*data_type*) **const**
Returns the parameters of an accumulation (sum) post-op.

Parameters

- `index`: Index of the sum post-op.
- `scale`: Scaling factor of the sum post-op.
- `data_type`: Data type of the sum post-op.

void **append_eltwise**(float *scale*, *algorithm aalgorithm*, float *alpha*, float *beta*)
Appends an elementwise post-op.

The kind of this post-op is *dnnl::primitive::kind::eltwise*.

In the simplest case when the elementwise is the only post-op, the computations would be `dst[:] := scale * eltwise_op (op(...))` instead of `dst[:] <- op(...)`, where eltwise_op is configured with the given parameters.

Parameters

- `scale`: Scaling factor.
- `aalgorithm`: Elementwise algorithm.
- `alpha`: Alpha parameter for the elementwise algorithm.
- `beta`: Beta parameter for the elementwise algorithm.

void **get_params_eltwise**(int *index*, float &*scale*, *algorithm* &*aalgorithm*, float &*alpha*, float &*beta*) **const**
Returns parameters of an elementwise post-up.

Parameters

- `index`: Index of the post-op.
- `scale`: Output scaling factor.
- `aalgorithm`: Output elementwise algorithm kind.
- `alpha`: Output alpha parameter for the elementwise algorithm.
- `beta`: Output beta parameter for the elementwise algorithm.

Scratchpad Mode

Some primitives might require a temporary buffer while performing their computations. For instance, the operations that do not have enough independent work to utilize all cores on a system might use parallelization over the reduction dimension (the K dimension in the GEMM notation). In this case different threads compute partial results in private temporary buffers, and then the private results are added to produce the final result. Another example is using matrix multiplication (GEMM) to implement convolution. Before calling GEMM, the source activations need to be transformed using the `im2col` operation. The transformation result is written to a temporary buffer that is then used as an input for the GEMM.

In both of these examples, the temporary buffer is no longer required once the primitive computation is completed. oneDNN refers to such kind of a memory buffer as a *scratchpad*.

Both types of implementation might need extra space for the reduction in case there are too few independent tasks. The amount of memory required by the `im2col` transformation is proportional to the size of the source image multiplied

by the weights spatial size. The size of a buffer for reduction is proportional to the tensor size to be reduced (e.g., `diff_weights` in the case of backward by weights) multiplied by the number of threads in the reduction groups (the upper bound is the total number of threads).

By contrast, some other primitives might require very little extra space. For instance, one of the implementation of the *dnnl::sum* primitive requires temporary space only to store the pointers to data for each and every input array (that is, the size of the scratchpad is `n * sizeof(void *)`, where `n` is the number of summands).

oneDNN supports two modes for handling scratchpads:

enum dnnl::**scratchpad_mode**

Scratchpad mode.

Values:

enumerator library

The library manages the scratchpad allocation. There may be multiple implementation-specific policies that can be configured via mechanisms that fall outside of the scope of this specification.

enumerator user

The user manages the scratchpad allocation by querying and providing the scratchpad memory to primitives. This mode is thread-safe as long as the scratchpad buffers are not used concurrently by two primitive executions.

The scratchpad mode is controlled though the *dnnl::primitive_attr::set_scratchpad_mode()* primitive attributes.

If the user provides scratchpad memory to a primitive, this memory must be created using the same engine that the primitive uses.

All primitives support both scratchpad modes.

Note: Primitives are not thread-safe by default. The only way to make the primitive execution fully thread-safe is to use the *dnnl::scratchpad_mode::user* mode and not pass the same scratchpad memory to two primitives that are executed concurrently.

Examples

Library Manages Scratchpad

As mentioned above, this is a default behavior. We only want to highlight how a user can query the amount of memory consumed by a primitive due to a scratchpad.

```
// Use default attr, hence the library allocates scratchpad
dnnl::primitive::primitive_desc op_pd(params, /* other arguments */);

// Print how much memory would be hold by a primitive due to scratchpad
std::cout << "primitive will use "
          << op_pd.query_s64(dnnl::query::memory_consumption_s64)
          << " bytes" << std::endl;

// In this case scratchpad is internal, hence user visible scratchpad memory
// descriptor should be empty:
auto zero_md = dnnl::memory::desc();
```

User Manages Scratchpad

```
// Create an empty (default) attributes
dnnl::primitive_attr attr;

// Default scratchpad mode is `library`:
assert(attr.get_scratchpad_mode() == dnnl::scratchpad_mode::library);

// Set scratchpad mode to `user`
attr.set_scratchpad_mode(dnnl::scratchpad_mode::user);

// Create a primitive descriptor with custom attributes
dnnl::primitive::primitive_desc op_pd(op_d, attr, engine);

// Query the scratchpad memory descriptor
dnnl::memory::desc scratchpad_md = op_pd.scratchpad_desc();

// Note, that a primitive doesn't consume memory in this configuration:
assert(op_pd.query_s64(dnnl::query::memory_consumption_s64) == 0);

// Create a primitive
dnnl::primitive prim(op_pd);

// ... more code ..

// Create a scratchpad memory
// NOTE: if scratchpad is not required for a particular primitive the
//       scratchpad_md.get_size() will return 0. It is fine to have
//       scratchpad_ptr == nullptr in this case.
void *scratchpad_ptr = user_memory_manager::allocate(scratchpad_md.get_size());
// NOTE: engine here must much the engine of the primitive
dnnl::memory scratchpad(scratchpad_md, engine, scratchpad_ptr);

// Pass a scratchpad memory to a primitive
prim.execute(stream, { /* other arguments */,
        {DNNL_ARG_SCRATCHPAD, scratchpad}});
```

Quantization

Primitives may support reduced precision computations which require quantization.

Quantization Model

The primary quantization model that the library assumes is the following:

$$x_{f32}[:] = scale_{f32} \cdot (x_{int8}[:] - 0_{x_{int8}})$$

where $scale_{f32}$ is a *scaling factor* that is somehow known in advance and $[:]$ is used to denote elementwise application of the formula to the arrays. Typically, the process of computing scale factors is called *calibration*. The library cannot compute any of the scale factors at run-time dynamically. Hence, the model is sometimes called a *static* quantization model. The main rationale to support only *static* quantization out-of-the-box is higher performance. To use *dynamic* quantization:

1. Compute the result in higher precision, like *dnnl::memory::data_type::s32*.

2. Find the required characteristics, like min and max values, and derive the scale factor.
3. Re-quantize to the lower precision data type.

oneDNN assumes a fixed zero position. For most of the primitives, the real zero value is mapped to the zero for quantized values; that is, $0_{x_{int8}} = 0$. For example, this is the only model that *Convolution and Deconvolution* and *Inner Product* currently support. The *RNN* primitives have limited support of shifted zero.

For the rest of this section we that $0_{x_{int8}} = 0$.

Example: Convolution Quantization Workflow

Consider a convolution without bias. The tensors are represented as:

- $\text{src}_{f32}[:] = scale_{\text{src}} \cdot \text{src}_{int8}[:]$
- $\text{weights}_{f32}[:] = scale_{\text{weights}} \cdot \text{weights}_{int8}[:]$
- $\text{dst}_{f32}[:] = scale_{\text{dst}} \cdot \text{dst}_{int8}[:]$

Here the $\text{src}_{f32}, \text{weights}_{f32}, \text{dst}_{f32}$ are not computed at all, the whole work happens with int8 tensors. As mentioned above, we also somehow know all the scaling factors: *scale_{src}, scale_{weights}, scale_{dst}*.

So the task is to compute the dst_{int8} tensor.

Mathematically, the computations are:

$$\text{dst}_{int8}[:] = \text{f32\_to\_int8}(output\_scale \cdot conv_{s32}(\text{src}_{int8}, \text{weights}_{int8})),$$

where

- $output\_scale := \frac{scale_{\text{src}} \cdot scale_{\text{weights}}}{scale_{\text{dst}}}$;
- $conv_{s32}$ is just a regular convolution which takes source and weights with int8 data type and compute the result in int32 data type (int32 is chosen to avoid overflows during the computations);
- $\text{f32\_to\_s8}()$ converts an *f32* value to *s8* with potential saturation if the values are out of the range of the int8 data type.

Note that in order to perform the operation, one doesn't need to know the exact scaling factors for all the tensors; it is enough to know only the *output_scale*. The library utilizes this fact: a user needs to provide only this one extra parameter to the convolution primitive (see the *Output Scaling Attribute* section below).

Per-Channel Scaling

Primitives may have limited support of multiple scales for a quantized tensor. The most popular use case is the *Convolution and Deconvolution* primitives that support per-output-channel scaling factors for the weights, meaning that the actual convolution computations would need to scale different output channels differently.

Let α denote scales:

- $\text{src}_{f32}(n, ic, ih, iw) = \alpha_{\text{src}} \cdot \text{src}_{int8}(n, ic, ih, iw)$
- $\text{weights}_{f32}(oc, ic, kh, kw) = \alpha_{\text{weights}}(oc) \cdot \text{weights}_{int8}(oc, ic, kh, kw)$
- $\text{dst}_{f32}(n, oc, oh, ow) = scale_{\text{dst}} \cdot \text{dst}_{int8}(n, oc, oh, ow)$

Note that now the weights' scaling factor depends on the oc.

To compute the dst_{int8} we need to perform the following:

$$\text{dst}_{int8}(n, oc, oh, ow) = \text{f32\_to\_int8}(output\_scale(oc) \cdot conv_{s32}(\text{src}_{int8}, \text{weights}_{int8})|_{(n,oc,oh,ow)}),$$

where

$$output\_scale(oc) := \frac{\alpha_{\text{src}} \cdot \alpha_{\text{weights}}(oc)}{\alpha_{\text{dst}}}.$$

The user is responsible for preparing quantized weights accordingly. To do that, oneDNN provides reorders that can perform per-channel scaling:

$$\text{weights}_{int8}(oc, ic, kh, kw) = \text{f32\_to\_int8}(output\_scale(oc) \cdot \text{weights}_{f32}(oc, ic, kh, kw)),$$

where

$$output\_scale(oc) := \frac{1}{\alpha_{\text{weights}}(oc)}.$$

Output Scaling Attribute

oneDNN provides *dnnl::primitive_attr::set_output_scales()* for setting scaling factors for most of the primitives.

The primitives may not support output scales if source (and weights) tensors are not of the int8 data type. In other words, convolution operating on the single precision floating point data type may not scale the output result.

In the simplest case, when there is only one common scale the attribute changes the op behavior from

$$\text{dst}[:] = Op(...)$$

to

$$\text{dst}[:] = scale \cdot Op(...).$$

To support scales per one or several dimensions, users must set the appropriate mask.

Say the primitive destination is a $D_0 \times ... \times D_{n-1}$ tensor and we want to have output scales per d_i dimension (where $0 \leq d_i < n$).

Then $mask = \sum_{d_i} 2^{d_i}$ and the number of scales should be `scales.size()` $= \prod_{d_i} D_{d_i}$.

The scaling happens in the single precision floating point data type (*dnnl::memory::data_type::f32*). Before it is stored, the result is converted to the destination data type with saturation if required. The rounding happens according to the current hardware setting.

Example 1: weights quantization with per-output-channel-and-group scaling

```
// weights dimensions
const int G, OC, IC, KH, KW;

// original f32 weights in plain format
dnnl::memory::desc wei_plain_f32_md(
        {G, OC/G, IC/G, KH, KW},          // dims
        dnnl::memory::data_type::f32,     // the data originally in f32
        dnnl::memory::format_tag::hwigo   // the plain memory format
        );

// the scaling factors for quantized weights
// An unique scale for each group and output-channel.
```

(continues on next page)

(continued from previous page)

```
std::vector<float> wei_scales(G * OC/G) = { /* values */ };

// int8 convolution primitive descriptor
dnnl::convolution_forward::primitive_desc conv_pd(/* see the next example */);

// query the convolution weights memory descriptor
dnnl::memory::desc wei_conv_s8_md = conv_pd.weights_desc();

// prepare the inverse of the scales
// (f32 = scale * int8 --> int8 = 1/scale * f32)
std::vector<float> inv_wei_scales(wei_scales.size());
for (size_t i = 0; i < wei_scales.size(); ++i)
    inv_wei_scales[i] = 1.f / wei_scales[i];

// prepare the attributes for the reorder
dnnl::primitive_attr attr;
const int mask = 0
    | (1 << 0)  // scale per  G dimension, which is the dim #0
    | (1 << 1); // scale per OC dimension, which is the dim #1
attr.set_output_scales(mask, inv_wei_scales);

// create reorder that would perform:
//   wei_s8(g, oc, ic, kh, kw) <- 1/scale(g, oc) * wei_f32(g, oc, ic, kh, kw)
// including the data format transformation.
auto wei_reorder_pd = dnnl::reorder::primitive_desc(
        wei_plain_f32_md, engine, // source
        wei_conv_s8_md, engine, // destination,
        attr);
auto wei_reorder = dnnl::reorder(wei_reorder_pd);
```

Example 2: convolution with groups, with per-output-channel quantization

This example is complementary to the previous example (which should ideally be the first one). Let's say we want to create an int8 convolution with per-output channel scaling.

```
const float src_scale; // src_f32[:] = src_scale * src_s8[:]
const float dst_scale; // dst_f32[:] = dst_scale * dst_s8[:]

// the scaling factors for quantized weights (as declared above)
// An unique scale for each group and output-channel.
std::vector<float> wei_scales(G * OC/G) = {...};


// Src, weights, and dst memory descriptors for convolution,
// with memory format tag == any to allow a convolution implementation
// to chose the appropriate memory format

dnnl::memory::desc src_conv_s8_any_md(
        {BATCH, IC, IH, IW},            // dims
        dnnl::memory::data_type::s8,  // the data originally in s8
        dnnl::memory::format_tag::any // let convolution to choose
        );

dnnl::memory::desc wei_conv_s8_any_md(
        {G, OC/G, IC/G, KH, KW},        // dims
```

(continues on next page)

(continued from previous page)

```
        dnnl::memory::data_type::s8,  // the data originally in s8
        dnnl::memory::format_tag::any // let convolution to choose
        );

dnnl::memory::desc dst_conv_s8_any_md(...);  // ditto

// Create a convolution operation descriptor
dnnl::convolution_forward::desc conv_d(
        dnnl::prop_kind::forward_inference,
        dnnl::algorithm::convolution_direct,
        src_conv_s8_any_md,                     // what's important is that
        wei_conv_s8_any_md,                     // we specified that we want
        dst_conv_s8_any_md,                     // computations in s8
        strides, padding_l, padding_r,
        dnnl::padding_kind::zero
        );

// prepare the attributes for the convolution
dnnl::primitive_attr attr;
const int mask = 0
    | (1 << 1); // scale per OC dimension, which is the dim #1 on dst tensor:
                // (BATCH, OC, OH, OW)
                //    0     1   2   3
std::vector<float> conv_output_scales(G * OC/G);
for (int g_oc = 0; G * OC/G; ++g_oc)
    conv_output_scales[g_oc] = src_scale * wei_scales(g_oc) / dst_scale;
attr.set_output_scales(mask, conv_output_scales);

// create a convolution primitive descriptor with the scaling factors
auto conv_pd = dnnl::convolution_forward::primitive_desc(
        conv_d, // general (non-customized) operation descriptor
        attr,   // the attributes contain the output scaling
        engine);
```

Interplay of Output Scales with Post-ops

In general, the *Post-ops* are independent from the output scales. The output scales are applied to the result first; then post-ops will take effect.

That has an implication on the scaling factors passed to the library, however. Consider the following example of a convolution with $\tanh$ post-op:

$$\mathrm{dst}_{s8}[:] = \frac{1}{scale_{\mathrm{dst}}} \cdot \tanh(scale_{\mathrm{src}} \cdot scale_{\mathrm{weights}} \cdot conv_{s32}(\mathrm{src}_{s8}, wei_{s8}))$$

- The convolution output scales are $conv\_output\_scale = scale_{\mathrm{src}} \cdot scale_{\mathrm{weights}}$, i.e. there is no division by $scale_{\mathrm{dst}}$.
- And the post-ops scale for $\tanh$ is set to $scale\_tanh\_post\_op = \frac{1}{scale_{\mathrm{dst}}}$.

Attribute Related Error Handling

Since the attributes are created separately from the corresponding primitive descriptor, consistency checks are delayed. Users can successfully set attributes in whatever configuration they want. However, when they try to create a primitive descriptor with the attributes they set, it might happen that there is no primitive implementation that supports such a configuration. In this case the library will throw the *dnnl::error* exception.

API

struct dnnl::**primitive_attr**

Primitive attributes.

Public Functions

primitive_attr()

Constructs default (empty) primitive attributes.

scratchpad_mode **get_scratchpad_mode**() **const**

Returns the scratchpad mode.

void **set_scratchpad_mode**(*scratchpad_mode mode*)

Sets scratchpad mode.

Parameters

- mode: Specified scratchpad mode.

void **get_output_scales**(int &*mask*, std::vector<float> &*scales*) **const**

Returns output scaling factors correspondence mask and values.

Parameters

- mask: Scaling factors correspondence mask that defines the correspondence between the output tensor dimensions and the scales vector. The set i-th bit indicates that a dedicated output scaling factor is used for each index along that dimension. The mask value of 0 implies a common output scaling factor for the whole output tensor.
- scales: Vector of output scaling factors.

void **set_output_scales**(int *mask*, **const** std::vector<float> &*scales*)

Sets output scaling factors correspondence mask and values.

Example usage:

```
int mb = 32, oc = 32,
    oh = 14, ow = 14; // convolution output params
// unique output scales per output channel
vector<float> scales = { ... };
int oc_dim = 1; // mb_dim = 0, channel_dim = 1, height_dim = 2, ...

// construct a convolution descriptor
dnnl::convolution::desc conv_d;

dnnl::primitive_attr attr;
attr.set_output_scales(attr, oc, 1 << oc_dim, scales);
```

(continues on next page)

(continued from previous page)

```
dnnl::primitive_desc conv_pd(conv_d, attr, engine);
```

Note The order of dimensions does not depend on how elements are laid out in memory. For example:

- for a 2D CNN activations tensor the order is always (n, c)
- for a 4D CNN activations tensor the order is always (n, c, h, w)
- for a 5D CNN weights tensor the order is always (g, oc, ic, kh, kw)

Parameters

- `mask`: Defines the correspondence between the output tensor dimensions and the `scales` vector. The set i-th bit indicates that a dedicated scaling factor is used for each index along that dimension. Set the mask to 0 to use a common output scaling factor for the whole output tensor.
- `scales`: Constant vector of output scaling factors. If the scaling factors are known at the time of this call, the following equality must hold: $scales.size() = \prod_{d \in mask} output.dims[d]$. Violations can only be detected when the attributes are used to create a primitive descriptor. If the scaling factors are not known at the time of the call, this vector must contain a single *DNNL_RUNTIME_F32_VAL* value and the output scaling factors must be passed at execution time as an argument with index *DNNL_ARG_ATTR_OUTPUT_SCALES*.

void **get_scales**(int *arg*, int &*mask*, std::vector<float> &*scales*) **const**

Returns scaling factors correspondence mask and values for a given memory argument.

Parameters

- `arg`: Parameter argument index as passed to the *primitive::execute()* call.
- `mask`: Scaling factors correspondence mask that defines the correspondence between the output tensor dimensions and the `scales` vector. The set i-th bit indicates that a dedicated scaling factor is used for each index along that dimension. Set the mask to 0 to use a common scaling factor for the whole output tensor.
- `scales`: Output vector of scaling factors.

void **set_scales**(int *arg*, int *mask*, **const** std::vector<float> &*scales*)

Sets scaling factors for primitive operations for a given memory argument.

See *dnnl::primitive_attr::set_output_scales*

Parameters

- `arg`: Parameter argument index as passed to the *primitive::execute()* call.
- `mask`: Scaling factors correspondence mask that defines the correspondence between the tensor dimensions and the `scales` vector. The set i-th bit indicates that a dedicated scaling factor is used for each index along that dimension. Set the mask to 0 to use a common scaling factor for the whole output tensor.
- `scales`: Constant vector of scaling factors. The following equality must hold: $scales.size() = \prod_{d \in mask} argument.dims[d]$.

void **get_zero_points**(int *arg*, int &*mask*, std::vector<int32_t> &*zero_points*) **const**

Returns zero points correspondence mask and values.

Parameters

- `arg`: Parameter argument index as passed to the *primitive::execute()* call.
- `mask`: Zero points correspondence mask that defines the correspondence between the output tensor dimensions and the `zero_points` vector. The set i-th bit indicates that a dedicated zero point is used for each index along that dimension. Set the mask to 0 to use a common zero point for the whole output tensor.
- `zero_points`: Output vector of zero points.

void **set_zero_points**(int *arg*, int *mask*, **const** std::vector<int32_t> &*zero_points*)
Sets zero points for primitive operations for a given memory argument.

See *dnnl::primitive_attr::set_output_scales*

Parameters

- `arg`: Parameter argument index as passed to the *primitive::execute()* call.
- `mask`: Zero point correspondence mask that defines the correspondence between the tensor dimensions and the `zero_points` vector. The set i-th bit indicates that a dedicated zero point is used for each index along that dimension. Set the mask to 0 to use a common zero point for the whole output tensor.
- `zero_points`: Constant vector of zero points. If the zero points are known at the time of this call, the following equality must hold: $zero\_points.size() = \prod_{d \in mask} argument.dims[d]$. If the zero points are not known at the time of the call, this vector must contain a single *DNNL_RUNTIME_F32_VAL* value and the zero points must be passed at execution time as an argument with index *DNNL_ARG_ATTR_ZERO_POINTS*.

const *post_ops* **get_post_ops**() **const**
Returns post-ops previously set via *set_post_ops()*.

Return Post-ops.

void **set_post_ops**(**const** *post_ops ops*)
Sets post-ops.

Note There is no way to check whether the post-ops would be supported by the target primitive. Any error will be reported by the respective primitive descriptor constructor.

Parameters

- `ops`: Post-ops object to copy post-ops from.

void **set_rnn_data_qparams**(float *scale*, float *shift*)
Sets quantization scale and shift parameters for RNN data tensors.

For performance reasons, the low-precision configuration of the RNN primitives expect input activations to have the unsigned 8-bit integer data type. The scale and shift parameters are used to quantize floating-point data to unsigned integer and must be passed to the RNN primitive using attributes.

The quantization formula is `scale * (data + shift)`.

Example usage:

```
// RNN parameters
int l = 2, t = 2, mb = 32, sic = 32, slc = 32, dic = 32, dlc = 32;
// Activations quantization parameters
float scale = 2.0f, shift = 0.5f;

primitive_attr attr;

// Set scale and shift for int8 quantization of activation
attr.set_rnn_data_qparams(scale, shift);

// Create and configure rnn op_desc
vanilla_rnn_forward::desc rnn_d(/* arguments */);
vanilla_rnn_forward::primitive_desc rnn_d(rnn_d, attr, engine);
```

Note Quantization scale and shift are common for src_layer, src_iter, dst_iter, and dst_layer.

Parameters

- `scale`: The value to scale the data by.
- `shift`: The value to shift the data by.

void **`set_rnn_weights_qparams`**(int *mask*, **`const`** std::vector<float> &*scales*)

Sets quantization scaling factors for RNN weights tensors. The low-precision configuration of the RNN primitives expect input weights to use the signed 8-bit integer data type. The scaling factors are used to quantize floating-point data to signed integer and must be passed to RNN primitives using attributes.

Note The dimension order is always native and does not depend on the actual layout used. For example, five-dimensional weights always have (l, d, i, g, o) logical dimension ordering.

Note Quantization scales are common for weights_layer and weights_iteration

Parameters

- `mask`: Scaling factors correspondence mask that defines the correspondence between the output tensor dimensions and the `scales` vector. The set i-th bit indicates that a dedicated scaling factor should be used each index along that dimension. Set the mask to 0 to use a common scaling factor for the whole output tensor.
- `scales`: Constant vector of output scaling factors. The following equality must hold: $scales.size() = \prod_{d \in mask} weights.dims[d]$. Violations can only be detected when the attributes are used to create a primitive descriptor.

5.5.3 Batch Normalization

The batch normalization primitive performs a forward or backward batch normalization operation on tensors with number of dimensions equal to 2 or more. Variable names follow the standard *Conventions*.

The batch normalization operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions.

The different flavors of the primitive are controlled by the `flags` parameter that is passed to the operation descriptor initialization function like *`dnnl::batch_normalization_forward::desc::desc()`*. Multiple flags can be combined using the bitwise OR operator (`|`).

Forward

$$\mathrm{dst}(n,c,h,w) = \gamma(c) \cdot \frac{\mathrm{src}(n,c,h,w) - \mu(c)}{\sqrt{\sigma^2(c) + \varepsilon}} + \beta(c),$$

where

- $\gamma(c)$ and $\beta(c)$ are optional scale and shift for a channel (controlled using the `use_scaleshift` flag),
- $\mu(c)$ and $\sigma^2(c)$ are mean and variance for a channel (controlled using the *`use_global_stats`* flag), and
- ε is a constant to improve numerical stability.

Mean and variance are computed at runtime or provided by a user. When mean and variance are computed at runtime, the following formulas are used:

- $\mu(c) = \frac{1}{NHW} \sum_{nhw} \mathrm{src}(n,c,h,w)$,
- $\sigma^2(c) = \frac{1}{NHW} \sum_{nhw} (\mathrm{src}(n,c,h,w) - \mu(c))^2$.

The $\gamma(c)$ and $\beta(c)$ tensors are considered learnable.

In the training mode, the primitive also optionally supports fusion with ReLU activation with zero negative slope applied to the result (see *`fuse_norm_relu`* flag).

Note: The batch normalization primitive computes population mean and variance and not the sample or unbiased versions that are typically used to compute running mean and variance. * Using the mean and variance computed by the batch normalization primitive, running mean and variance $\hat{\mu}_i$ and $\hat{\sigma}_i^2$ where i is iteration number, can be computed as:

$$\hat{\mu}_{i+1} = \alpha \cdot \hat{\mu}_i + (1-\alpha) \cdot \mu,$$
$$\hat{\sigma}^2_{i+1} = \alpha \cdot \hat{\sigma}^2_i + (1-\alpha) \cdot \sigma^2.$$

Difference Between Forward Training and Forward Inference

- If mean and variance are computed at runtime (i.e., *`use_global_stats`* is not set), they become outputs for the propagation kind *`forward_training`* (because they would be required during the backward propagation) and are not exposed for the propagation kind *`forward_inference`*.
- If batch normalization is created with ReLU fusion (i.e., *`fuse_norm_relu`* is set), for the propagation kind *`forward_training`* the primitive would produce a `workspace` memory as one extra output. This memory is required to compute the backward propagation. When the primitive is executed with propagation kind *`forward_inference`*, the workspace is not produced. Behavior would be the same as creating a batch normalization primitive with ReLU as a post-op (see section below).

Backward

The backward propagation computes diff_src(n, c, h, w), diff_$\gamma(c)^*$, and diff_$\beta(c)^*$ based on diff_dst(n, c, h, w), src(n, c, h, w), $\mu(c)$, $\sigma^2(c)$, $\gamma(c)^*$, and $\beta(c)^*$.

The tensors marked with an asterisk are used only when the primitive is configured to use $\gamma(c)$ and $\beta(c)$ (i.e., `use_scaleshift` is set).

Execution Arguments

Depending on the flags and propagation kind, the batch normalization primitive requires different inputs and outputs. For clarity, a summary is shown below.

	forward_inference	*forward_training*	*backward*	*backward_data*
none	*In*: src; *Out*: dst	*In*: src; *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2; *Out*: diff_src	Same as for *backward*
use_global_stats	*In*: src, μ, σ^2; *Out*: dst	*In*: src, μ, σ^2; *Out*: dst	*In*: diff_dst, src, μ, σ^2; *Out*: diff_src	Same as for *backward*
`use_scaleshift`	*In*: src, γ, β; *Out*: dst	*In*: src, γ, β; *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2, γ, β; *Out*: diff_src, diff_γ, diff_β	Not supported
use_global_stats | `use_scaleshift`	*In*: src, μ, σ^2, γ, β; *Out*: dst	*In*: src, μ, σ^2, γ, β; *Out*: dst	*In*: diff_dst, src, μ, σ^2, γ, β; *Out*: diff_src, diff_γ, diff_β	Not supported
`flags` | *fuse_norm_relu*	*In*: same as with `flags`; *Out*: same as with `flags`	*In*: same as with `flags`; *Out*: same as with `flags`, workspace	*In*: same as with `flags`, workspace; *Out*: same as with `flags`	Same as for *backward* if `flags` do not contain `use_scaleshift`; not supported otherwise

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
γ, β	*DNNL_ARG_SCALE_SHIFT*
mean (μ)	*DNNL_ARG_MEAN*
variance (σ)	*DNNL_ARG_VARIANCE*
dst	*DNNL_ARG_DST*
workspace	*DNNL_ARG_WORKSPACE*
diff_dst	*DNNL_ARG_DIFF_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_γ, diff_β	*DNNL_ARG_DIFF_SCALE_SHIFT*

Operation Details

1. For forward propagation, the mean and variance might be either computed at runtime (in which case they are outputs of the primitive) or provided by a user (in which case they are inputs). In the latter case, a user must set the *use_global_stats* flag. For the backward propagation, the mean and variance are always input parameters.
2. The memory format and data type for src and dst are assumed to be the same, and in the API they are typically referred to as data (e.g., see data_desc in *dnnl::batch_normalization_forward::desc::desc()*). The same is true for diff_src and diff_dst. The corresponding memory descriptors are referred to as diff_data_desc.
3. Both forward and backward propagation support in-place operations, meaning that src can be used as input and output for forward propagation, and diff_dst can be used as input and output for backward propagation. In case of an in-place operation, the original data will be overwritten. Note, however, that backward propagation requires original src, hence the corresponding forward propagation should not be performed in-place.
4. As mentioned above, the batch normalization primitive can be fused with ReLU activation even in the training mode. In this case, on the forward propagation the primitive has one additional output, workspace, that should be passed during the backward propagation.

Data Types Support

The operation supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination	Mean / Variance / ScaleShift
forward / backward	*f32*, *bf16*	*f32*
forward	*f16*	*f32*
forward	*s8*	*f32*

Data Representation

Source, Destination, and Their Gradients

Like other CNN primitives, the batch normalization primitive expects data to be $N \times C \times SP_n \times \cdots \times SP_0$ tensor.

The batch normalization primitive is optimized for the following memory formats:

Spatial	Logical tensor	Implementations optimized for memory formats
0D	NC	*nc* (*ab*)
1D	NCW	*ncw* (*abc*), *nwc* (*acb*), *optimized*
2D	NCHW	*nchw* (*abcd*), *nhwc* (*acdb*), *optimized*
3D	NCDHW	*ncdhw* (*abcde*), *ndhwc* (*acdeb*), *optimized*

Here *optimized* means the format chosen by the preceding compute-intensive primitive.

Statistics Tensors

The mean (μ) and variance (σ^2) are separate 1D tensors of size C.

The format of the corresponding memory object must be `x` (`a`).

If used, the scale (γ) and shift (β) are combined in a single 2D tensor of shape $2 \times C$.

The format of the corresponding memory object must be `nc` (`ab`).

Post-ops and Attributes

Propagation	Type	Operation	Description
forward	post-op	eltwise	Applies an eltwise operation to the output.

Note: Using ReLU as a post-op does not produce additional output in the `workspace` that is required to compute backward propagation correctly. Hence, one should use the `fuse_norm_relu` flag for training.

API

struct `dnnl::batch_normalization_forward` : **public** dnnl::*primitive*

Batch normalization forward propagation primitive.

Public Functions

batch_normalization_forward()

Default constructor. Produces an empty object.

batch_normalization_forward(**const** *primitive_desc* &*pd*)

Constructs a batch normalization forward propagation primitive.

Parameters

- `pd`: Primitive descriptor for a batch normalization forward propagation primitive.

struct desc

Descriptor for a batch normalization forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, **const** *memory*::*desc* &*data_desc*, float *epsilon*, *normalization_flags flags*)

Constructs a batch normalization descriptor for forward propagation.

Note In-place operation is supported: the dst can refer to the same memory as the src.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training* and *dnnl::prop_kind::forward_inference*.
- `data_desc`: Source and destination memory descriptors.
- `epsilon`: Batch normalization epsilon parameter.
- `flags`: Batch normalization flags (*dnnl::normalization_flags*).

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a batch normalization forward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a batch normalization forward propagation primitive.

Parameters

- `adesc`: Descriptor for a batch normalization forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a batch normalization forward propagation primitive.

Parameters

- `adesc`: Descriptor for a batch normalization forward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **weights_desc**() **const**

Returns a weights memory descriptor.

Return Weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **workspace_desc**() **const**

Returns the workspace memory descriptor.

Return Workspace memory descriptor.

Return A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **mean_desc**() **const**

Returns memory descriptor for mean.

Return Memory descriptor for mean.

memory::*desc* **variance_desc**() **const**

Returns memory descriptor for variance.

Return Memory descriptor for variance.

struct dnnl::**batch_normalization_backward** : **public** dnnl::*primitive*

Batch normalization backward propagation primitive.

Public Functions

batch_normalization_backward()

Default constructor. Produces an empty object.

batch_normalization_backward(**const** *primitive_desc* &*pd*)

Constructs a batch normalization backward propagation primitive.

Parameters

- pd: Primitive descriptor for a batch normalization backward propagation primitive.

struct desc

Descriptor for a batch normalization backward propagation primitive.

Public Functions

desc(*prop_kind* *aprop_kind*, **const** *memory*::*desc* &*diff_data_desc*, **const** *memory*::*desc* &*data_desc*, float *epsilon*, *normalization_flags* *flags*)

Constructs a batch normalization descriptor for backward propagation.

Parameters

- aprop_kind: Propagation kind. Possible values are *dnnl::prop_kind::backward_data* and *dnnl::prop_kind::backward* (diffs for all parameters are computed in this case).
- diff_data_desc: Diff source and diff destination memory descriptor.
- data_desc: Source memory descriptor.
- epsilon: Batch normalization epsilon parameter.
- flags: Batch normalization flags (*dnnl::normalization_flags*).

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a batch normalization backward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *batch_normalization_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a batch normalization backward propagation primitive.

Parameters

- adesc: Descriptor for a batch normalization backward propagation primitive.
- aengine: Engine to use.
- hint_fwd_pd: Primitive descriptor for a batch normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *batch_normalization_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a batch normalization backward propagation primitive.

Parameters

- `adesc`: Descriptor for a batch normalization backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a batch normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() **const**

Returns a weights memory descriptor.

Return Weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() **const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **diff_src_desc**() **const**

Returns a diff source memory descriptor.

Return Diff source memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() **const**

Returns a diff destination memory descriptor.

Return Diff destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_weights_desc**() **const**

Returns a diff weights memory descriptor.

Return Diff weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **mean_desc**() **const**

Returns memory descriptor for mean.

Return Memory descriptor for mean.

memory::*desc* **variance_desc**() **const**

Returns memory descriptor for variance.

Return Memory descriptor for variance.

memory::*desc* **workspace_desc**() **const**

Returns the workspace memory descriptor.

Return Workspace memory descriptor.

Return A zero memory descriptor if the primitive does not require workspace parameter.

5.5.4 Binary

The binary primitive computes a result of a binary elementwise operation between tensors source 0 and source 1.

$$\mathrm{dst}(\overline{x}) = \mathrm{src}_0(\overline{x}) \; op \; \mathrm{src}_1(\overline{x}),$$

where $\overline{x} = (x_0, \ldots, x_n)$ and op is an operator like addition, multiplication, maximum or minimum. Variable names follow the standard *Conventions*.

Forward and Backward

The binary primitive does not have a notion of forward or backward propagations.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src_0	*DNNL_ARG_SRC_0*
src_1	*DNNL_ARG_SRC_1*
dst	*DNNL_ARG_DST*

Operation Details

- The binary primitive requires all source and destination tensors to have the same number of dimensions.
- The binary primitive supports implicit broadcast semantics for source 1. It means that if some dimension has value of one, this value will be used to compute an operation with each point of source 0 for this dimension.
- The dst memory format can be either specified explicitly or by *dnnl::memory::format_tag::any* (recommended), in which case the primitive will derive the most appropriate memory format based on the format of the source 0 tensor.
- Destination memory descriptor should completely match source 0 memory descriptor.
- The binary primitive supports in-place operations, meaning that source 0 tensor may be used as the destination, in which case its data will be overwritten.

Post-ops and Attributes

The following attributes should be supported:

Type	Operation	Description	Restrictions
Attribute	*Scales*	Scales the corresponding input tensor by the given scale factor(s).	The corresponding tensor has integer data type. Only one scale per tensor is supported. Input tensors only.
Post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it.	Must precede eltwise post-op.
Post-op	*Eltwise*	Applies an elementwise operation to the result.	

Data Types Support

The source and destination tensors may have *dnnl::memory::data_type::f32*, *dnnl::memory::data_type::bf16*, *dnnl::memory::data_type::s8* or *dnnl::memory::data_type::u8* data types.

Data Representation

The binary primitive works with arbitrary data tensors. There is no special meaning associated with any of tensors dimensions.

API

struct dnnl::**binary** : **public** dnnl::*primitive*

Elementwise binary operator primitive.

Public Functions

binary()

Default constructor. Produces an empty object.

binary(**const** *primitive_desc* &*pd*)

Constructs an elementwise binary operation primitive.

Parameters

- pd: Primitive descriptor for an elementwise binary operation primitive.

struct desc

Descriptor for an elementwise binary operator primitive.

Public Functions

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*src0*, **const** *memory*::*desc* &*src1*, **const** *memory*::*desc* &*dst*)

Constructs a descriptor for an elementwise binary operator primitive.

Parameters

- aalgorithm: Elementwise algorithm.
- src0: Memory descriptor for source tensor #0.
- src1: Memory descriptor for source tensor #1.
- dst: Memory descriptor for destination tensor.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for an elementwise binary operator primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for an elementwise binary operator primitive.

Parameters
- `adesc`: Descriptor for an elementwise binary operator primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for an elementwise binary operator primitive.

Parameters
- `adesc`: Descriptor for an elementwise binary operator primitive.
- `aengine`: Engine to use.
- `attr`: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**(int *idx* = 0) **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter with index `pdx`.
Parameters
- `idx`: Source index.

memory::*desc* **src0_desc**() **const**
Returns the memory descriptor for source #0.

memory::*desc* **src1_desc**() **const**
Returns the memory descriptor for source #1.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

5.5.5 Concat

A primitive to concatenate data by arbitrary dimension.

The concat primitive concatenates N tensors over `concat_dimension` (here denoted as C), and is defined as

$$\mathrm{dst}(\overline{ou}, c, \overline{in}) = \mathrm{src}_i(\overline{ou}, c', \overline{in}),$$

where

- $c = C_1 + \ldots + C_{i-1} + c'$,
- $\overline{ou}$ is the outermost indices (to the left from concat axis),
- $\overline{in}$ is the innermost indices (to the right from concat axis), and

Variable names follow the standard *Conventions*.

Forward and Backward

The concat primitive does not have a notion of forward or backward propagations. The backward propagation for the concatenation operation is simply an identity operation.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_MULTIPLE_SRC*
dst	*DNNL_ARG_DST*

Operation Details

1. The dst memory format can be either specified by a user or derived by the primitive. The recommended way is to allow the primitive to choose the most appropriate format.
2. The concat primitive requires all source and destination tensors to have the same shape except for the `concat_dimension`. The destination dimension for the `concat_dimension` must be equal to the sum of the `concat_dimension` dimensions of the sources (i.e. $C = \sum_i C_i$). Implicit broadcasting is not supported.

Data Types Support

The concat primitive supports arbitrary data types for source and destination tensors. However, it is required that all source tensors are of the same data type (but not necessarily matching the data type of the destination tensor).

Data Representation

The concat primitive does not assign any special meaning associated with any logical dimensions.

Post-ops and Attributes

The concat primitive does not support any post-ops or attributes.

API

struct `dnnl::concat` : **public** dnnl::*primitive*

Tensor concatenation (concat) primitive.

Public Functions

concat()
Default constructor. Produces an empty object.

concat(**const** *primitive_desc* &*pd*)
Constructs a concatenation primitive.

Parameters

- `pd`: Primitive descriptor for concatenation primitive.

struct primitive_desc : **public** dnnl::*primitive_desc_base*
Primitive descriptor for a concat primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *memory*::*desc* &*dst*, int *concat_dimension*, **const** std::vector<*memory*::*desc*> &*srcs*, **const** *engine* &*aengine*, **const** *primitive_attr* &*attr* = *primitive_attr*())
Constructs a primitive descriptor for an out-of-place concatenation primitive.

Parameters

- `dst`: Destination memory descriptor.
- `concat_dimension`: Source tensors will be concatenated over dimension with this index. Note that order of dimensions does not depend on memory format.
- `srcs`: Vector of source memory descriptors.
- `aengine`: Engine to perform the operation on.
- `attr`: Primitive attributes to use (optional).

primitive_desc(int *concat_dimension*, **const** std::vector<*memory*::*desc*> &*srcs*, **const** *engine* &*aengine*, **const** *primitive_attr* &*attr* = *primitive_attr*())
Constructs a primitive descriptor for an out-of-place concatenation primitive.

This version derives the destination memory descriptor automatically.

Parameters

- `concat_dimension`: Source tensors will be concatenated over dimension with this index. Note that order of dimensions does not depend on memory format.
- `srcs`: Vector of source memory descriptors.
- `aengine`: Engine to perform the operation on.
- `attr`: Primitive attributes to use (optional).

memory::*desc* **src_desc**(int *idx* = 0) **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter with index `pdx`.
Parameters

- `idx`: Source index.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

5.5.6 Convolution and Deconvolution

The convolution and deconvolution primitives compute forward, backward, or weight update for a batched convolution or deconvolution operations on 1D, 2D, or 3D spatial data with bias.

The operations are defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard *Conventions*.

Forward

Let src, $\mathrm{weights}$ and dst be $N \times IC \times IH \times IW$, $OC \times IC \times KH \times KW$, and $N \times OC \times OH \times OW$ tensors respectively. Let bias be a 1D tensor with OC elements.

Furthermore, let the remaining convolution parameters be:

Parameter	Depth	Height	Width	Comment
Padding: Front, top, and left	PD_L	PH_L	PW_L	In the API `padding_l` indicates the corresponding vector of paddings (`_l` in the name stands for **left**)
Padding: Back, bottom, and right	PD_R	PH_R	PW_R	In the API `padding_r` indicates the corresponding vector of paddings (`_r` in the name stands for **right**)
Stride	SD	SH	SW	Convolution without strides is defined by setting the stride parameters to 1
Dilation	DD	DH	DW	Non-dilated convolution is defined by setting the dilation parameters to 0

The following formulas show how oneDNN computes convolutions. They are broken down into several types to simplify the exposition, but in reality the convolution types can be combined.

To further simplify the formulas, we assume that $\mathrm{src}(n, ic, ih, iw) = 0$ if $ih < 0$, or $ih \geq IH$, or $iw < 0$, or $iw \geq IW$.

Regular Convolution

$$\begin{aligned}&\mathrm{dst}(n, oc, oh, ow) = \mathrm{bias}(oc) \\ &+ \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \mathrm{src}(n, ic, oh', ow') \cdot \mathrm{weights}(oc, ic, kh, kw).\end{aligned}$$

Here:

- $oh' = oh \cdot SH + kh - PH_L$,
- $ow' = ow \cdot SW + kw - PW_L$,
- $OH = \left\lfloor \frac{IH - KH + PH_L + PH_R}{SH} \right\rfloor + 1$,
- $OW = \left\lfloor \frac{IW - KW + PW_L + PW_R}{SW} \right\rfloor + 1$.

Convolution with Groups

oneDNN adds a separate groups dimension to memory objects representing weights tensors and represents weights as $G \times OC_G \times IC_G \times KH \times KW$ 5D tensors for 2D convolutions with groups.

$$\begin{aligned}\mathrm{dst}(n, g \cdot OC_G + oc_g, oh, ow) &= \mathrm{bias}(g \cdot OC_G + oc_g) \\ &+ \sum_{ic_g=0}^{IC_G-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \mathrm{src}(n, g \cdot IC_G + ic_g, oh', ow') \cdot \mathrm{weights}(g, oc_g, ic_g, kh, kw),\end{aligned}$$

where

- $IC_G = \frac{IC}{G}$,
- $OC_G = \frac{OC}{G}$, and
- $oc_g \in [0, OC_G)$.

The case when $OC_G = IC_G = 1$ is also known as *a depthwise convolution*.

Convolution with Dilation

$$\begin{aligned}&\mathrm{dst}(n, oc, oh, ow) = \mathrm{bias}(oc) + \\ &+ \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \mathrm{src}(n, ic, oh'', ow'') \cdot \mathrm{weights}(oc, ic, kh, kw).\end{aligned}$$

Here:

- $oh'' = oh \cdot SH + kh \cdot (DH + 1) - PH_L$,
- $ow'' = ow \cdot SW + kw \cdot (DW + 1) - PW_L$,
- $OH = \left\lfloor \frac{IH - DKH + PH_L + PH_R}{SH} \right\rfloor + 1$, where $DKH = 1 + (KH - 1) \cdot (DH + 1)$, and
- $OW = \left\lfloor \frac{IW - DKW + PW_L + PW_R}{SW} \right\rfloor + 1$, where $DKW = 1 + (KW - 1) \cdot (DW + 1)$.

Deconvolution (Transposed Convolution)

Deconvolutions (also called fractionally-strided convolutions or transposed convolutions) can be defined by swapping the forward and backward passes of a convolution. One way to put it is to note that the weights define a convolution, but whether it is a direct convolution or a transposed convolution is determined by how the forward and backward passes are computed.

Difference Between Forward Training and Forward Inference

There is no difference between the `forward_training` and `forward_inference` propagation kinds.

Backward

The backward propagation computes diff_src based on diff_dst and weights.

The weights update computes diff_weights and diff_bias based on diff_dst and src.

Note: The *optimized* memory formats src and weights might be different on forward propagation, backward propagation, and weights update.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
weights	*DNNL_ARG_WEIGHTS*
bias	*DNNL_ARG_BIAS*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_weights	*DNNL_ARG_DIFF_WEIGHTS*
diff_bias	*DNNL_ARG_DIFF_BIAS*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

N/A

Data Types Support

Convolution primitive supports the following combination of data types for source, destination, and weights memory objects.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source	Weights	Destination	Bias
forward / backward	*f32*	*f32*	*f32*	*f32*
forward	*f16*	*f16*	*f16*	*f16*
forward	*u8*, *s8*	*s8*	*u8*, *s8*, *s32*, *f32*	*u8*, *s8*, *s32*, *f32*
forward	*bf16*	*bf16*	*f32*, *bf16*	*f32*, *bf16*
backward	*f32*, *bf16*	*bf16*	*bf16*	
weights update	*bf16*	*f32*, *bf16*	*bf16*	*f32*, *bf16*

Data Representation

Like other CNN primitives, the convolution primitive expects the following tensors:

Spatial	Source / Destination	Weights
1D	$N \times C \times W$	$[G\times]OC \times IC \times KW$
2D	$N \times C \times H \times W$	$[G\times]OC \times IC \times KH \times KW$
3D	$N \times C \times D \times H \times W$	$[G\times]OC \times IC \times KD \times KH \times KW$

Memory format of data and weights memory objects is critical for convolution primitive performance. In the oneDNN programming model, convolution is one of the few primitives that support the placeholder memory format tag `any` and can define data and weight memory objects format based on the primitive parameters. When using `any` it is necessary to first create a convolution primitive descriptor and then query it for the actual data and weight memory objects formats.

While convolution primitives can be created with memory formats specified explicitly, the performance is likely to be suboptimal.

The table below shows the combinations for which *plain* memory formats the convolution primitive is optimized for.

Spatial	Convolution Type	Data / Weights logical tensor	Implementation optimized for memory formats
1D, 2D, 3D		`any`	*optimized*
1D	f32, bf16	NCW / OIW, GOIW	`ncw` (`abc`) / `oiw` (`abc`), `goiw` (`abcd`)
1D	f32, bf16	NCW / OIW, GOIW	`nwc` (`acb`) / `wio` (`cba`), `wigo` (`dcab`)
1D	int8	NCW / OIW	`nwc` (`acb`) / `wio` (`cba`)
2D	f32, bf16	NCHW / OIHW, GOIHW	`nchw` (`abcd`) / `oihw` (`abcd`), `goihw` (`abcde`)
2D	f32, bf16	NCHW / OIHW, GOIHW	`nhwc` (`acdb`) / `hwio` (`cdba`), `hwigo` (`decab`)
2D	int8	NCHW / OIHW, GOIHW	`nhwc` (`acdb`) / `hwio` (`cdba`), `hwigo` (`decab`)
3D	f32, bf16	NCDHW / OIDHW, GOIDHW	`ncdhw` (`abcde`) / `oidhw` (`abcde`), `goidhw` (`abcdef`)
3D	f32, bf16	NCDHW / OIDHW, GOIDHW	`ndhwc` (`acdeb`) / `dhwio` (`cdeba`), `dhwigo` (`defcab`)
3D	int8	NCDHW / OIDHW	`ndhwc` (`acdeb`) / `dhwio` (`cdeba`)

Post-ops and Attributes

Post-ops and attributes enable you to modify the behavior of the convolution primitive by applying the output scale to the result of the primitive and by chaining certain operations after the primitive. The following attributes and post-ops are supported:

Propagation	Type	Operation	Description	Restrictions
forward	attribute	*Output scale*	Scales the result of convolution by given scale factor(s)	int8 convolutions only
forward	post-op	*Eltwise*	Applies an elementwise operation to the result	
forward	post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it	

The primitive supports dynamic quantization via run-time output scales. That means a user could configure attributes with output scales set to the *DNNL_RUNTIME_F32_VAL* wildcard value instead of the actual scales, if the scales are not known at the primitive descriptor creation stage. In this case, the user must provide the scales as an additional input memory object with argument *DNNL_ARG_ATTR_OUTPUT_SCALES* during the execution stage.

Note: The library does not prevent using post-ops in training, but note that not all post-ops are feasible for training usage. For instance, using ReLU with non-zero negative slope parameter as a post-op would not produce an additional output `workspace` that is required to compute backward propagation correctly. Hence, in this particular case one should use separate convolution and eltwise primitives for training.

The following post-ops chaining should be supported by the library:

Type of convolutions	Post-ops sequence supported
f32 and bf16 convolution	eltwise, sum, sum -> eltwise
int8 convolution	eltwise, sum, sum -> eltwise, eltwise -> sum

The attributes and post-ops take effect in the following sequence:

- Output scale attribute,
- Post-ops, in order they were attached.

The operations during attributes and post-ops applying are done in single precision floating point data type. The conversion to the actual destination data type happens just before the actual storing.

Example 1

Consider the following pseudo code:

```
attribute attr;
attr.set_output_scale(alpha);
attr.set_post_ops({
        { sum={scale=beta} },
        { eltwise={scale=gamma, type=tanh, alpha=ignore, beta=ignored }
    });

convolution_forward(src, weights, dst, attr)
```

The would lead to the following:

$$\mathrm{dst}(\overline{x}) = \gamma \cdot \tanh\left(\alpha \cdot conv(\mathrm{src}, \mathrm{weights}) + \beta \cdot \mathrm{dst}(\overline{x})\right)$$

Example 2

The following pseudo code:

```
attribute attr;
attr.set_output_scale(alpha);
attr.set_post_ops({
        { eltwise={scale=gamma, type=relu, alpha=eta, beta=ignored }
        { sum={scale=beta} },
    });

convolution_forward(src, weights, dst, attr)
```

That would lead to the following:

$$\mathrm{dst}(\overline{x}) = \beta \cdot \mathrm{dst}(\overline{x}) + \gamma \cdot ReLU\left(\alpha \cdot conv(\mathrm{src}, \mathrm{weights}), \eta\right)$$

Algorithms

oneDNN implementations may implement convolution primitives using several different algorithms which can be chosen by the user.

- *Direct* (`dnnl::algorithm::convolution_direct`). The convolution operation is computed directly using SIMD instructions. This also includes implicit GEMM formulations which notably may require workspace.
- *Winograd* (`dnnl::algorithm::convolution_winograd`). This algorithm reduces computational complexity of convolution at the expense of accuracy loss and additional memory operations. The implementation is based on the Fast Algorithms for Convolutional Neural Networks by A. Lavin and S. Gray. The Winograd algorithm often results in the best performance, but it is applicable only to particular shapes. Moreover, Winograd only supports int8 and f32 data types.
- *Auto* (`dnnl::algorithm::convolution_auto`). In this case the library should automatically select the *best* algorithm based on the heuristics that take into account tensor shapes and the number of logical processors available.

API

struct `dnnl::convolution_forward` : **public** dnnl::*primitive*

Convolution forward propagation primitive.

Public Functions

convolution_forward()
Default constructor. Produces an empty object.

convolution_forward(**const** *primitive_desc* &*pd*)
Constructs a convolution forward propagation primitive.

Parameters

- `pd`: Primitive descriptor for a convolution forward propagation primitive.

struct desc
Descriptor for a convolution forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)
Constructs a descriptor for a convolution forward propagation primitive with bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `bias_desc`: Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `dst_desc`: Destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)
Constructs a descriptor for a convolution forward propagation primitive without bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `dst_desc`: Destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- `padding_r`: Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a dilated convolution forward propagation primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `bias_desc`: Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `dst_desc`: Destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- `padding_r`: Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a dilated convolution forward propagation primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `dst_desc`: Destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a convolution forward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a convolution forward propagation primitive.

Parameters

- `adesc`: Descriptor for a convolution forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a convolution forward propagation primitive.

Parameters

- `adesc`: Descriptor for a convolution forward propagation primitive.
- `aengine`: Engine to use.
- `attr`: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() **const**

Returns a weights memory descriptor.

Return Weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **bias_desc**() **const**
Returns the bias memory descriptor.
Return The bias memory descriptor.
Return A zero memory descriptor of the primitive does not have a bias parameter.

struct dnnl::**convolution_backward_data** : **public** dnnl::*primitive*
Convolution backward propagation primitive.

Public Functions

convolution_backward_data()
Default constructor. Produces an empty object.

convolution_backward_data(**const** *primitive_desc* &*pd*)
Constructs a convolution backward propagation primitive.

Parameters

- pd: Primitive descriptor for a convolution backward propagation primitive.

struct desc
Descriptor for a convolution backward propagation primitive.

Public Functions

desc(*algorithm* *aalgorithm*, **const** *memory*::*desc* &*diff_src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)
Constructs a descriptor for a convolution backward propagation primitive.

Arrays strides, padding_l, and padding_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.
Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of format_tag.

Parameters

- aalgorithm: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- diff_src_desc: Diff source memory descriptor.
- weights_desc: Weights memory descriptor.
- diff_dst_desc: Diff destination memory descriptor.
- strides: Strides for each spatial dimension.
- padding_l: Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- padding_r: Vector of padding values for high indices for each spatial dimension ([[back,] bottom,] right).

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*diff_src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for dilated convolution backward propagation primitive.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aalgorithm`: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- `diff_src_desc`: Diff source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a convolution backward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *convolution_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a convolution backward propagation primitive.

Parameters

- `adesc`: Descriptor for a convolution backward propagation primitive.
- `aengine`: Engine to perform the operation on.
- `hint_fwd_pd`: Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *convolution_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a convolution backward propagation primitive.

Parameters

- `adesc`: Descriptor for a convolution backward propagation primitive.
- `aengine`: Engine to perform the operation on.
- `attr`: Primitive attributes to use.

- `hint_fwd_pd`: Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() **const**
Returns a diff source memory descriptor.
Return Diff source memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **weights_desc**() **const**
Returns a weights memory descriptor.
Return Weights memory descriptor.
Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **diff_dst_desc**() **const**
Returns a diff destination memory descriptor.
Return Diff destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination parameter.

struct dnnl::**convolution_backward_weights** : **public** dnnl::*primitive*
Convolution weights gradient primitive.

Public Functions

convolution_backward_weights()
Default constructor. Produces an empty object.

convolution_backward_weights(**const** *primitive_desc* &*pd*)
Constructs a convolution weights gradient primitive.

Parameters

- `pd`: Primitive descriptor for a convolution weights gradient primitive.

struct desc
Descriptor for a convolution weights gradient primitive.

Public Functions

desc(*algorithm* *aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)
Constructs a descriptor for a convolution weights gradient primitive with bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.
Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aalgorithm`: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- `src_desc`: Source memory descriptor.

- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_bias_desc`: Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- `padding_r`: Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a convolution weights gradient primitive without bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aalgorithm`: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- `padding_r`: Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a dilated convolution weights gradient primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aalgorithm`: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_bias_desc`: Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corre-

sponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- `padding_r`: Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a dilated convolution weights gradient primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- `aalgorithm`: Convolution algorithm. Possible values are *dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*, and *dnnl::algorithm::convolution_auto*.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- `padding_r`: Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a convolution weights gradient primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *convolution_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a convolution weights gradient primitive.

Parameters
- `adesc`: Descriptor for a convolution weights gradient primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *convolution_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a convolution weights gradient primitive.

Parameters

- `adesc`: Descriptor for a convolution weights gradient primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_weights_desc**() **const**

Returns a diff weights memory descriptor.

Return Diff weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **diff_dst_desc**() **const**

Returns a diff destination memory descriptor.

Return Diff destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_bias_desc**() **const**

Returns the diff bias memory descriptor.

Return The diff bias memory descriptor.

Return A zero memory descriptor of the primitive does not have a diff bias parameter.

struct dnnl::**deconvolution_forward** : **public** dnnl::*primitive*

Deconvolution forward propagation primitive.

Public Functions

deconvolution_forward()

Default constructor. Produces an empty object.

deconvolution_forward(**const** *primitive_desc* &*pd*)

Constructs a deconvolution forward propagation primitive.

Parameters

- `pd`: Primitive descriptor for a deconvolution forward propagation primitive.

struct desc

Descriptor for a deconvolution forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a deconvolution forward propagation primitive with bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: Deconvolution algorithm: *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `bias_desc`: Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `dst_desc`: Destination memory descriptor.
- `strides`: Vector of strides for spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a deconvolution forward propagation primitive without bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: Deconvolution algorithm: *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `dst_desc`: Destination memory descriptor.
- `strides`: Vector of strides for spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a dilated deconvolution forward propagation primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: Deconvolution algorithm: *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `bias_desc`: Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `dst_desc`: Destination memory descriptor.
- `strides`: Vector of strides for spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a dilated deconvolution forward propagation primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: Deconvolution algorithm: *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `dst_desc`: Destination memory descriptor.
- `strides`: Vector of strides for spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a deconvolution forward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a deconvolution forward propagation primitive.

Parameters

- `adesc`: Descriptor for a deconvolution forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a deconvolution forward propagation primitive.

Parameters

- `adesc`: Descriptor for a deconvolution forward propagation primitive.
- `aengine`: Engine to use.
- `attr`: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() **const**

Returns a weights memory descriptor.

Return Weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() **const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **bias_desc**() **const**

Returns the bias memory descriptor.

Return The bias memory descriptor.

Return A zero memory descriptor of the primitive does not have a bias parameter.

struct dnnl::**deconvolution_backward_data** : **public** dnnl::*primitive*

Deconvolution backward propagation primitive.

Public Functions

deconvolution_backward_data()
Default constructor. Produces an empty object.

deconvolution_backward_data(**const** *primitive_desc* &*pd*)
Constructs a deconvolution backward propagation primitive.

Parameters

- `pd`: Primitive descriptor for a deconvolution backward propagation primitive.

struct desc
Descriptor for a deconvolution backward propagation primitive.

Public Functions

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*diff_src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)
Constructs a descriptor for a deconvolution backward propagation primitive.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aalgorithm`: Deconvolution algorithm (*dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*).
- `diff_src_desc`: Diff source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*diff_src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)
Constructs a descriptor for a dilated deconvolution backward propagation primitive.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aalgorithm`: Deconvolution algorithm (*dnnl::algorithm::convolution_direct*, *dnnl::algorithm::convolution_winograd*).
- `diff_src_desc`: Diff source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `diff_dst_desc`: Diff destination memory descriptor.

- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a deconvolution backward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *deconvolution_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a deconvolution backward propagation primitive.

Parameters

- `adesc`: Descriptor for a deconvolution backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *deconvolution_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a deconvolution backward propagation primitive.

Parameters

- `adesc`: Descriptor for a deconvolution backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() **const**

Returns a diff source memory descriptor.

Return Diff source memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **weights_desc**() **const**

Returns a weights memory descriptor.

Return Weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **diff_dst_desc**() **const**

Returns a diff destination memory descriptor.

Return Diff destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination parameter.

struct dnnl::**deconvolution_backward_weights** : **public** dnnl::*primitive*

Deconvolution weights gradient primitive.

Public Functions

deconvolution_backward_weights()

Default constructor. Produces an empty object.

deconvolution_backward_weights(**const** *primitive_desc* &*pd*)

Constructs a deconvolution weights gradient primitive.

Parameters

- pd: Primitive descriptor for a deconvolution weights gradient primitive.

struct desc

Descriptor for a deconvolution weights gradient primitive.

Public Functions

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a deconvolution weights gradient primitive with bias.

Arrays strides, padding_l, and padding_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of format_tag.

Parameters

- aalgorithm: Deconvolution algorithm. Possible values are *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- src_desc: Source memory descriptor.
- diff_weights_desc: Diff weights memory descriptor.
- diff_bias_desc: Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- diff_dst_desc: Diff destination memory descriptor.
- strides: Strides for each spatial dimension.
- padding_l: Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- padding_r: Vector of padding values for high indices for each spatial dimension ([[back,] bottom,] right).

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a deconvolution weights gradient primitive without bias.

Arrays strides, padding_l, and padding_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of format_tag.

Parameters

- `aalgorithm`: Deconvolution algorithm. Possible values are *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a dilated deconvolution weights gradient primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aalgorithm`: Deconvolution algorithm. Possible values are *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_bias_desc`: Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*dilates*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for a dilated deconvolution weights gradient primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aalgorithm`: Deconvolution algorithm. Possible values are *dnnl::algorithm::deconvolution_direct*, and *dnnl::algorithm::deconvolution_winograd*.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.

- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a deconvolution weights gradient primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *deconvolution_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a deconvolution weights update primitive.

Parameters

- `adesc`: Descriptor for a deconvolution weights gradient primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *deconvolution_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a deconvolution weights update primitive.

Parameters

- `adesc`: Descriptor for a deconvolution weights gradient primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_weights_desc**() **const**

Returns a diff weights memory descriptor.

Return Diff weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **diff_dst_desc**() **const**

Returns a diff destination memory descriptor.

Return Diff destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_bias_desc() const**
Returns the diff bias memory descriptor.
Return The diff bias memory descriptor.
Return A zero memory descriptor of the primitive does not have a diff bias parameter.

5.5.7 Elementwise

The elementwise primitive applies an operation to every element of the tensor. Variable names follow the standard *Conventions*.

$$\mathrm{dst}(\overline{x}) = Operation(\mathrm{src}(\overline{x})),$$

for $\overline{x} = (x_0, \ldots, x_n)$.

Forward

The following forward operations are supported. Here s and d denote src and dst, tensor values respectively.

Elementwise algorithm	Forward formula
eltwise_abs	$d = \begin{cases} s & \text{if } s > 0 \\ -s & \text{if } s \leq 0 \end{cases}$
eltwise_bounded_relu	$d = \begin{cases} \alpha & \text{if } s > \alpha \geq 0 \\ s & \text{if } 0 < s \leq \alpha \\ 0 & \text{if } s \leq 0 \end{cases}$
eltwise_clip	$d = \begin{cases} \beta & \text{if } s > \beta \geq \alpha \\ s & \text{if } \alpha < s \leq \beta \\ \alpha & \text{if } s \leq \alpha \end{cases}$
eltwise_elu, *eltwise_elu_use_dst_for_bwd*	$d = \begin{cases} s & \text{if } s > 0 \\ \alpha(e^s - 1) & \text{if } s \leq 0 \end{cases}$
eltwise_exp, *eltwise_exp_use_dst_for_bwd*	$d = e^s$
eltwise_gelu_erf	$d = 0.5s(1 + erf[\frac{s}{\sqrt{2}}])$
eltwise_gelu_tanh	$d = 0.5s(1 + tanh[\sqrt{\frac{2}{\pi}}(s + 0.044715s^3)])$
eltwise_linear	$d = \alpha s + \beta$
eltwise_log	$d = \log_e s$
eltwise_logistic, *eltwise_logistic_use_dst_for_bwd*	$d = \frac{1}{1+e^{-s}}$
eltwise_pow	$d = \alpha s^\beta$
eltwise_relu, *eltwise_relu_use_dst_for_bwd*	$d = \begin{cases} s & \text{if } s > 0 \\ \alpha s & \text{if } s \leq 0 \end{cases}$
eltwise_round	$d = round(s)$
eltwise_soft_relu	$d = \log_e(1 + e^s)$
eltwise_sqrt, *eltwise_sqrt_use_dst_for_bwd*	$d = \sqrt{s}$
eltwise_square	$d = s^2$
eltwise_swish	$d = \frac{s}{1+e^{-\alpha s}}$
eltwise_tanh, *eltwise_tanh_use_dst_for_bwd*	$d = \tanh s$

Backward

The backward propagation computes $\mathrm{diff\_src}(\bar{s})$, based on $\mathrm{diff\_dst}(\bar{s})$ and $\mathrm{src}(\bar{s})$. However, some operations support a computation using $\mathrm{dst}(\bar{s})$ memory produced during forward propagation. Refer to the table above for a list of operations supporting destination as input memory and the corresponding formulas.

The following backward operations are supported. Here s, d, ds and dd denote src, dst, $\mathrm{diff\_src}$, and a $\mathrm{diff\_dst}$ tensor values respectively.

Elementwise algorithm	Backward formula
eltwise_abs	$ds = \begin{cases} dd & \text{if } s > 0 \\ -dd & \text{if } s < 0 \\ 0 & \text{if } s = 0 \end{cases}$
eltwise_bounded_relu	$ds = \begin{cases} dd & \text{if } 0 < s \leq \alpha, \\ 0 & \text{otherwise} \end{cases}$
eltwise_clip	$ds = \begin{cases} dd & \text{if } \alpha < s \leq \beta \\ 0 & \text{otherwise} \end{cases}$
eltwise_elu	$ds = \begin{cases} dd & \text{if } s > 0 \\ dd \cdot \alpha e^s & \text{if } s \leq 0 \end{cases}$
eltwise_elu_use_dst_for_bwd	$ds = \begin{cases} dd & \text{if } d > 0 \\ dd \cdot (d + \alpha) & \text{if } d \leq 0 \end{cases}$ only if $\alpha \geq 0$
eltwise_exp	$ds = dd \cdot e^s$
eltwise_exp_use_dst_for_bwd	$ds = dd \cdot d$
eltwise_gelu_erf	$ds = dd \cdot \left(0.5 + 0.5 \operatorname{erf}\left(\frac{s}{\sqrt{2}}\right) + \frac{s}{\sqrt{2\pi}} e^{-0.5 s^2}\right)$
eltwise_gelu_tanh	$ds = dd \cdot 0.5(1 + \tanh[\sqrt{\frac{2}{\pi}}(s + 0.044715 s^3)]) \cdot (1 + \sqrt{\frac{2}{\pi}}(s + 0.134145 s^3) \cdot (1 - \tanh[\sqrt{\frac{2}{\pi}}(s + 0.044715 s^3)]))$
eltwise_linear	$ds = \alpha \cdot dd$
eltwise_log	$ds = \frac{dd}{s}$
eltwise_logistic	$ds = \frac{dd}{1+e^{-s}} \cdot (1 - \frac{1}{1+e^{-s}})$
eltwise_logistic_use_dst_for_bwd	$ds = dd \cdot d \cdot (1 - d)$
eltwise_pow	$ds = dd \cdot \alpha \beta s^{\beta - 1}$
eltwise_relu	$ds = \begin{cases} dd & \text{if } s > 0 \\ \alpha \cdot dd & \text{if } s \leq 0 \end{cases}$
eltwise_relu_use_dst_for_bwd	$ds = \begin{cases} dd & \text{if } d > 0 \\ \alpha \cdot dd & \text{if } d \leq 0 \end{cases}$ only if $alpha \geq 0$
eltwise_soft_relu	$ds = \frac{dd}{1+e^{-s}}$
eltwise_sqrt	$ds = \frac{dd}{2\sqrt{s}}$
eltwise_sqrt_use_dst_for_bwd	$ds = \frac{dd}{2d}$
eltwise_square	$ds = dd \cdot 2s$
eltwise_swish	$ds = \frac{dd}{1+e^{-\alpha s}}(1 + \alpha s(1 - \frac{1}{1+e^{-\alpha s}}))$
eltwise_tanh	$ds = dd \cdot (1 - \tanh^2 s)$
eltwise_tanh_use_dst_for_bwd	$ds = dd \cdot (1 - d^2)$

Difference Between Forward Training and Forward Inference

There is no difference between the #dnnl_forward_training and #dnnl_forward_inference propagation kinds.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. The *dnnl::eltwise_forward::desc::desc()* and *dnnl::eltwise_backward::desc::desc()* constructors take both parameters α, and β. These parameters are ignored if they are unused by the algorithm.
2. The memory format and data type for src and dst are assumed to be the same, and in the API are typically denoted as `data` (for example *dnnl::eltwise_forward::desc::desc()* has a `data_desc` argument). The same holds for $\mathrm{diff\_src}$ and $\mathrm{diff\_dst}$. The corresponding memory descriptors are denoted as `diff_data_desc`.
3. Both forward and backward propagation support in-place operations, meaning that src can be used as input and output for forward propagation, and $\mathrm{diff\_dst}$ can be used as input and output for backward propagation. In case of an in-place operation, the original data will be overwritten. Note, however, that some algorithms for backward propagation require original src, hence the corresponding forward propagation should not be performed in-place for those algorithms. Algorithms that use dst for backward propagation can be safely done in-place.
4. For some operations it might be beneficial to compute backward propagation based on $\mathrm{dst}(\bar{s})$, rather than on $\mathrm{src}(\bar{s})$, for improved performance.

Note: For operations supporting destination memory as input, dst can be used instead of src when backward propagation is computed. This enables several performance optimizations (see the tips below).

Data Type Support

The eltwise primitive should support the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination	Intermediate data type
forward / backward	*f32*, *bf16*	*f32*
forward	*f16*	*f16*
forward	*s32* / *s8* / *u8*	*f32*

Here the intermediate data type means that the values coming in are first converted to the intermediate data type, then the operation is applied, and finally the result is converted to the output data type.

Data Representation

The eltwise primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

Post-ops and Attributes

The eltwise primitive does not have to support any post-ops or attributes.

API

struct dnnl::**eltwise_forward** : **public** dnnl::*primitive*

Elementwise unary operation forward propagation primitive.

Public Functions

eltwise_forward()

Default constructor. Produces an empty object.

eltwise_forward(**const** *primitive_desc* &*pd*)

Constructs an eltwise forward propagation primitive.

Parameters

- pd: Primitive descriptor for an eltwise forward propagation primitive.

struct desc

Descriptor for an elementwise forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*data_desc*, float *alpha* = 0, float *beta* = 0)

Constructs a descriptor for an elementwise forward propagation primitive.

Parameters

- aprop_kind: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- aalgorithm: Elementwise algorithm kind.
- data_desc: Source and destination memory descriptors.
- alpha: The alpha parameter for the elementwise operation. Specific meaning depends on the algorithm.
- beta: The beta parameter for the elementwise operation. Specific meaning depends on the algorithm.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for an elementwise forward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for an elementwise forward propagation primitive.

Parameters
- `adesc`: Descriptor for an elementwise forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for an elementwise forward propagation primitive.

Parameters
- `adesc`: Descriptor for an elementwise forward propagation primitive.
- `aengine`: Engine to use.
- `attr`: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

struct dnnl::**eltwise_backward** : **public** dnnl::*primitive*
Elementwise unary operation backward propagation primitive.

See *eltwise_forward*

Public Functions

eltwise_backward()
Default constructor. Produces an empty object.

eltwise_backward(**const** *primitive_desc* &*pd*)
Constructs an eltwise backward propagation primitive.

Parameters
- `pd`: Primitive descriptor for an eltwise backward propagation primitive.

struct desc
Descriptor for an elementwise backward propagation primitive.

Public Functions

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*diff_data_desc*, **const** *memory*::*desc* &*data_desc*, float *alpha* = 0, float *beta* = 0)

Constructs a descriptor for an elementwise backward propagation primitive.

Parameters

- `aalgorithm`: Elementwise algorithm kind.
- `diff_data_desc`: Diff source and destination memory descriptors.
- `data_desc`: Source memory descriptor.
- `alpha`: The alpha parameter for the elementwise operation. Specific meaning depends on the algorithm.
- `beta`: The beta parameter for the elementwise operation. Specific meaning depends on the algorithm.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for eltwise backward propagation.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *eltwise_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for an elementwise backward propagation primitive.

Parameters

- `adesc`: Descriptor for an elementwise backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an elementwise forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *eltwise_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for an elementwise backward propagation primitive.

Parameters

- `adesc`: Descriptor for an elementwise backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an elementwise forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_src_desc**() **const**

Returns a diff source memory descriptor.

Return Diff source memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() **const**
Returns a diff destination memory descriptor.
Return Diff destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination parameter.

5.5.8 Inner Product

The inner product primitive (sometimes called *fully connected layer*) treats each activation in the minibatch as a vector and computes its product with a weights 2D tensor producing a 2D tensor as an output.

Forward

Let src, $\mathrm{weights}$, bias and dst be $N \times IC$, $OC \times IC$, OC, and $N \times OC$ tensors, respectively. Variable names follow the standard *Conventions*. Then:

$$\mathrm{dst}(n, oc) = \mathrm{bias}(oc) + \sum_{ic=0}^{IC-1} \mathrm{src}(n, ic) \cdot \mathrm{weights}(oc, ic)$$

In cases where the src and $\mathrm{weights}$ tensors have spatial dimensions, they are flattened to 2D. For example, if they are 4D $N \times IC' \times IH \times IW$ and $OC \times IC' \times KH \times KW$ tensors, then the formula above is applied with $IC = IC' \cdot IH \cdot IW$. In such cases, the src and $\mathrm{weights}$ tensors must have equal spatial dimensions (e.g. $KH = IH$ and $KW = IW$ for 4D tensors).

Difference Between Forward Training and Forward Inference

There is no difference between the `forward_training` and `forward_inference` propagation kinds.

Backward

The backward propagation computes $\mathrm{diff\_src}$ based on $\mathrm{diff\_dst}$ and $\mathrm{weights}$.

The weights update computes $\mathrm{diff\_weights}$ and $\mathrm{diff\_bias}$ based on $\mathrm{diff\_dst}$ and src.

Note: The *optimized* memory formats src and $\mathrm{weights}$ might be different on forward propagation, backward propagation, and weights update.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
weights	*DNNL_ARG_WEIGHTS*
bias	*DNNL_ARG_BIAS*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_weights	*DNNL_ARG_DIFF_WEIGHTS*
diff_bias	*DNNL_ARG_DIFF_BIAS*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

N/A

Data Types Support

Inner product primitive supports the following combination of data types for source, destination, weights, and bias.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source	Weights	Destination	Bias
forward / backward	*f32*	*f32*	*f32*	*f32*
forward	*f16*	*f16*	*f16*	*f16*
forward	*u8*, *s8*	*s8*	*u8*, *s8*, *s32*, *f32*	*u8*, *s8*, *s32*, *f32*
forward	*bf16*	*bf16*	*f32*, *bf16*	*f32*, *bf16*
backward	*f32*, *bf16*	*bf16*	*bf16*	
weights update	*bf16*	*f32*, *bf16*	*bf16*	*f32*, *bf16*

Data Representation

Like other CNN primitives, the inner product primitive expects the following tensors:

Spatial	Source	Destination	Weights
1D	$N \times C \times W$	$N \times C$	$OC \times IC \times KW$
2D	$N \times C \times H \times W$	$N \times C$	$OC \times IC \times KH \times KW$
3D	$N \times C \times D \times H \times W$	$N \times C$	$OC \times IC \times KD \times KH \times KW$

Memory format of data and weights memory objects is critical for inner product primitive performance. In the oneDNN programming model, inner product primitive is one of the few primitives that support the placeholder format *any* and can define data and weight memory objects formats based on the primitive parameters. When using *any* it is necessary to first create an inner product primitive descriptor and then query it for the actual data and weight memory objects formats.

The table below shows the combinations for which **plain** memory formats the inner product primitive is optimized for. For the destination tensor (which is always $N \times C$) the memory format is always *nc* (*ab*).

Spatial	Source / Weights logical tensor	Implementation optimized for memory formats
0D	NC / OI	*nc* (*ab*) / *oi* (*ab*)
0D	NC / OI	*nc* (*ab*) / *io* (*ba*)
1D	NCW / OIW	*ncw* (*abc*) / *oiw* (*abc*)
1D	NCW / OIW	*nwc* (*acb*) / *wio* (*cba*)
2D	NCHW / OIHW	*nchw* (*abcd*) / *oihw* (*abcd*)
2D	NCHW / OIHW	*nhwc* (*acdb*) / *hwio* (*cdba*)
3D	NCDHW / OIDHW	*ncdhw* (*abcde*) / *oidhw* (*abcde*)
3D	NCDHW / OIDHW	*ndhwc* (*acdeb*) / *dhwio* (*cdeba*)

Post-ops and Attributes

The following post-ops should be supported by inner product primitives:

Propagation	Type	Operation	Description	Restrictions
forward	attribute	*Output scale*	Scales the result of inner product by given scale factor(s)	int8 inner products only
forward	post-op	*Eltwise*	Applies an elementwise operation to the result	
forward	post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it	

API

struct dnnl::**inner_product_forward** : **public** dnnl::*primitive*

Inner product forward propagation primitive.

Public Functions

inner_product_forward()

Default constructor. Produces an empty object.

inner_product_forward(**const** *primitive_desc* &*pd*)

Constructs an inner product forward propagation primitive.

Parameters

- pd: Primitive descriptor for an inner product forward propagation primitive.

struct desc

Descriptor for an inner product forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_desc*)

Constructs a descriptor for an inner product forward propagation primitive with bias.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `src_desc`: Memory descriptor for src.
- `weights_desc`: Memory descriptor for diff weights.
- `bias_desc`: Memory descriptor for diff bias.
- `dst_desc`: Memory descriptor for diff dst.

desc(*prop_kind aprop_kind*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*dst_desc*)

Constructs a descriptor for an inner product forward propagation primitive without bias.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `src_desc`: Memory descriptor for src.
- `weights_desc`: Memory descriptor for diff weights.
- `dst_desc`: Memory descriptor for dst.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for an inner product forward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for an inner product forward propagation primitive.

Parameters

- `adesc`: Descriptor for an inner product forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for an inner product forward propagation primitive.

Parameters

- `adesc`: Descriptor for an inner product forward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() **const**
Returns a weights memory descriptor.
Return Weights memory descriptor.
Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **bias_desc**() **const**
Returns the bias memory descriptor.
Return The bias memory descriptor.
Return A zero memory descriptor of the primitive does not have a bias parameter.

struct dnnl::**inner_product_backward_data** : **public** dnnl::*primitive*
Inner product backward propagation primitive.

Public Functions

inner_product_backward_data()
Default constructor. Produces an empty object.

inner_product_backward_data(**const** *primitive_desc* &*pd*)
Constructs an inner product backward propagation primitive.

Parameters

- pd: Primitive descriptor for an inner product backward propagation primitive.

struct desc
Descriptor for an inner product backward propagation primitive.

Public Functions

desc(**const** *memory*::*desc* &*diff_src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*)
Constructs a descriptor for an inner product backward propagation primitive.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of format_tag.

Parameters

- diff_src_desc: Memory descriptor for diff src.
- weights_desc: Memory descriptor for weights.
- diff_dst_desc: Memory descriptor for diff dst.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for an inner product backward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *inner_product_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for an inner product backward propagation primitive.

Parameters
- adesc: Descriptor for an inner product backward propagation primitive.
- aengine: Engine to use.
- hint_fwd_pd: Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *inner_product_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for an inner product backward propagation primitive.

Parameters
- adesc: Descriptor for an inner product backward propagation primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.
- hint_fwd_pd: Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() **const**
Returns a diff source memory descriptor.
Return Diff source memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **weights_desc**() **const**
Returns a weights memory descriptor.
Return Weights memory descriptor.
Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **diff_dst_desc**() **const**
Returns a diff destination memory descriptor.
Return Diff destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination parameter.

struct dnnl::**inner_product_backward_weights** : **public** dnnl::*primitive*
Inner product weights gradient primitive.

Public Functions

inner_product_backward_weights()
Default constructor. Produces an empty object.

inner_product_backward_weights(**const** *primitive_desc* &*pd*)
Constructs an inner product weights gradient primitive.

Parameters

- pd: Primitive descriptor for an inner product weights gradient primitive.

struct desc
Descriptor for an inner product weights gradient primitive.

Public Functions

desc(**const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_desc*)
Constructs a descriptor for an inner product descriptor weights update primitive with bias.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of format_tag.

Parameters

- src_desc: Memory descriptor for src.
- diff_weights_desc: Memory descriptor for diff weights.
- diff_bias_desc: Memory descriptor for diff bias.
- diff_dst_desc: Memory descriptor for diff dst.

desc(**const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*diff_weights_desc*, **const** *memory*::*desc* &*diff_dst_desc*)
Constructs a descriptor for an inner product descriptor weights update primitive without bias.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of format_tag.

Parameters

- src_desc: Memory descriptor for src.
- diff_weights_desc: Memory descriptor for diff weights.
- diff_dst_desc: Memory descriptor for diff dst.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for an inner product weights gradient primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *inner_product_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for an inner product weights update primitive.

Parameters

- adesc: Descriptor for an inner product weights gradient primitive.
- aengine: Engine to use.
- hint_fwd_pd: Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *inner_product_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for an inner product weights update primitive.

Parameters

- `adesc`: Descriptor for an inner product weights gradient primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_weights_desc**() **const**

Returns a diff weights memory descriptor.

Return Diff weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **diff_dst_desc**() **const**

Returns a diff destination memory descriptor.

Return Diff destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_bias_desc**() **const**

Returns the diff bias memory descriptor.

Return The diff bias memory descriptor.

Return A zero memory descriptor of the primitive does not have a diff bias parameter.

5.5.9 Layer normalization

The layer normalization primitive performs a forward or backward layer normalization operation on a 2-5D data tensor.

The layer normalization operation performs normalization over the last logical axis of the data tensor and is defined by the following formulas. We show formulas only for 3D data, which are straightforward to generalize to cases of higher dimensions. Variable names follow the standard *Conventions*.

Forward

$$\mathrm{dst}(t,n,c) = \gamma(c) \cdot \frac{\mathrm{src}(t,n,c) - \mu(t,n)}{\sqrt{\sigma^2(t,n) + \varepsilon}} + \beta(c),$$

where

- $\gamma(c), \beta(c)$ are optional scale and shift for a channel (see the `use_scaleshift` flag),
- $\mu(t,n), \sigma^2(t,n)$ are mean and variance (see *use_global_stats* flag), and
- ε is a constant to improve numerical stability.

Mean and variance are computed at runtime or provided by a user. When mean and variance are computed at runtime, the following formulas are used:

- $\mu(t,n) = \frac{1}{C}\sum_c \mathrm{src}(t,n,c)$,
- $\sigma^2(t,n) = \frac{1}{C}\sum_c (\mathrm{src}(t,n,c) - \mu(t,n))^2$.

The $\gamma(c)$ and $\beta(c)$ tensors are considered learnable.

Difference Between Forward Training and Forward Inference

If mean and variance are computed at runtime (i.e., *use_global_stats* is not set), they become outputs for the propagation kind *forward_training* (because they would be required during the backward propagation). Data layout for mean and variance must be specified during initialization of the layer normalization descriptor by passing the memory descriptor for statistics (e.g., by passing `stat_desc` in *dnnl::layer_normalization_forward::desc::desc()*). Mean and variance are not exposed for the propagation kind *forward_inference*.

Backward

The backward propagation computes diff_src(t,n,c), diff_$\gamma(c)^*$, and diff_$\beta(c)^*$ based on diff_dst(t,n,c), $src(t,n,c)$, $\mu(t,n)$, $\sigma^2(t,n)$, $\gamma(c)^*$, and $\beta(c)^*$.

The tensors marked with an asterisk are used only when the primitive is configured to use $\gamma(c)$, and $\beta(c)$ (i.e., `use_scaleshift` is set).

Execution Arguments

Depending on the flags and propagation kind, the layer normalization primitive requires different inputs and outputs. For clarity, a summary is shown below.

	forward_inference	*forward_training*	*backward*	*backward_data*
none	*In*: src *Out*: dst	*In*: src *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2 *Out*: diff_src	Same as for *backward*
use_global_stats	*In*: src, μ, σ^2 *Out*: dst	*In*: src, μ, σ^2 *Out*: dst	*In*: diff_dst, src, μ, σ^2 *Out*: diff_src	Same as for *backward*
`use_scaleshift`	*In*: src, γ, β *Out*: dst	*In*: src, γ, β *Out*: dst, μ, σ^2	*In*: diff_dst, src, μ, σ^2, γ, β *Out*: diff_src, diff_γ, diff_β	Not supported
use_global_stats | `use_scaleshift`	*In*: src, μ, σ^2, γ, β *Out*: dst	*In*: src, μ, σ^2, γ, β *Out*: dst	*In*: diff_dst, src, μ, σ^2, γ, β *Out*: diff_src, diff_γ, diff_β	Not supported

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
γ, β	*DNNL_ARG_SCALE_SHIFT*
mean (μ)	*DNNL_ARG_MEAN*
variance (σ)	*DNNL_ARG_VARIANCE*
dst	*DNNL_ARG_DST*
diff_dst	*DNNL_ARG_DIFF_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_γ, diff_β	*DNNL_ARG_DIFF_SCALE_SHIFT*

Operation Details

1. The different flavors of the primitive are partially controlled by the `flags` parameter that is passed to the operation descriptor initialization function (e.g., *dnnl::layer_normalization_forward::desc::desc()*). Multiple flags can be combined using the bitwise OR operator (`|`).
2. For forward propagation, the mean and variance might be either computed at runtime (in which case they are outputs of the primitive) or provided by a user (in which case they are inputs). In the latter case, a user must set the *use_global_stats* flag. For the backward propagation, the mean and variance are always input parameters.
3. The memory format and data type for `src` and `dst` are assumed to be the same, and in the API they are typically referred to as `data` (e.g., see `data_desc` in dnnl::layer_normalization_forward::desc::desc()). The same is true for `diff_src` and `diff_dst`. The corresponding memory descriptors are referred to as `diff_data_desc`.
4. Both forward and backward propagation support in-place operations, meaning that src can be used as input and output for forward propagation, and diff_dst can be used as input and output for backward propagation. In case of an in-place operation, the original data will be overwritten. Note, however, that backward propagation requires original src, hence the corresponding forward propagation should not be performed in-place.

Data Types Support

The layer normalization supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination	Mean / Variance / ScaleShift
forward / backward	*f32*	*f32*
forward	*f16*	*f32*

Data Representation

Mean and Variance

The mean (μ) and variance (σ^2) are separate tensors with number of dimensions equal to ($data\_ndims - 1$) and size ($data\_dim[0], data\_dim[1], ..., data\_dim[ndims - 2]$).

The corresponding memory object can have an arbitrary memory format. Unless mean and variance are computed at runtime and not exposed (i.e., propagation kind is *forward_inference* and *use_global_stats* is not set), the user should provide a memory descriptor for statistics when initializing the layer normalization descriptor. For best performance, it is advised to use the memory format that follows the data memory format; i.e., if the data format is *tnc*, the best performance can be expected for statistics with the *tn* format and suboptimal for statistics with the *nt* format.

Scale and Shift

If used, the scale (γ) and shift (β) are combined in a single 2D tensor of shape $2 \times C$.

The format of the corresponding memory object must be *nc* (*ab*).

Source, Destination, and Their Gradients

The layer normalization primitive works with an arbitrary data tensor; however, it was designed for RNN data tensors (i.e., *nc*, *tnc*, *ldnc*). Unlike CNN data tensors, RNN data tensors have a single feature dimension. Layer normalization performs normalization over the last logical dimension (feature dimension for RNN tensors) across non-feature dimensions.

The layer normalization primitive is optimized for the following memory formats:

Logical tensor	Implementations optimized for memory formats
NC	*nc* (*ab*)
TNC	*tnc* (*abc*), *ntc* (*bac*)
LDNC	*ldnc* (*abcd*)

API

struct dnnl::**layer_normalization_forward** : **public** dnnl::*primitive*

Layer normalization forward propagation primitive.

Public Functions

layer_normalization_forward()

Default constructor. Produces an empty object.

layer_normalization_forward(**const** *primitive_desc* &*pd*)

Constructs a layer normalization forward propagation primitive.

Parameters

- pd: Primitive descriptor for a layer normalization forward propagation primitive.

struct desc

Descriptor for a layer normalization forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, **const** *memory*::*desc* &*data_desc*, **const** *memory*::*desc* &*stat_desc*, float *epsilon*, *normalization_flags flags*)
Constructs a descriptor for layer normalization forward propagation primitive.

Parameters
- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `data_desc`: Source and destination memory descriptor.
- `stat_desc`: Statistics memory descriptors.
- `epsilon`: Layer normalization epsilon parameter.
- `flags`: Layer normalization flags (*dnnl::normalization_flags*).

desc(*prop_kind aprop_kind*, **const** *memory*::*desc* &*data_desc*, float *epsilon*, *normalization_flags flags*)
Constructs a descriptor for layer normalization forward propagation primitive.

Parameters
- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `data_desc`: Source and destination memory descriptor.
- `epsilon`: Layer normalization epsilon parameter.
- `flags`: Layer normalization flags (*dnnl::normalization_flags*).

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for a layer normalization forward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a layer normalization forward propagation primitive.

Parameters
- `adesc`: Descriptor for a layer normalization forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a layer normalization forward propagation primitive.

Parameters
- `adesc`: Descriptor for a layer normalization forward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **weights_desc**() **const**
Returns a weights memory descriptor.
Return Weights memory descriptor.
Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **mean_desc**() **const**
Returns memory descriptor for mean.
Return Memory descriptor for mean.

memory::*desc* **variance_desc**() **const**
Returns memory descriptor for variance.
Return Memory descriptor for variance.

struct dnnl::**layer_normalization_backward** : **public** dnnl::*primitive*
Layer normalization backward propagation primitive.

Public Functions

layer_normalization_backward()
Default constructor. Produces an empty object.

layer_normalization_backward(**const** *primitive_desc* &*pd*)
Constructs a layer normalization backward propagation primitive.

Parameters

- pd: Primitive descriptor for a layer normalization backward propagation primitive.

struct desc
Descriptor for a layer normalization backward propagation primitive.

Public Functions

desc(*prop_kind* *aprop_kind*, **const** *memory*::*desc* &*diff_data_desc*, **const** *memory*::*desc* &*data_desc*, **const** *memory*::*desc* &*stat_desc*, float *epsilon*, *normalization_flags* *flags*)
Constructs a descriptor for layer normalization backward propagation primitive.

Parameters

- aprop_kind: Propagation kind. Possible values are *dnnl::prop_kind::backward_data* and *dnnl::prop_kind::backward* (diffs for all parameters are computed in this case).
- diff_data_desc: Diff source and diff destination memory descriptor.
- data_desc: Source memory descriptor.
- stat_desc: Statistics memory descriptors.
- epsilon: Layer normalization epsilon parameter.
- flags: Layer normalization flags (*dnnl::normalization_flags*).

desc(*prop_kind aprop_kind*, **const** *memory*::*desc* &*diff_data_desc*, **const** *memory*::*desc* &*data_desc*, float *epsilon*, *normalization_flags flags*)

Constructs a descriptor for layer normalization backward propagation primitive.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::backward_data* and *dnnl::prop_kind::backward* (diffs for all parameters are computed in this case).
- `diff_data_desc`: Diff source and diff destination memory descriptor.
- `data_desc`: Source memory descriptor.
- `epsilon`: Layer normalization epsilon parameter.
- `flags`: Layer normalization flags (*dnnl::normalization_flags*).

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a layer normalization backward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *layer_normalization_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a layer normalization backward propagation primitive.

Parameters

- `adesc`: Descriptor for a layer normalization backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a layer normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *layer_normalization_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a layer normalization backward propagation primitive.

Parameters

- `adesc`: Descriptor for a layer normalization backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a layer normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() **const**

Returns a weights memory descriptor.

Return Weights memory descriptor.

Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **diff_src_desc**() **const**
Returns a diff source memory descriptor.
Return Diff source memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() **const**
Returns a diff destination memory descriptor.
Return Diff destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **diff_weights_desc**() **const**
Returns a diff weights memory descriptor.
Return Diff weights memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff weights parameter.

memory::*desc* **mean_desc**() **const**
Returns memory descriptor for mean.
Return Memory descriptor for mean.

memory::*desc* **variance_desc**() **const**
Returns memory descriptor for variance.
Return Memory descriptor for variance.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

5.5.10 LogSoftmax

The logsoftmax primitive performs softmax along a particular axis on data with arbitrary dimensions followed by the logarithm function. All other axes are treated as independent (batch).

In general form, the operation is defined by the following formulas. Variable names follow the standard *Conventions*.

Forward

The second form is used as more numerically stable:

$$\begin{aligned}\operatorname{dst}(\overline{ou}, c, \overline{in}) &= \ln\left(\frac{e^{\operatorname{src}(\overline{ou}, c, \overline{in}) - \nu(\overline{ou}, \overline{in})}}{\sum\limits_{ic} e^{\operatorname{src}(\overline{ou}, ic, \overline{in}) - \nu(\overline{ou}, \overline{in})}}\right) \\ &= \left(\operatorname{src}(\overline{ou}, c, \overline{in}) - \nu(\overline{ou}, \overline{in})\right) - \ln\left(\sum_{ic} e^{\operatorname{src}(\overline{ou}, ic, \overline{in}) - \nu(\overline{ou}, \overline{in})}\right),\end{aligned}$$

where

- c axis over which the logsoftmax computation is computed on,
- $\overline{ou}$ is the outermost index (to the left of logsoftmax axis),
- $\overline{in}$ is the innermost index (to the right of logsoftmax axis), and
- ν is used to produce more accurate results and defined as:

$$\nu(\overline{ou}, \overline{in}) = \max_{ic} \operatorname{src}(\overline{ou}, ic, \overline{in})$$

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes $\operatorname{diff\_src}(ou, c, in)$, based on $\operatorname{diff\_dst}(ou, c, in)$ and $\operatorname{dst}(ou, c, in)$.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

Both forward and backward propagation support in-place operations, meaning that `src` can be used as input and output for forward propagation, and `diff_dst` can be used as input and output for backward propagation. In case of in-place operation, the original data will be overwritten.

Post-ops and Attributes

The logsoftmax primitive does not support any post-ops or attributes.

Data Type Support

The logsoftmax primitive supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*bf16*, *f32*

Data Representation

Source, Destination, and Their Gradients

The logsoftmax primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions. However, the logsoftmax axis is typically referred to as channels (hence in formulas we use c).

API

struct dnnl::**logsoftmax_forward** : **public** dnnl::*primitive*

Logsoftmax forward propagation primitive.

Public Functions

logsoftmax_forward()

Default constructor. Produces an empty object.

logsoftmax_forward(**const** *primitive_desc* &*pd*)

Constructs a logsoftmax forward propagation primitive.

Parameters

- pd: Primitive descriptor for a logsoftmax forward propagation primitive.

struct desc

Descriptor for a logsoftmax forward propagation primitive.

Public Functions

desc()

Default constructor. Produces an empty object.

desc(*prop_kind aprop_kind*, **const** *memory*::*desc* &*data_desc*, int *logsoftmax_axis*)

Constructs a descriptor for a logsoftmax forward propagation primitive.

Parameters

- aprop_kind: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- data_desc: Source and destination memory descriptor.
- logsoftmax_axis: Axis over which softmax is computed.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a logsoftmax forward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a logsoftmax forward propagation primitive.

Parameters

- adesc: descriptor for a logsoftmax forward propagation primitive.
- aengine: Engine to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a logsoftmax forward propagation primitive.

Parameters

- `adesc`: Descriptor for a logsoftmax forward propagation primitive.
- `aengine`: Engine to use.
- `attr`: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

struct dnnl::**logsoftmax_backward** : **public** dnnl::*primitive*

Logsoftmax backward propagation primitive.

Public Functions

logsoftmax_backward()

Default constructor. Produces an empty object.

logsoftmax_backward(**const** *primitive_desc* &*pd*)

Constructs a logsoftmax backward propagation primitive.

Parameters

- `pd`: Primitive descriptor for a logsoftmax backward propagation primitive.

struct desc

Descriptor for a logsoftmax backward propagation primitive.

Public Functions

desc()

Default constructor. Produces an empty object.

desc(**const** *memory*::*desc* &*diff_data_desc*, **const** *memory*::*desc* &*data_desc*, int *logsoftmax_axis*)

Constructs a descriptor for a logsoftmax backward propagation primitive.

Parameters

- `diff_data_desc`: Diff source and diff destination memory descriptors.
- `data_desc`: Destination memory descriptor.
- `logsoftmax_axis`: Axis over which softmax is computed.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a logsoftmax backward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *logsoftmax_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a logsoftmax backward propagation primitive.

Parameters

- `adesc`: Descriptor for a logsoftmax backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a logsoftmax forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *logsoftmax_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a logsoftmax backward propagation primitive.

Parameters

- `adesc`: Descriptor for a logsoftmax backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a logsoftmax forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **dst_desc**() **const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **diff_src_desc**() **const**

Returns a diff source memory descriptor.

Return Diff source memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() **const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

5.5.11 Local Response Normalization

The LRN primitive performs a forward or backward local response normalization operation defined by the following formulas. Variable names follow the standard *Conventions*.

Forward

LRN *across channels*:

$$\mathrm{dst}(n,c,h,w) = \left\{k + \frac{\alpha}{n_l} \sum_{i=-(n_l-1)/2}^{(n_l+1)/2-1} (\mathrm{src}(n,c+i,h,w))^2\right\}^{-\beta} \cdot \mathrm{src}(n,c,h,w),$$

LRN *within channel*:

$$\mathrm{dst}(n,c,h,w) = \left\{k + \frac{\alpha}{n_l} \sum_{i=-(n_l-1)/2}^{(n_l+1)/2-1} \sum_{j=-(n_l-1)/2}^{(n_l+1)/2-1} (\mathrm{src}(n,c,h+i,w+j))^2\right\}^{-\beta} \cdot \mathrm{src}(n,c,h,w),$$

where n_l is the `local_size`. Formulas are provided for 2D spatial data case.

Backward

The backward propagation computes $\mathrm{diff\_src}(n,c,h,w)$, based on $\mathrm{diff\_dst}(n,c,h,w)$ and $\mathrm{src}(n,c,h,w)$.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
workspace	*DNNL_ARG_WORKSPACE*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. During training, LRN might or might not require a workspace on forward and backward passes. The behavior is implementation specific. Optimized implementations typically require a workspace and use it to save some intermediate results from the forward pass that accelerate computations on the backward pass. To check whether a workspace is required, query the LRN primitive descriptor for the workspace. Success indicates that the workspace is required and its description will be returned.
2. The memory format and data type for `src` and `dst` are assumed to be the same, and in the API are typically referred to as `data` (e.g., see `data_desc` in dnnl::lrn_forward::desc::desc()). The same holds for `diff_src` and `diff_dst`. The corresponding memory descriptors are referred to as `diff_data_desc`.

Data Type Support

The LRN primitive supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*f32*, *bf16*
forward	*f16*

Data Representation

Source, Destination, and Their Gradients

Like most other primitives, the LRN primitive expects the following tensors:

Spatial	Source / Destination
0D	$N \times C$
1D	$N \times C \times W$
2D	$N \times C \times H \times W$
3D	$N \times C \times D \times H \times W$

The LRN primitive is optimized for the following memory formats:

Spatial	Logical tensor	Implementations optimized for memory formats
2D	NCHW	*nchw* (*abcd*), *nhwc* (*acdb*), *optimized*

Here *optimized* means the format chosen by the preceding compute-intensive primitive.

Post-ops and Attributes

The LRN primitive does not support any post-ops or attributes.

API

struct dnnl::**lrn_forward** : **public** dnnl::*primitive*

Local response normalization (LRN) forward propagation primitive.

Public Functions

lrn_forward()
Default constructor. Produces an empty object.

lrn_forward(**const** *primitive_desc* &*pd*)
Constructs an LRN forward propagation primitive.

Parameters

- `pd`: Primitive descriptor for an LRN forward propagation primitive.

struct desc
Descriptor for an LRN forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*data_desc*, *memory*::*dim local_size*, float *alpha*, float *beta*, float *k* = 1.f)
Constructs a descriptor for a LRN forward propagation primitive.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: LRN algorithm kind: either *dnnl::algorithm::lrn_across_channels*, or *dnnl::algorithm::lrn_within_channel*.
- `data_desc`: Source and destination memory descriptors.
- `local_size`: Regularization local size.
- `alpha`: The alpha regularization parameter.
- `beta`: The beta regularization parameter.
- `k`: The k regularization parameter.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for an LRN forward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for an LRN forward propagation primitive.

Parameters

- `adesc`: Descriptor for an LRN forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for an LRN forward propagation primitive.

Parameters

- `adesc`: Descriptor for an LRN forward propagation primitive.
- `aengine`: Engine to use.
- `attr`: Primitive attributes to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

struct dnnl::**lrn_backward** : **public** dnnl::*primitive*
Local response normalization (LRN) backward propagation primitive.

Public Functions

lrn_backward()
Default constructor. Produces an empty object.

lrn_backward(**const** *primitive_desc* &*pd*)
Constructs an LRN backward propagation primitive.

Parameters

- `pd`: Primitive descriptor for an LRN backward propagation primitive.

struct desc
Descriptor for an LRN backward propagation primitive.

Public Functions

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*data_desc*, **const** *memory*::*desc* &*diff_data_desc*, *memory*::*dim local_size*, float *alpha*, float *beta*, float *k* = 1.f)
Constructs a descriptor for an LRN backward propagation primitive.

Parameters

- `aalgorithm`: LRN algorithm kind: either *dnnl::algorithm::lrn_across_channels*, or *dnnl::algorithm::lrn_within_channel*.
- `diff_data_desc`: Diff source and diff destination memory descriptor.
- `data_desc`: Source memory descriptor.
- `local_size`: Regularization local size.
- `alpha`: The alpha regularization parameter.
- `beta`: The beta regularization parameter.
- `k`: The k regularization parameter.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for an LRN backward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *lrn_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for an LRN backward propagation primitive.

Parameters

- `adesc`: Descriptor for an LRN backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an LRN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *lrn_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for an LRN backward propagation primitive.

Parameters

- `adesc`: Descriptor for an LRN backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an LRN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **diff_dst_desc**() **const**
Returns a diff destination memory descriptor.
Return Diff destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

5.5.12 Matrix Multiplication

The matrix multiplication (MatMul) primitive computes the product of two 2D tensors with optional bias addition. Variable names follow the standard *Conventions*.

$$\mathrm{dst}(m,n) = \sum_{k=0}^{K} (\mathrm{src}(m,k) \cdot \mathrm{weights}(k,n)) + \mathrm{bias}(m,n)$$

The MatMul primitive also supports batching multiple independent matrix multiplication operations, in which case the tensors must be 3D:

$$\mathrm{dst}(mb,m,n) = \sum_{k=0}^{K} (\mathrm{src}(mb,m,k) \cdot \mathrm{weights}(mb,k,n)) + \mathrm{bias}(mb,m,n)$$

The bias tensor is optional and supports implicit broadcast semantics: any of its dimensions can be 1 and the same value would be used across the corresponding dimension. However, bias must have the same number of dimensions as the dst.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
weights	*DNNL_ARG_WEIGHTS*
bias	*DNNL_ARG_BIAS*
dst	*DNNL_ARG_DST*

Operation Details

The MatMul primitive supports input and output tensors with run-time specified shapes and memory formats. The run-time specified dimensions or strides are specified using the *DNNL_RUNTIME_DIM_VAL* wildcard value during the primitive initialization and creation stage. At the execution stage, the user must pass fully specified memory objects so that the primitive is able to perform the computations. Note that the less information about shapes or format is available at the creation stage, the less performant execution will be. In particular, if the shape is not known at creation stage, one cannot use the special format tag *any* to enable an implementation to choose the most appropriate memory format for the corresponding input or output shapes. On the other hand, run-time specified shapes enable users to create a primitive once and use it in different situations.

Data Types Support

The MatMul primitive supports the following combinations of data types for source, destination, weights, and bias tensors.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Source	Weights	Destination	Bias
f32	*f32*	*f32*	*f32*
f16	*f16*	*f16*	*f16*
bf16	*bf16*	*bf16*	*bf16*, *f32*
u8, *s8*	*s8*, *u8*	*u8*, *s8*, *s32*, *f32*	*u8*, *s8*, *s32*, *f32*

Data Representation

The MatMul primitive expects the following tensors:

Dims	Source	Weights	Destination	Bias (optional)
2D	$M \times K$	$K \times N$	$M \times N$	$(M$ or $1) \times (N$ or $1)$
3D	$MB \times M \times K$	$MB \times K \times N$	$MB \times M \times N$	$(MB$ or $1) \times (M$ or $1) \times (N$ or $1)$

The MatMul primitive is generally optimized for the case in which memory objects use plain memory formats (with some restrictions; see the table below). However, it is recommended to use the placeholder memory format *any* if an input tensor is reused across multiple executions. In this case, the primitive will set the most appropriate memory format for the corresponding input tensor.

The table below shows the combinations of memory formats for which the MatMul primitive is optimized. The memory format of the destination tensor should always be *ab* for the 2D case and *abc* for the 3D one.

Dims	Logical tensors	MatMul is optimized for the following memory formats
2D	Source: $M \times K$, Weights: $K \times N$	Source: *ab* or *ba*, Weights: *ab* or *ba*
3D	Source: $MB \times M \times K$, Weights: $MB \times K \times N$	Source: *abc* or *acb*, Weights: *abc* or *acb*

Attributes and Post-ops

Attributes and post-ops enable modifying the behavior of the MatMul primitive. The following attributes and post-ops are supported:

Type	Operation	Description	Restrictions
Attribute	*Output scales*	Scales the result by given scale factor(s)	
Attribute	*Zero points*	Sets zero point(s) for the corresponding tensors	Int8 computations only
Post-op	*Eltwise*	Applies an elementwise operation to the result	
Post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it	

To facilitate dynamic quantization, the primitive should support run-time output scales. That means a user could configure attributes with output scales set to the *DNNL_RUNTIME_F32_VAL* wildcard value instead of the actual scales, if the scales are not known at the primitive descriptor creation stage. In this case, the user must provide the scales as an additional input memory object with argument *DNNL_ARG_ATTR_OUTPUT_SCALES* during the execution stage.

Similarly to run-time output scales, the primitive supports run-time zero points. The wildcard value for zero points is *DNNL_RUNTIME_S32_VAL*. During the execution stage, the corresponding memory object needs to be passed in the argument with index set to (`DNNL_ARG_ATTR_ZERO_POINTS | DNNL_ARG_${MEMORY}`). For instance,

source tensor zero points memory argument would be passed with index (`DNNL_ARG_ATTR_ZERO_POINTS | DNNL_ARG_SRC`).

API

struct `dnnl::matmul` : **public** dnnl::*primitive*
Matrix multiplication (matmul) primitive.

Public Functions

matmul()
Default constructor. Produces an empty object.

matmul(**const** *primitive_desc* &*pd*)
Constructs a matmul primitive.

Parameters

- `pd`: Primitive descriptor for a matmul primitive.

struct desc
Descriptor for a matmul primitive.

Public Functions

desc(**const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*dst_desc*)
Constructs a descriptor for a matmul primitive.

Parameters

- `src_desc`: Memory descriptor for source (matrix A).
- `weights_desc`: Memory descriptor for weights (matrix B).
- `dst_desc`: Memory descriptor for destination (matrix C).

desc(**const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*weights_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_desc*)
Constructs a descriptor for a matmul primitive.

Parameters

- `src_desc`: Memory descriptor for source (matrix A).
- `weights_desc`: Memory descriptor for weights (matrix B).
- `dst_desc`: Memory descriptor for destination (matrix C).
- `bias_desc`: Memory descriptor for bias.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for a matmul primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a matmul primitive.

Parameters
- `adesc`: Descriptor for a matmul primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a matmul primitive.

Parameters
- `adesc`: Descriptor for a matmul primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **weights_desc**() **const**
Returns a weights memory descriptor.
Return Weights memory descriptor.
Return A zero memory descriptor if the primitive does not have a weights parameter.

memory::*desc* **bias_desc**() **const**
Returns the bias memory descriptor.
Return The bias memory descriptor.
Return A zero memory descriptor of the primitive does not have a bias parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

5.5.13 Pooling

The pooling primitive performs forward or backward max or average pooling operation on 1D, 2D, or 3D spatial data.

The pooling operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard *Conventions*.

Forward

Max pooling:

$$\text{dst}(n, c, oh, ow) = \max_{kh,kw} \left(\text{src}(n, c, oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L)\right)$$

Average pooling:

$$\text{dst}(n, c, oh, ow) = \frac{1}{DENOM} \sum_{kh,kw} \text{src}(n, c, oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L)$$

Here output spatial dimensions are calculated similarly to how they are done for *Convolution and Deconvolution*.

Average pooling supports two algorithms:

- `pooling_avg_include_padding`, in which case $DENOM = KH \cdot KW$,
- `pooling_avg_exclude_padding`, in which case $DENOM$ equals to the size of overlap between an averaging window and images.

Difference Between Forward Training and Forward Inference

Max pooling requires a `workspace` for the `forward_training` propagation kind, and does not require it for `forward_inference` (see details below).

Backward

The backward propagation computes $\text{diff\_src}(n, c, h, w)$, based on $\text{diff\_dst}(n, c, h, w)$ and, in case of max pooling, `workspace`.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	`DNNL_ARG_SRC`
dst	`DNNL_ARG_DST`
workspace	`DNNL_ARG_WORKSPACE`
diff_src	`DNNL_ARG_DIFF_SRC`
diff_dst	`DNNL_ARG_DIFF_DST`

Operation Details

1. During training, max pooling requires a workspace on forward (`forward_training`) and backward passes to save indices where a maximum was found. The workspace format is opaque, and the indices cannot be restored from it. However, one can use backward pooling to perform up-sampling (used in some detection topologies). The workspace can be created via `dnnl::pooling_forward::primitive_desc::workspace_desc()`.
2. A user can use memory format tag `any` for `dst` memory descriptor when creating pooling forward propagation. The library would derive the appropriate format from the `src` memory descriptor. However, the `src` itself must be defined. Similarly, a user can use memory format tag `any` for the `diff_src` memory descriptor when creating pooling backward propagation.

Data Type Support

The pooling primitive supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

Propagation	Source / Destination	Accumulation data type (used for average pooling only)
forward / backward	`f32`, `bf16`	`f32`
forward	`f16`	`f16`
forward	`s8`, `u8`, `s32`	`s32`

Data Representation

Source, Destination, and Their Gradients

Like other CNN primitives, the pooling primitive expects data to be an $N \times C \times W$ tensor for the 1D spatial case, an $N \times C \times H \times W$ tensor for the 2D spatial case, and an $N \times C \times D \times H \times W$ tensor for the 3D spatial case.

The pooling primitive is optimized for the following memory formats:

Spatial	Logical tensor	Data type	Implementations optimized for memory formats
1D	NCW	f32	`ncw` (`abc`), `nwc` (`acb`), *optimized^*
1D	NCW	s32, s8, u8	`nwc` (`acb`), *optimized^*
2D	NCHW	f32	`nchw` (`abcd`), `nhwc` (`acdb`), *optimized^*
2D	NCHW	s32, s8, u8	`nhwc` (`acdb`), *optimized^*
3D	NCDHW	f32	`ncdhw` (`abcde`), `ndhwc` (`acdeb`), *optimized^*
3D	NCDHW	s32, s8, u8	`ndhwc` (`acdeb`), *optimized^*

Here *optimized^* means the format that comes out of any preceding compute-intensive primitive.

Post-ops and Attributes

The pooling primitive does not support any post-ops or attributes.

API

struct dnnl::**pooling_forward** : **public** dnnl::*primitive*

Pooling forward propagation primitive.

Public Functions

pooling_forward()
Default constructor. Produces an empty object.

pooling_forward(**const** *primitive_desc* &*pd*)
Constructs a pooling forward propagation primitive.

Parameters

- pd: Primitive descriptor for a pooling forward propagation primitive.

struct desc
Descriptor for a pooling forward propagation primitive.

Public Functions

desc(*prop_kind* *aprop_kind*, *algorithm* *aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*kernel*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)
Constructs a descriptor for pooling forward propagation primitive.

Arrays strides, kernel, padding_l, and padding_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Parameters

- aprop_kind: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- aalgorithm: Pooling algorithm kind: either *dnnl::algorithm::pooling_max*, *dnnl::algorithm::pooling_avg_include_padding*, or *dnnl::algorithm::pooling_avg* (same as *dnnl::algorithm::pooling_avg_exclude_padding*).
- src_desc: Source memory descriptor.
- dst_desc: Destination memory descriptor.
- strides: Vector of strides for spatial dimension.
- kernel: Vector of kernel spatial dimensions.
- padding_l: Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- padding_r: Vector of padding values for high indices for each spatial dimension ([[back,] bottom,] right).

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for a pooling forward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a pooling forward propagation primitive.

Parameters

- adesc: Descriptor for a pooling forward propagation primitive.
- aengine: Engine to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a pooling forward propagation primitive.

Parameters

- `adesc`: Descriptor for a pooling forward propagation primitive.
- `aengine`: Engine to use.
- `attr`: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **workspace_desc**() **const**

Returns the workspace memory descriptor.

Return Workspace memory descriptor.

Return A zero memory descriptor if the primitive does not require workspace parameter.

struct dnnl::**pooling_backward** : **public** dnnl::*primitive*

Pooling backward propagation primitive.

Public Functions

pooling_backward()

Default constructor. Produces an empty object.

pooling_backward(**const** *primitive_desc* &*pd*)

Constructs a pooling backward propagation primitive.

Parameters

- `pd`: Primitive descriptor for a pooling backward propagation primitive.

struct desc

Descriptor for a pooling backward propagation primitive.

Public Functions

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*diff_src_desc*, **const** *memory*::*desc* &*diff_dst_desc*, **const** *memory*::*dims* &*strides*, **const** *memory*::*dims* &*kernel*, **const** *memory*::*dims* &*padding_l*, **const** *memory*::*dims* &*padding_r*)

Constructs a descriptor for pooling backward propagation primitive.

Arrays `strides`, `kernel`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

Parameters

- `aalgorithm`: Pooling algorithm kind: either *dnnl::algorithm::pooling_max*, *dnnl::algorithm::pooling_avg_include_padding*, or *dnnl::algorithm::pooling_avg* (same as *dnnl::algorithm::pooling_avg_exclude_padding*).
- `diff_src_desc`: Diff source memory descriptor.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Vector of strides for spatial dimension.
- `kernel`: Vector of kernel spatial dimensions.
- `padding_l`: Vector of padding values for low indices for each spatial dimension `([[front,] top,] left)`.
- `padding_r`: Vector of padding values for high indices for each spatial dimension `([[back,] bottom,] right)`.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a pooling backward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *pooling_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a pooling backward propagation primitive.

Parameters

- `adesc`: Descriptor for a pooling backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a pooling forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *pooling_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a pooling backward propagation primitive.

Parameters

- `adesc`: Descriptor for a pooling backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a pooling forward propagation primitive. It is used as a hint for deciding which memory format to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **`diff_src_desc`**`()` **`const`**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **`diff_dst_desc`**`()` **`const`**

Returns a diff destination memory descriptor.

Return Diff destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination parameter.

memory::*desc* **`workspace_desc`**`()` **`const`**

Returns the workspace memory descriptor.

Return Workspace memory descriptor.

Return A zero memory descriptor if the primitive does not require workspace parameter.

5.5.14 Reorder

A primitive to copy data between two memory objects. This primitive is typically used to change the way that the data is laid out in memory.

The reorder primitive copies data between different memory formats but does not change the tensor from mathematical perspective. Variable names follow the standard *Conventions*.

$$\mathrm{dst}(\overline{x}) = \mathrm{src}(\overline{x})$$

for $\overline{x} = (x_0, \ldots, x_n)$.

As described in *Introduction* in order to achieve the best performance some primitives (such as convolution) require special memory format which is typically referred to as an *optimized* memory format. The *optimized* memory format may match or may not match memory format that data is currently kept in. In this case a user can use reorder primitive to copy (reorder) the data between the memory formats.

Using the attributes and post-ops users can also use reorder primitive to quantize the data (and if necessary change the memory format simultaneously).

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_FROM*
dst	*DNNL_ARG_TO*

Operation Details

1. The reorder primitive requires the source and destination tensors to have the same shape. Implicit broadcasting is not supported.
2. While in most of the cases the reorder should be able to handle arbitrary source and destination memory formats and data types, it might happen than some combinations are not implemented. For instance:
   - Reorder implementations between weights in non-plain memory formats might be limited (but if encountered in real practice should be treated as a bug and reported to oneDNN team);
   - Weights in one Winograd format cannot be reordered to the weights of the other Winograd format;
   - Quantized weights for convolution with #dnnl_s8 source data type cannot be dequantized back to the #dnnl_f32 data type;
3. To alleviate the problem a user may rely on fact that the reorder from original plain memory format and user's data type to the *optimized* format with chosen data type should be always implemented.

Data Types Support

The reorder primitive supports arbitrary data types for the source and destination.

When converting the data from one data type to a smaller one saturation is used. For instance:

```
reorder(src={1024, data_type=f32}, dst={, data_type=s8})
// dst == {127}

reorder(src={-124, data_type=f32}, dst={, data_type=u8})
// dst == {0}
```

Data Representation

The reorder primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

Post-ops and Attributes

The reorder primitive should support the following attributes and post-ops:

Type	Operation	Description	Restrictions
Attribute	*Output scales*	Scales the result by given scale factor(s)	
Post-op	*Sum*	Adds the operation result to the destination tensor instead of overwriting it	

For instance, the following pseudo-code

```
reorder(
        src = {dims={N, C, H, W}, data_type=dt_src, memory_format=fmt_src},
        dst = {dims={N, C, H, W}, data_type=dt_dst, memory_format=fmt_dst},
        attr ={
            output_scale=alpha,
```

(continues on next page)

(continued from previous page)

```
            post-ops = { sum={scale=beta} },
        })
```

would lead to the following operation:

$$\mathrm{dst}(\overline{x}) = \alpha \cdot \mathrm{src}(\overline{x}) + \beta \cdot \mathrm{dst}(\overline{x})$$

Note: The intermediate operations are being done using single precision floating point data type.

API

struct dnnl::**reorder** : **public** dnnl::*primitive*

Reorder primitive.

Public Functions

reorder()

Default constructor. Produces an empty object.

reorder(**const** *primitive_desc* &*pd*)

Constructs a reorder primitive.

Parameters

- pd: Primitive descriptor for reorder primitive.

reorder(**const** *memory* &*src*, **const** *memory* &*dst*, **const** *primitive_attr* &*attr* = *primitive_attr*())

Constructs a reorder primitive that would reorder data between memory objects having the same memory descriptors as memory objects src and dst.

Parameters

- src: Source memory object.
- dst: Destination memory object.
- attr: Primitive attributes to use (optional).

void **execute**(**const** *stream* &*astream*, *memory* &*src*, *memory* &*dst*) **const**

Executes the reorder primitive.

Parameters

- astream: Stream object. The stream must belong to the same engine as the primitive.
- src: Source memory object.
- dst: Destination memory object.

struct primitive_desc : **public** dnnl::*primitive_desc_base*

Primitive descriptor for a reorder primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *engine* &*src_engine*, **const** *memory*::*desc* &*src_md*, **const** *engine* &*dst_engine*, **const** *memory*::*desc* &*dst_md*, **const** *primitive_attr* &*attr* = *primitive_attr*(), bool *allow_empty* = false)
Constructs a primitive descriptor for reorder primitive.

Note If `allow_empty` is true, the constructor does not throw if a primitive descriptor cannot be created.

Parameters

- `src_engine`: Engine on which the source memory object will be located.
- `src_md`: Source memory descriptor.
- `dst_engine`: Engine on which the destination memory object will be located.
- `dst_md`: Destination memory descriptor.
- `attr`: Primitive attributes to use (optional).
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *memory* &*src*, **const** *memory* &*dst*, **const** *primitive_attr* &*attr* = *primitive_attr*(), bool *allow_empty* = false)
Constructs a primitive descriptor for reorder primitive.

Parameters

- `src`: Source memory object. It is used to obtain the source memory descriptor and engine.
- `dst`: Destination memory object. It is used to obtain the destination memory descriptor and engine.
- `attr`: Primitive attributes to use (optional).
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

engine **get_src_engine**() **const**
Returns the engine on which the source memory is allocated.
Return The engine on which the source memory is allocated.

engine **get_dst_engine**() **const**
Returns the engine on which the destination memory is allocated.
Return The engine on which the destination memory is allocated.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

5.5.15 Resampling

The resampling primitive computes forward or backward resampling operation on 1D, 2D, or 3D spatial data. Resampling performs spatial scaling of original tensor using one of the supported interpolation algorithms:

- Nearest Neighbor
- Linear (or Bilinear for 2D spatial tensor, Trilinear for 3D spatial tensor).

Resampling operation is defined by the source tensor and scaling factors in each spatial dimension. Upsampling and downsampling are the alternative terms for resampling that are used when all scaling factors are greater (upsampling) or less (downsampling) than one.

The resampling operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard *Conventions*.

Let src and dst be $N \times C \times IH \times IW$ and $N \times C \times OH \times OW$ tensors respectively. Let $F_h = \frac{OH}{IH}$ and $F_w = \frac{OW}{IW}$ define scaling factors in each spatial dimension.

The following formulas show how oneDNN computes resampling for nearest neighbor and bilinear interpolation methods. To further simplify the formulas, we assume the following:

- $\mathrm{src}(n, ic, ih, iw) = 0$ if $ih < 0$ or $iw < 0$,
- $\mathrm{src}(n, ic, ih, iw) = \mathrm{src}(n, ic, IH - 1, iw)$ if $ih \geq IH$,
- $\mathrm{src}(n, ic, ih, iw) = \mathrm{src}(n, ic, ih, IW - 1)$ if $iw \geq IW$.

Forward

Nearest Neighbor Resampling

$$\mathrm{dst}(n, c, oh, ow) = \mathrm{src}(n, c, ih, iw)$$

where

- $ih = [\frac{oh+0.5}{F_h} - 0.5]$,
- $iw = [\frac{ow+0.5}{F_w} - 0.5]$.

Bilinear Resampling

$$\begin{aligned}\mathrm{dst}(n, c, oh, ow) = \mathrm{src}(n, c, ih_0, iw_0) \cdot W_{ih} \cdot W_{iw} + \\ \mathrm{src}(n, c, ih_1, iw_0) \cdot (1 - W_{ih}) \cdot W_{iw} + \\ \mathrm{src}(n, c, ih_0, iw_1) \cdot W_{ih} \cdot (1 - W_{iw}) + \\ \mathrm{src}(n, c, ih_1, iw_1) \cdot (1 - W_{ih}) \cdot (1 - W_{iw})\end{aligned}$$

where

- $ih_0 = \left\lfloor \frac{oh+0.5}{F_h} - 0.5 \right\rfloor$,
- $ih_1 = \left\lceil \frac{oh+0.5}{F_h} - 0.5 \right\rceil$,

- $iw_0 = \left\lfloor \frac{ow+0.5}{F_w} - 0.5 \right\rfloor$,
- $iw_1 = \left\lceil \frac{ow+0.5}{F_w} - 0.5 \right\rceil$,
- $W_{ih} = \frac{oh+0.5}{F_h} - 0.5 - ih_0$,
- $W_{iw} = \frac{ow+0.5}{F_w} - 0.5 - iw_0$.

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes diff_src based on diff_dst.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. Resampling implementation supports data with arbitrary data tag (*nchw*, *nhwc*, etc.) but memory tags for `src` and `dst` are expected to be the same. Resampling primitive supports `dst` and `diff_src` memory tag *any* and can define destination format based on source format.
2. Resampling descriptor can be created by specifying the source and destination memory descriptors, only the source descriptor and floating point factors, or the source and destination memory descriptors and factors. In case when user does not provide the destination descriptor, the destination dimensions are deduced using the factors: $output\_spatial\_size = \left\lfloor \frac{input\_spatial\_size}{F} \right\rfloor$.

Note: Resampling algorithm uses factors as defined by the relation $F = \frac{output\_spatial\_size}{input\_spatial\_size}$ that do not necessarily equal to the ones passed by the user.

Data Types Support

Resampling primitive supports the following combination of data types for source and destination memory objects.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*f32*, *bf16*
forward	*f16*, *s8*, *u8*

Post-ops and Attributes

The resampling primitive does not support any post-ops or attributes.

API

struct dnnl::**resampling_forward** : **public** dnnl::*primitive*

Resampling forward propagation.

Public Functions

resampling_forward()

Default constructor. Produces an empty object.

resampling_forward(**const** *primitive_desc* &*pd*)

Constructs a resampling forward propagation primitive.

Parameters

- pd: Primitive descriptor for a resampling forward propagation primitive.

struct desc

Descriptor for resampling forward propagation.

Public Functions

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*dst_desc*)

Constructs a descriptor for a resampling forward propagation primitive using source and destination memory descriptors.

Note The destination memory descriptor may be initialized with *dnnl::memory::format_tag::any* value of format_tag.

Parameters

- aprop_kind: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- aalgorithm: resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- src_desc: Source memory descriptor.
- dst_desc: Destination memory descriptor.

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** std::vector<float> &*factors*, **const** *memory*::*desc* &*src_desc*)
Constructs a descriptor for a resampling forward propagation primitive using source memory descriptor and factors.

Parameters
- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- `factors`: Vector of scaling factors for spatial dimension.
- `src_desc`: Source memory descriptor.

desc(*prop_kind aprop_kind*, *algorithm aalgorithm*, **const** std::vector<float> &*factors*, **const** *memory*::*desc* &*src_desc*, **const** *memory*::*desc* &*dst_desc*)
Constructs a descriptor for a resampling forward propagation primitive.

Note The destination memory descriptor may be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `aalgorithm`: resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- `factors`: Vector of scaling factors for spatial dimension.
- `src_desc`: Source memory descriptor.
- `dst_desc`: Destination memory descriptor.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for a resampling forward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a resampling forward propagation primitive.

Parameters
- `adesc`: Descriptor for a resampling forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a resampling forward propagation primitive.

Parameters
- `adesc`: Descriptor for a resampling forward propagation primitive.
- `aengine`: Engine to use.
- `attr`: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

struct dnnl::**resampling_backward** : **public** dnnl::*primitive*
Resampling backward propagation primitive.

Public Functions

resampling_backward()
Default constructor. Produces an empty object.

resampling_backward(**const** *primitive_desc* &*pd*)
Constructs a resampling backward propagation primitive.

Parameters

- pd: Primitive descriptor for a resampling backward propagation primitive.

struct desc
Descriptor for a resampling backward propagation primitive.

Public Functions

desc(*algorithm aalgorithm*, **const** *memory*::*desc* &*diff_src_desc*, **const** *memory*::*desc* &*diff_dst_desc*)
Constructs a descriptor for a resampling backward propagation primitive using source and destination memory descriptors.

Parameters

- aalgorithm: resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- diff_src_desc: Diff source memory descriptor.
- diff_dst_desc: Diff destination memory descriptor.

desc(*algorithm aalgorithm*, **const** std::vector<float> &*factors*, **const** *memory*::*desc* &*diff_src_desc*, **const** *memory*::*desc* &*diff_dst_desc*)
Constructs a descriptor for resampling backward propagation primitive.

Parameters

- aalgorithm: resampling algorithm kind: either *dnnl::algorithm::resampling_nearest*, or *dnnl::algorithm::resampling_linear*
- factors: Vector of scaling factors for spatial dimension.
- diff_src_desc: Diff source memory descriptor.
- diff_dst_desc: Diff destination memory descriptor.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for resampling backward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *resampling_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for a resampling backward propagation primitive.

Parameters
- `adesc`: Descriptor for a resampling backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a resampling forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *resampling_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for a resampling backward propagation primitive.

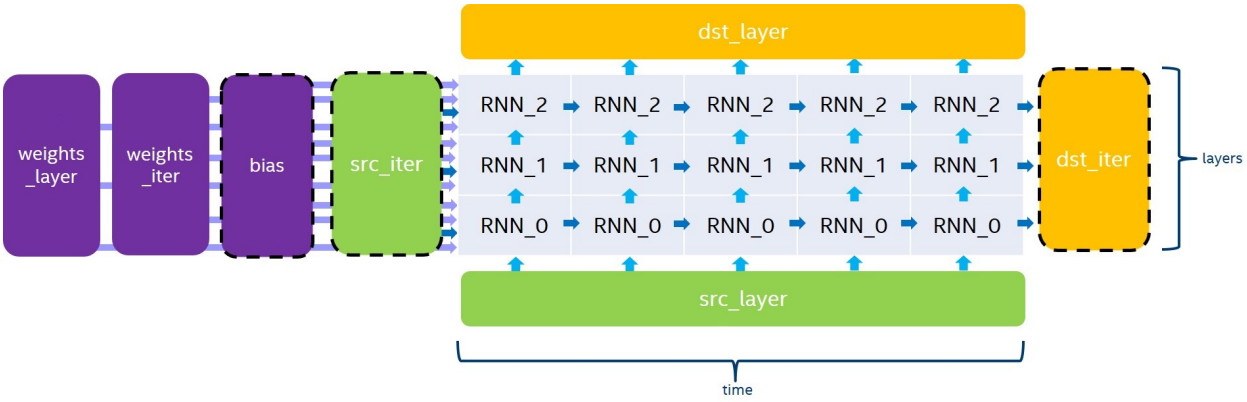
Parameters
- `adesc`: Descriptor for a resampling backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a resampling forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() **const**
Returns a diff source memory descriptor.
Return Diff source memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc**() **const**
Returns a diff destination memory descriptor.
Return Diff destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination parameter.

5.5.16 RNN

The RNN primitive computes a stack of unrolled recurrent cells, as depicted in Figure 1. bias, src_iter and dst_iter are optional parameters. If not provided, bias and src_iter default to 0. Variable names follow the standard *Conventions*.



The RNN primitive supports four modes for evaluation direction:

- `left2right` will process the input data timestamps by increasing order,
- `right2left` will process the input data timestamps by decreasing order,
- `bidirectional_concat` will process all the stacked layers from `left2right` and from `right2left` independently, and will concatenate the output in dst_layer over the channel dimension,
- `bidirectional_sum` will process all the stacked layers from `left2right` and from `right2left` independently, and will sum the two outputs to dst_layer.

Even though the RNN primitive supports passing a different number of channels for src_layer, src_iter, dst_layer, and dst_iter, we always require the following conditions in order for the dimension to be consistent:

- $channels(\text{dst\_layer}) = channels(\text{dst\_iter})$,
- when $T > 1$, $channels(\text{src\_iter}) = channels(\text{dst\_iter})$,
- when $L > 1$, $channels(\text{src\_layer}) = channels(\text{dst\_layer})$,
- when using the `bidirectional_concat` direction, $channels(\text{dst\_layer}) = 2 * channels(\text{dst\_iter})$.

The general formula for the execution of a stack of unrolled recurrent cells depends on the current iteration of the previous layer ($h_{t,l-1}$ and $c_{t,l-1}$) and the previous iteration of the current layer ($h_{t-1,l}$). Here is the exact equation for non-LSTM cells:

$$h_{t,l} = Cell(h_{t,l-1}, h_{t-1,l})$$

where t, l are the indices of the timestamp and the layer of the cell being executed.

And here is the equation for LSTM cells:

$$(h_{t,l}, c_{t,l}) = Cell(h_{t,l-1}, h_{t-1,l}, c_{t-1,l})$$

where t, l are the indices of the timestamp and the layer of the cell being executed.

Cell Functions

The RNN API provides four cell functions:

- *Vanilla RNN*, a single-gate recurrent cell,
- *LSTM*, a four-gate long short-term memory cell,
- *GRU*, a three-gate gated recurrent unit cell,
- *Linear-before-reset GRU*, a three-gate recurrent unit cell with the linear layer before the reset gate.

Vanilla RNN

A single-gate recurrent cell initialized with *dnnl::vanilla_rnn_forward::desc* or *dnnl::vanilla_rnn_forward::desc* as in the following example.

```
auto vanilla_rnn_desc = dnnl::vanilla_rnn_forward::desc(
    aprop, activation, direction, src_layer_desc, src_iter_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc);
```

The Vanilla RNN cell should support the ReLU, Tanh and Sigmoid activation functions. The following equations defines the mathematical operation performed by the Vanilla RNN cell for the forward pass:

$$a_t = W \cdot h_{t,l-1} + U \cdot h_{t-1,l} + B$$
$$h_t = activation(a_t)$$

LSTM

LSTM (or Vanilla LSTM)

A four-gate long short-term memory recurrent cell initialized with *dnnl::lstm_forward::desc* or *dnnl::lstm_backward::desc* as in the following example.

```
auto lstm_desc = dnnl::lstm_forward::desc(
    aprop, direction, src_layer_desc, src_iter_h_desc, src_iter_c_desc,
    weights_layer_desc, weights_iter_desc, bias_desc, dst_layer_desc,
    dst_iter_h_desc, dst_iter_c_desc);
```

Note that for all tensors with a dimension depending on the gates number, we implicitly require the order of these gates to be i, f, $\tilde{c}$, and o. The following equation gives the mathematical description of these gates and output for the forward pass:

$$i_t = \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + B_i)$$
$$f_t = \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + B_f)$$

$$\tilde{c}_t = tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}})$$
$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

$$o_t = \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + B_o)$$
$$h_t = tanh(c_t) * o_t$$

where W_* are stored in weights_layer, U_* are stored in weights_iter and B_* are stored in bias.

Note: In order for the dimensions to be consistent, we require $channels(\text{src\_iter\_c}) = channels(\text{dst\_iter\_c}) = channels(\text{dst\_iter})$.

LSTM with Peephole

A four-gate long short-term memory recurrent cell with peephole initialized with *dnnl::lstm_forward::desc* or *dnnl::lstm_backward::desc* as in the following example.

```
auto lstm_desc = dnnl::lstm_forward::desc(
    aprop, direction, src_layer_desc, src_iter_h_desc, src_iter_c_desc,
    weights_layer_desc, weights_iter_desc, weights_peephole_desc,
    bias_desc, dst_layer_desc, dst_iter_h_desc, dst_iter_c_desc);
```

Similarly to vanilla LSTM, we implicitly require the order of these gates to be i, f, $\tilde{c}$, and o. For peephole weights, the gates order is:math:i, f, o. The following equation gives the mathematical description of these gates and output for the forward pass:

$$
\begin{aligned}
i_t &= \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + P_i \cdot c_{t-1} + B_i) \\
f_t &= \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + P_f \cdot c_{t-1} + B_f)
\end{aligned}
$$

$$
\begin{aligned}
\tilde{c}_t &= tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}}) \\
c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t
\end{aligned}
$$

$$
\begin{aligned}
o_t &= \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + P_o \cdot c_t + B_o) \\
h_t &= tanh(c_t) * o_t
\end{aligned}
$$

where P_* are stored in `weights_peephole`, and the other parameters are the same as in vanilla LSTM.

Note: If the `weights_peephole_desc` passed to the operation descriptor constructor is a zero memory descriptor, the primitive will behave the same as in LSTM primitive without peephole.

LSTM with Projection

A four-gate long short-term memory recurrent cell with projection initialized with *dnnl::lstm_forward::desc* or *dnnl::lstm_backward::desc* as in the following example.

```
auto lstm_desc = dnnl::lstm_forward::desc(
    aprop, direction, src_layer_desc, src_iter_h_desc, src_iter_c_desc,
    weights_layer_desc, weights_iter_desc, weights_peephole_desc,
    weights_projection_desc, bias_desc, dst_layer_desc, dst_iter_h_desc,
    dst_iter_c_desc);
```

Similarly to vanilla LSTM, we implicitly require the order of the gates to be i, f, $\tilde{c}$, and o for all tensors with a dimension depending on the gates. The following equation gives the mathematical description of these gates and

output for the forward pass (for simplicity, LSTM without peephole is shown):

$$
\begin{aligned}
i_t &= \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + B_i) \\
f_t &= \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + B_f) \\
\\
\tilde{c}_t &= \tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}}) \\
c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t \\
\\
o_t &= \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + B_o) \\
h_t &= R \cdot (\tanh(c_t) * o_t)
\end{aligned}
$$

where R is stored in `weights_projection`, and the other parameters are the same as in vanilla LSTM.

Note: If the `weights_projection_desc` passed to the operation descriptor constructor is a zero memory descriptor, the primitive will behave the same as in LSTM primitive without projection.

GRU

A three-gate gated recurrent unit cell, initialized with *dnnl::gru_forward::desc* or *dnnl::gru_backward::desc* as in the following example.

```
auto gru_desc = dnnl::gru_forward::desc(
    aprop, direction, src_layer_desc, src_iter_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc);
```

Note that for all tensors with a dimension depending on the gates number, we implicitly require the order of these gates to be:math:u, r, and o. The following equation gives the mathematical definition of these gates.

$$
\begin{aligned}
u_t &= \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u) \\
r_t &= \sigma(W_r \cdot h_{t,l-1} + U_r \cdot h_{t-1,l} + B_r) \\
o_t &= tanh(W_o \cdot h_{t,l-1} + U_o \cdot (r_t * h_{t-1,l}) + B_o) \\
h_t &= u_t * h_{t-1,l} + (1 - u_t) * o_t
\end{aligned}
$$

where W_* are in weights_layer, U_* are in weights_iter, and B_* are stored in bias.

Note: If you need to replace u_t by $(1 - u_t)$ when computing h_t, you can achieve this by multiplying W_u, U_u and B_u by -1. This is possible as $u_t = \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u)$, and $1^{\smile}\sigma(a) = \sigma(-a)$.

Linear-Before-Reset GRU

A three-gate gated recurrent unit cell with linear layer applied before the reset gate, initialized with *dnnl::lbr_gru_forward::desc* or *dnnl::lbr_gru_backward::desc* as in the following example.

```
auto lbr_gru_desc = dnnl::lbr_gru_forward::desc(
    aprop, direction, src_layer_desc, src_iter_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc);
```

The following equation describes the mathematical behavior of the Linear-Before-Reset GRU cell.

$$
\begin{aligned}
u_t &= \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u) \\
r_t &= \sigma(W_r \cdot h_{t,l-1} + U_r \cdot h_{t-1,l} + B_r) \\
o_t &= tanh(W_o \cdot h_{t,l-1} + r_t * (U_o \cdot h_{t-1,l} + B_{u'}) + B_o) \\
h_t &= u_t * h_{t-1,l} + (1 - u_t) * o_t
\end{aligned}
$$

Note that for all tensors with a dimension depending on the gates number, except the bias, we implicitly require the order of these gates to be u, r, and o. For the bias tensor, we implicitly require the order of the gates to be u, r, o, and u'.

Note: If you need to replace u_t by $(1 - u_t)$ when computing h_t, you can achieve this by multiplying W_u, U_u and B_u by -1. This is possible as $u_t = \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u)$, and $1^{\smile}\sigma(a) = \sigma(-a)$.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src_layer	*DNNL_ARG_SRC_LAYER*
src_iter	*DNNL_ARG_SRC_ITER*
src_iter_c	*DNNL_ARG_SRC_ITER_C*
weights_layer	*DNNL_ARG_WEIGHTS_LAYER*
weights_iter	*DNNL_ARG_WEIGHTS_ITER*
weights_peephole	DNNL_ARG_WEIGHTS_PEEPHOLE
weights_projection	DNNL_ARG_WEIGHTS_PROJECTION
bias	*DNNL_ARG_BIAS*
dst_layer	*DNNL_ARG_DST_LAYER*
dst_iter	*DNNL_ARG_DST_ITER*
dst_iter_c	*DNNL_ARG_DST_ITER_C*
workspace	*DNNL_ARG_WORKSPACE*
diff_src_layer	*DNNL_ARG_DIFF_SRC_LAYER*
diff_src_iter	*DNNL_ARG_DIFF_SRC_ITER*
diff_src_iter_c	*DNNL_ARG_DIFF_SRC_ITER_C*
diff_weights_layer	*DNNL_ARG_DIFF_WEIGHTS_LAYER*
diff_weights_iter	*DNNL_ARG_DIFF_WEIGHTS_ITER*
diff_weights_peephole	DNNL_ARG_DIFF_WEIGHTS_PEEPHOLE
diff_weights_projection	DNNL_ARG_DIFF_WEIGHTS_PROJECTION
diff_bias	*DNNL_ARG_DIFF_BIAS*
diff_dst_layer	*DNNL_ARG_DIFF_DST_LAYER*
diff_dst_iter	*DNNL_ARG_DIFF_DST_ITER*
diff_dst_iter_c	*DNNL_ARG_DIFF_DST_ITER_C*

Operation Details

N/A

Data Types Support

The following table lists the combination of data types that should be supported by the RNN primitive for each input and output memory object.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Cell Function	Input Data	Recurrent Data (1)	Weights	Bias	Output Data
Forward / Backward	All	*f32*	*f32*	*f32*	*f32*	*f32*
Forward / Backward (2)	All (3)	*bf16*	*bf16*	*bf16*	*f32*	*bf16*
Forward	All (3)	*f16*	*f16*	*f16*	*f16*	*f16*
Forward inference	Vanilla LSTM	*u8*	*u8*	*s8*	*f32*	*u8*, *f32*

(1) With LSTM and Peephole LSTM cells, the cell state data type is always f32.

(2) In backward propagation, all `diff_*` tensors are in f32.

(3) Projection LSTM is not defined yet.

Data Representation

In the oneDNN programming model, the RNN primitive is one of a few that support the placeholder memory format #dnnl::memory::format_tag::any (shortened to `any` from now on) and can define data and weight memory objects format based on the primitive parameters.

The following table summarizes the data layouts supported by the RNN primitive.

Input/Output Data	Recurrent Data	Layer and Iteration Weights	Peephole Weights and Bias	Projection LSTM Weights
any	*any*	*any*	*ldgo*	*any*, *ldio* (Forward propagation)
ntc, *tnc*	*ldnc*	*ldigo*, *ldgoi*	*ldgo*	*any*, *ldio* (Forward propagation)

While an RNN primitive can be created with memory formats specified explicitly, the performance is likely to be sub-optimal. When using *any* it is necessary to first create an RNN primitive descriptor and then query it for the actual data and weight memory objects formats.

Note: The RNN primitive should support padded tensors and views. So even if two memory descriptors share the same data layout, they might still be different.

Post-ops and Attributes

Currently post-ops and attributes are only used by the int8 variant of LSTM.

API

enum dnnl::**rnn_flags**
RNN cell flags.

Values:

enumerator undef
Undefined RNN flags.

enum dnnl::**rnn_direction**
A direction of RNN primitive execution.

Values:

enumerator unidirectional_left2right
Unidirectional execution of RNN primitive from left to right.

enumerator unidirectional_right2left
Unidirectional execution of RNN primitive from right to left.

enumerator bidirectional_concat
Bidirectional execution of RNN primitive with concatenation of the results.

enumerator bidirectional_sum
Bidirectional execution of RNN primitive with summation of the results.

enumerator unidirectional
Alias for *dnnl::rnn_direction::unidirectional_left2right*.

struct dnnl::**vanilla_rnn_forward** : **public** dnnl::*primitive*
Vanilla RNN forward propagation primitive.

Public Functions

vanilla_rnn_forward()
Default constructor. Produces an empty object.

vanilla_rnn_forward(**const** *primitive_desc* &*pd*)
Constructs a vanilla RNN forward propagation primitive.

Parameters

- pd: Primitive descriptor for a vanilla RNN forward propagation primitive.

struct desc
Descriptor for a vanilla RNN forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, *algorithm activation*, *rnn_direction direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, *rnn_flags flags* = *rnn_flags*::*undef*, float *alpha* = 0.0f, float *beta* = 0.0f)

Constructs a descriptor for a vanilla RNN forward propagation primitive.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc`,
- `bias_desc`,
- `dst_iter_desc`.

This would then indicate that the RNN forward propagation primitive should not use them and should default to zero values instead.

Note All memory descriptors except `src_iter_desc` can be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `activation`: Activation kind. Possible values are *dnnl::algorithm::eltwise_relu*, *dnnl::algorithm::eltwise_tanh*, or *dnnl::algorithm::eltwise_logistic*.
- `direction`: RNN direction. See *dnnl::rnn_direction* for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `flags`: Unused.
- `alpha`: Negative slope if activation is *dnnl::algorithm::eltwise_relu*.
- `beta`: Unused.

struct primitive_desc : **public** dnnl::*rnn_primitive_desc_base*

Primitive descriptor for a vanilla RNN forward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a vanilla RNN forward propagation primitive.

Parameters

- `adesc`: Descriptor for a vanilla RNN forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a vanilla RNN forward propagation primitive.

Parameters

- `adesc`: Descriptor for a vanilla RNN forward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() **const**
Returns source layer memory descriptor.
Return Source layer memory descriptor.

memory::*desc* **src_iter_desc**() **const**
Returns source iteration memory descriptor.
Return Source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() **const**
Returns weights layer memory descriptor.
Return Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() **const**
Returns weights iteration memory descriptor.
Return Weights iteration memory descriptor.

memory::*desc* **bias_desc**() **const**
Returns bias memory descriptor.
Return Bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() **const**
Returns destination layer memory descriptor.
Return Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() **const**
Returns destination iteration memory descriptor.
Return Destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

struct dnnl::**vanilla_rnn_backward** : **public** dnnl::*primitive*
Vanilla RNN backward propagation primitive.

Public Functions

vanilla_rnn_backward()
Default constructor. Produces an empty object.

vanilla_rnn_backward(**const** *primitive_desc* &*pd*)
Constructs a vanilla RNN backward propagation primitive.

Parameters

- `pd`: Primitive descriptor for a vanilla RNN backward propagation primitive.

struct desc

Descriptor for a vanilla RNN backward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, *algorithm activation*, *rnn_direction direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, **const** *memory*::*desc* &*diff_src_layer_desc*, **const** *memory*::*desc* &*diff_src_iter_desc*, **const** *memory*::*desc* &*diff_weights_layer_desc*, **const** *memory*::*desc* &*diff_weights_iter_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_layer_desc*, **const** *memory*::*desc* &*diff_dst_iter_desc*, *rnn_flags flags* = *rnn_flags*::*undef*, float *alpha* = 0.0f, float *beta* = 0.0f)

Constructs a descriptor for a vanilla RNN backward propagation primitive.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `diff_src_iter_desc`,
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `diff_dst_iter_desc`.

This would then indicate that the RNN backward propagation primitive should not use the respective data and should use zero values instead.

Note All the memory descriptors may be initialized with the *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Must be *dnnl::prop_kind::backward*.
- `activation`: Activation kind. Possible values are *dnnl::algorithm::eltwise_relu*, *dnnl::algorithm::eltwise_tanh*, or *dnnl::algorithm::eltwise_logistic*.
- `direction`: RNN direction. See *dnnl::rnn_direction* for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `diff_src_layer_desc`: Memory descriptor for the diff of input vector.
- `diff_src_iter_desc`: Memory descriptor for the diff of input recurrent hidden state vector.
- `diff_weights_layer_desc`: Memory descriptor for the diff of weights applied to the layer input.
- `diff_weights_iter_desc`: Memory descriptor for the diff of weights applied to the recurrent input.
- `diff_bias_desc`: Diff bias memory descriptor.
- `diff_dst_layer_desc`: Memory descriptor for the diff of output vector.
- `diff_dst_iter_desc`: Memory descriptor for the diff of output recurrent hidden state vector.
- `flags`: Unused.
- `alpha`: Negative slope if activation is *dnnl::algorithm::eltwise_relu*.
- `beta`: Unused.

struct primitive_desc : **public** dnnl::*rnn_primitive_desc_base*

Primitive descriptor for an RNN backward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *vanilla_rnn_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for a vanilla RNN backward propagation primitive.

Parameters
- `adesc`: Descriptor for a vanilla RNN backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a vanilla RNN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *vanilla_rnn_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for a vanilla RNN backward propagation primitive.

Parameters
- `adesc`: Descriptor for a vanilla RNN backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a vanilla RNN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() **const**
Returns source layer memory descriptor.
Return Source layer memory descriptor.

memory::*desc* **src_iter_desc**() **const**
Returns source iteration memory descriptor.
Return Source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() **const**
Returns weights layer memory descriptor.
Return Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() **const**
Returns weights iteration memory descriptor.
Return Weights iteration memory descriptor.

memory::*desc* **bias_desc**() **const**
Returns bias memory descriptor.
Return Bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() **const**
Returns destination layer memory descriptor.
Return Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() **const**
Returns destination iteration memory descriptor.
Return Destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **diff_src_layer_desc**() **const**
Returns diff source layer memory descriptor.
Return Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() **const**
Returns diff source iteration memory descriptor.
Return Diff source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_weights_layer_desc**() **const**
Returns diff weights layer memory descriptor.
Return Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() **const**
Returns diff weights iteration memory descriptor.
Return Diff weights iteration memory descriptor.

memory::*desc* **diff_bias_desc**() **const**
Returns diff bias memory descriptor.
Return Diff bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() **const**
Returns diff destination layer memory descriptor.
Return Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() **const**
Returns diff destination iteration memory descriptor.
Return Diff destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

struct dnnl::**lstm_forward** : **public** dnnl::*primitive*
LSTM forward propagation primitive.

Public Functions

lstm_forward()
Default constructor. Produces an empty object.

lstm_forward(**const** *primitive_desc* &*pd*)
Constructs an LSTM forward propagation primitive.

Parameters

- pd: Primitive descriptor for an LSTM forward propagation primitive.

struct desc
Descriptor for an LSTM forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, *rnn_direction direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*src_iter_c_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*weights_peephole_desc*, **const** *memory*::*desc* &*weights_projection_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, **const** *memory*::*desc* &*dst_iter_c_desc*, *rnn_flags flags* = *rnn_flags*::*undef*)

Constructs a descriptor for an LSTM (with or without peephole and with or without projection) forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc` together with `src_iter_c_desc`,
- `weights_peephole_desc`,
- `bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`.

This would then indicate that the LSTM forward propagation primitive should not use them and should default to zero values instead.

The `weights_projection_desc` may point to a zero memory descriptor. This would then indicate that the LSTM doesn't have recurrent projection layer.

Note All memory descriptors can be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters
- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `direction`: RNN direction. See *dnnl::rnn_direction* for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `src_iter_c_desc`: Memory descriptor for the input recurrent cell state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `weights_peephole_desc`: Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- `weights_projection_desc`: Memory descriptor for the weights applied to the hidden states to get the recurrent projection (according to the Projection LSTM formula).
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `dst_iter_c_desc`: Memory descriptor for the output recurrent cell state vector.
- `flags`: Unused.

desc(*prop_kind aprop_kind*, *rnn_direction direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*src_iter_c_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*weights_peephole_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, **const** *memory*::*desc* &*dst_iter_c_desc*, *rnn_flags flags* = *rnn_flags*::*undef*)

Constructs a descriptor for an LSTM (with or without peephole) forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- `src_iter_desc` together with `src_iter_c_desc`,
- `weights_peephole_desc`,
- `bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`.

This would then indicate that the LSTM forward propagation primitive should not use them and should default to zero values instead.

Note All memory descriptors can be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `direction`: RNN direction. See *dnnl::rnn_direction* for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `src_iter_c_desc`: Memory descriptor for the input recurrent cell state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `weights_peephole_desc`: Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `dst_iter_c_desc`: Memory descriptor for the output recurrent cell state vector.
- `flags`: Unused.

desc(*prop_kind aprop_kind*, *rnn_direction direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*src_iter_c_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, **const** *memory*::*desc* &*dst_iter_c_desc*, *rnn_flags flags* = *rnn_flags*::*undef*)

Constructs a descriptor for an LSTM forward propagation primitive.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `src_iter_c_desc`,
- `bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`.

This would then indicate that the LSTM forward propagation primitive should not use them and should default to zero values instead.

Note All memory descriptors can be initialized with an *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `direction`: RNN direction. See *dnnl::rnn_direction* for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `src_iter_c_desc`: Memory descriptor for the input recurrent cell state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `dst_iter_c_desc`: Memory descriptor for the output recurrent cell state vector.
- `flags`: Unused.

struct primitive_desc : **public** dnnl::*rnn_primitive_desc_base*

Primitive descriptor for an LSTM forward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for an LSTM forward propagation primitive.

Parameters
- `adesc`: Descriptor for an LSTM forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for an LSTM forward propagation primitive.

Parameters
- `adesc`: Descriptor for an LSTM forward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() **const**
Returns source layer memory descriptor.
Return Source layer memory descriptor.

memory::*desc* **src_iter_desc**() **const**
Returns source iteration memory descriptor.
Return Source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **src_iter_c_desc**() **const**
Returns source iteration memory descriptor.
Return Source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() **const**
Returns weights layer memory descriptor.
Return Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() **const**
Returns weights iteration memory descriptor.
Return Weights iteration memory descriptor.

memory::*desc* **weights_peephole_desc**() **const**
Returns weights peephole memory descriptor.
Return Weights peephole memory descriptor.

memory::*desc* **weights_projection_desc**() **const**
Returns weights projection memory descriptor.
Return Weights projection memory descriptor.

memory::*desc* **bias_desc**() **const**
Returns bias memory descriptor.
Return Bias memory descriptor.

Return A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() **const**
Returns destination layer memory descriptor.
Return Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() **const**
Returns destination iteration memory descriptor.
Return Destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **dst_iter_c_desc**() **const**
Returns source iteration memory descriptor.
Return Source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

struct dnnl::**lstm_backward** : **public** dnnl::*primitive*
LSTM backward propagation primitive.

Public Functions

lstm_backward()
Default constructor. Produces an empty object.

lstm_backward(**const** *primitive_desc* &*pd*)
Constructs an LSTM backward propagation primitive.

Parameters

- pd: Primitive descriptor for an LSTM backward propagation primitive.

struct desc
Descriptor for an LSTM backward propagation primitive.

Public Functions

desc(*prop_kind* *aprop_kind*, *rnn_direction* *direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*src_iter_c_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*weights_peephole_desc*, **const** *memory*::*desc* &*weights_projection_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, **const** *memory*::*desc* &*dst_iter_c_desc*, **const** *memory*::*desc* &*diff_src_layer_desc*, **const** *memory*::*desc* &*diff_src_iter_desc*, **const** *memory*::*desc* &*diff_src_iter_c_desc*, **const** *memory*::*desc* &*diff_weights_layer_desc*, **const** *memory*::*desc* &*diff_weights_iter_desc*, **const** *memory*::*desc* &*diff_weights_peephole_desc*, **const** *memory*::*desc* &*diff_weights_projection_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_layer_desc*, **const** *memory*::*desc* &*diff_dst_iter_desc*, **const** *memory*::*desc* &*diff_dst_iter_c_desc*, *rnn_flags* *flags* = *rnn_flags*::*undef*)
projection) descriptor for backward propagation using prop_kind, direction, and memory descriptors.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `src_iter_c_desc`, `diff_src_iter_desc`, and `diff_src_iter_c_desc`,
- `weights_peephole_desc` together with `diff_weights_peephole_desc`
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`, `diff_dst_iter_desc`, and `diff_dst_iter_c_desc`.

This would then indicate that the LSTM backward propagation primitive should not use them and should default to zero values instead.

The `weights_projection_desc` together with `diff_weights_projection_desc` may point to a zero memory descriptor. This would then indicate that the LSTM doesn't have recurrent projection layer.

Note All memory descriptors can be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Must be *dnnl::prop_kind::backward*.
- `direction`: RNN direction. See *dnnl::rnn_direction* for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `src_iter_c_desc`: Memory descriptor for the input recurrent cell state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `weights_peephole_desc`: Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- `weights_projection_desc`: Memory descriptor for the weights applied to the hidden states to get the recurrent projection (according to the Projection LSTM formula).
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `dst_iter_c_desc`: Memory descriptor for the output recurrent cell state vector.
- `diff_src_layer_desc`: Memory descriptor for the diff of input vector.
- `diff_src_iter_desc`: Memory descriptor for the diff of input recurrent hidden state vector.
- `diff_src_iter_c_desc`: Memory descriptor for the diff of input recurrent cell state vector.
- `diff_weights_layer_desc`: Memory descriptor for the diff of weights applied to the layer input.
- `diff_weights_iter_desc`: Memory descriptor for the diff of weights applied to the recurrent input.
- `diff_weights_peephole_desc`: Memory descriptor for the diff of weights applied to the cell states (according to the Peephole LSTM formula).
- `diff_weights_projection_desc`: Memory descriptor for the diff of weights applied to the hidden states to get the recurrent projection (according to the Projection LSTM formula).
- `diff_bias_desc`: Diff bias memory descriptor.
- `diff_dst_layer_desc`: Memory descriptor for the diff of output vector.
- `diff_dst_iter_desc`: Memory descriptor for the diff of output recurrent hidden state vector.
- `diff_dst_iter_c_desc`: Memory descriptor for the diff of output recurrent cell state vector.
- `flags`: Unused.

desc(*prop_kind aprop_kind*, *rnn_direction direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*src_iter_c_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*weights_peephole_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, **const** *memory*::*desc* &*dst_iter_c_desc*, **const** *memory*::*desc* &*diff_src_layer_desc*, **const** *memory*::*desc* &*diff_src_iter_desc*, **const** *memory*::*desc* &*diff_src_iter_c_desc*, **const** *memory*::*desc* &*diff_weights_layer_desc*, **const** *memory*::*desc* &*diff_weights_iter_desc*, **const** *memory*::*desc* &*diff_weights_peephole_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_layer_desc*, **const** *memory*::*desc* &*diff_dst_iter_desc*, **const** *memory*::*desc* &*diff_dst_iter_c_desc*, *rnn_flags flags* = *rnn_flags*::*undef*)

Constructs an LSTM (with or without peephole) descriptor for backward propagation using `prop_kind`, `direction`, and memory descriptors.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `src_iter_c_desc`, `diff_src_iter_desc`, and `diff_src_iter_c_desc`,
- `weights_peephole_desc` together with `diff_weights_peephole_desc`
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`, `diff_dst_iter_desc`, and `diff_dst_iter_c_desc`.

This would then indicate that the LSTM backward propagation primitive should not use them and should default to zero values instead.

Note All memory descriptors may be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Must be *dnnl::prop_kind::backward*.
- `direction`: RNN direction. See *dnnl::rnn_direction* for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `src_iter_c_desc`: Memory descriptor for the input recurrent cell state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `weights_peephole_desc`: Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `dst_iter_c_desc`: Memory descriptor for the output recurrent cell state vector.
- `diff_src_layer_desc`: Memory descriptor for the diff of input vector.
- `diff_src_iter_desc`: Memory descriptor for the diff of input recurrent hidden state vector.
- `diff_src_iter_c_desc`: Memory descriptor for the diff of input recurrent cell state vector.
- `diff_weights_layer_desc`: Memory descriptor for the diff of weights applied to the layer input.
- `diff_weights_iter_desc`: Memory descriptor for the diff of weights applied to the recurrent input.
- `diff_weights_peephole_desc`: Memory descriptor for the diff of weights applied to the cell states (according to the Peephole LSTM formula).
- `diff_bias_desc`: Diff bias memory descriptor.
- `diff_dst_layer_desc`: Memory descriptor for the diff of output vector.
- `diff_dst_iter_desc`: Memory descriptor for the diff of output recurrent hidden state vector.

- `diff_dst_iter_c_desc`: Memory descriptor for the diff of output recurrent cell state vector.
- `flags`: Unused.

desc(*prop_kind aprop_kind*, *rnn_direction direction*, **const** *memory*::*desc* *&src_layer_desc*, **const** *memory*::*desc* *&src_iter_desc*, **const** *memory*::*desc* *&src_iter_c_desc*, **const** *memory*::*desc* *&weights_layer_desc*, **const** *memory*::*desc* *&weights_iter_desc*, **const** *memory*::*desc* *&bias_desc*, **const** *memory*::*desc* *&dst_layer_desc*, **const** *memory*::*desc* *&dst_iter_desc*, **const** *memory*::*desc* *&dst_iter_c_desc*, **const** *memory*::*desc* *&diff_src_layer_desc*, **const** *memory*::*desc* *&diff_src_iter_desc*, **const** *memory*::*desc* *&diff_src_iter_c_desc*, **const** *memory*::*desc* *&diff_weights_layer_desc*, **const** *memory*::*desc* *&diff_weights_iter_desc*, **const** *memory*::*desc* *&diff_bias_desc*, **const** *memory*::*desc* *&diff_dst_layer_desc*, **const** *memory*::*desc* *&diff_dst_iter_desc*, **const** *memory*::*desc* *&diff_dst_iter_c_desc*, *rnn_flags flags* = *rnn_flags*::*undef*)

Constructs an LSTM descriptor for backward propagation using `prop_kind`, `direction`, and memory descriptors.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `src_iter_c_desc`, `diff_src_iter_desc`, and `diff_src_iter_c_desc`,
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`, `diff_dst_iter_desc`, and `diff_dst_iter_c_desc`.

This would then indicate that the LSTM backward propagation primitive should not use them and should default to zero values instead.

Note All memory descriptors may be initialized with *dnnl::memory::format_tag::any* value of `format_tag`.

Parameters

- `aprop_kind`: Propagation kind. Must be *dnnl::prop_kind::backward*.
- `direction`: RNN direction. See *dnnl::rnn_direction* for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `src_iter_c_desc`: Memory descriptor for the input recurrent cell state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `dst_iter_c_desc`: Memory descriptor for the output recurrent cell state vector.
- `diff_src_layer_desc`: Memory descriptor for the diff of input vector.
- `diff_src_iter_desc`: Memory descriptor for the diff of input recurrent hidden state vector.
- `diff_src_iter_c_desc`: Memory descriptor for the diff of input recurrent cell state vector.
- `diff_weights_layer_desc`: Memory descriptor for the diff of weights applied to the layer input.
- `diff_weights_iter_desc`: Memory descriptor for the diff of weights applied to the recurrent input.
- `diff_bias_desc`: Diff bias memory descriptor.
- `diff_dst_layer_desc`: Memory descriptor for the diff of output vector.
- `diff_dst_iter_desc`: Memory descriptor for the diff of output recurrent hidden state vector.
- `diff_dst_iter_c_desc`: Memory descriptor for the diff of output recurrent cell state vector.
- `flags`: Unused.

struct primitive_desc : **public** dnnl::*rnn_primitive_desc_base*

Primitive descriptor for LSTM backward propagation.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *lstm_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for an LSTM backward propagation primitive.

Parameters

- `adesc`: Descriptor for LSTM backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an LSTM forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *lstm_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for an LSTM backward propagation primitive.

Parameters

- `adesc`: Descriptor for an LSTM backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an LSTM forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() **const**

Returns source layer memory descriptor.

Return Source layer memory descriptor.

memory::*desc* **src_iter_desc**() **const**

Returns source iteration memory descriptor.

Return Source iteration memory descriptor.

Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **src_iter_c_desc**() **const**

Returns source iteration memory descriptor.

Return Source iteration memory descriptor.

Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() **const**

Returns weights layer memory descriptor.

Return Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() **const**

Returns weights iteration memory descriptor.

Return Weights iteration memory descriptor.

memory::*desc* **weights_peephole_desc**() **const**
Returns weights peephole memory descriptor.
Return Weights peephole memory descriptor.

memory::*desc* **weights_projection_desc**() **const**
Returns weights projection memory descriptor.
Return Weights projection memory descriptor.

memory::*desc* **bias_desc**() **const**
Returns bias memory descriptor.
Return Bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() **const**
Returns destination layer memory descriptor.
Return Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() **const**
Returns destination iteration memory descriptor.
Return Destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **dst_iter_c_desc**() **const**
Returns source iteration memory descriptor.
Return Source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **diff_src_layer_desc**() **const**
Returns diff source layer memory descriptor.
Return Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() **const**
Returns diff source iteration memory descriptor.
Return Diff source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_src_iter_c_desc**() **const**
Returns diff source recurrent cell state memory descriptor.
Return Diff source recurrent cell state memory descriptor.

memory::*desc* **diff_weights_layer_desc**() **const**
Returns diff weights layer memory descriptor.
Return Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() **const**
Returns diff weights iteration memory descriptor.
Return Diff weights iteration memory descriptor.

memory::*desc* **diff_weights_peephole_desc**() **const**
Returns diff weights peephole memory descriptor.
Return Diff weights peephole memory descriptor.

memory::*desc* **diff_weights_projection_desc**() **const**
Returns diff weights projection memory descriptor.
Return Diff weights projection memory descriptor.

memory::*desc* **diff_bias_desc**() **const**
Returns diff bias memory descriptor.
Return Diff bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() **const**
Returns diff destination layer memory descriptor.
Return Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() **const**
Returns diff destination iteration memory descriptor.
Return Diff destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

memory::*desc* **diff_dst_iter_c_desc**() **const**
Returns diff destination recurrent cell state memory descriptor.
Return Diff destination recurrent cell state memory descriptor.

struct dnnl::**gru_forward** : **public** dnnl::*primitive*
GRU forward propagation primitive.

Public Functions

gru_forward()
Default constructor. Produces an empty object.

gru_forward(**const** *primitive_desc* &*pd*)
Constructs a GRU forward propagation primitive.

Parameters

- pd: Primitive descriptor for a GRU forward propagation primitive.

struct desc
Descriptor for a GRU forward propagation primitive.

Public Functions

desc(*prop_kind* *aprop_kind*, *rnn_direction* *direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, *rnn_flags* *flags* = *rnn_flags*::*undef*)
Constructs a descriptor for a GRU forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- src_iter_desc,
- bias_desc,
- dst_iter_desc.

This would then indicate that the GRU forward propagation primitive should not use them and should default to zero values instead.

Note All memory descriptors except src_iter_desc may be initialized with an *dnnl::memory::format_tag::any* value of format_tag.

Parameters
- aprop_kind: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- direction: RNN direction. See *dnnl::rnn_direction* for more info.
- src_layer_desc: Memory descriptor for the input vector.

- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `flags`: Unused.

struct primitive_desc : **public** dnnl::*rnn_primitive_desc_base*

Primitive descriptor GRU forward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a GRU forward propagation primitive.

Parameters

- `adesc`: Descriptor for a GRU forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)

Constructs a primitive descriptor for a GRU forward propagation primitive.

Parameters

- `adesc`: Descriptor for a GRU forward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() **const**

Returns source layer memory descriptor.

Return Source layer memory descriptor.

memory::*desc* **src_iter_desc**() **const**

Returns source iteration memory descriptor.

Return Source iteration memory descriptor.

Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() **const**

Returns weights layer memory descriptor.

Return Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() **const**

Returns weights iteration memory descriptor.

Return Weights iteration memory descriptor.

memory::*desc* **bias_desc**() **const**

Returns bias memory descriptor.

Return Bias memory descriptor.

Return A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() **const**
Returns destination layer memory descriptor.
Return Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() **const**
Returns destination iteration memory descriptor.
Return Destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

struct dnnl::**gru_backward** : **public** dnnl::*primitive*
GRU backward propagation primitive.

Public Functions

gru_backward()
Default constructor. Produces an empty object.

gru_backward(**const** *primitive_desc* &*pd*)
Constructs a GRU backward propagation primitive.

Parameters

- pd: Primitive descriptor for a GRU backward propagation primitive.

struct desc
Descriptor for a GRU backward propagation primitive.

Public Functions

desc(*prop_kind* *aprop_kind*, *rnn_direction* *direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, **const** *memory*::*desc* &*diff_src_layer_desc*, **const** *memory*::*desc* &*diff_src_iter_desc*, **const** *memory*::*desc* &*diff_weights_layer_desc*, **const** *memory*::*desc* &*diff_weights_iter_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_layer_desc*, **const** *memory*::*desc* &*diff_dst_iter_desc*, *rnn_flags* *flags* = *rnn_flags*::*undef*)
Constructs a descriptor for a GRU backward propagation primitive.

The following arguments may point to a zero memory descriptor:
- src_iter_desc together with diff_src_iter_desc,
- bias_desc together with diff_bias_desc,
- dst_iter_desc together with diff_dst_iter_desc.

This would then indicate that the GRU backward propagation primitive should not use them and should default to zero values instead.

Note All memory descriptors may be initialized with *dnnl::memory::format_tag::any* value of format_tag.

Parameters
- aprop_kind: Propagation kind. Must be *dnnl::prop_kind::backward*.
- direction: RNN direction. See *dnnl::rnn_direction* for more info.

- src_layer_desc: Memory descriptor for the input vector.
- src_iter_desc: Memory descriptor for the input recurrent hidden state vector.
- weights_layer_desc: Memory descriptor for the weights applied to the layer input.
- weights_iter_desc: Memory descriptor for the weights applied to the recurrent input.
- bias_desc: Bias memory descriptor.
- dst_layer_desc: Memory descriptor for the output vector.
- dst_iter_desc: Memory descriptor for the output recurrent hidden state vector.
- diff_src_layer_desc: Memory descriptor for the diff of input vector.
- diff_src_iter_desc: Memory descriptor for the diff of input recurrent hidden state vector.
- diff_weights_layer_desc: Memory descriptor for the diff of weights applied to the layer input.
- diff_weights_iter_desc: Memory descriptor for the diff of weights applied to the recurrent input.
- diff_bias_desc: Diff bias memory descriptor.
- diff_dst_layer_desc: Memory descriptor for the diff of output vector.
- diff_dst_iter_desc: Memory descriptor for the diff of output recurrent hidden state vector.
- flags: Unused.

struct primitive_desc : **public** dnnl::*rnn_primitive_desc_base*

Primitive descriptor for a GRU backward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *gru_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a GRU backward propagation primitive.

Parameters

- adesc: Descriptor for a GRU backward propagation primitive.
- aengine: Engine to use.
- hint_fwd_pd: Primitive descriptor for a GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *gru_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for a GRU backward propagation primitive.

Parameters

- adesc: Descriptor for a GRU backward propagation primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.
- hint_fwd_pd: Primitive descriptor for a GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() **const**
Returns source layer memory descriptor.
Return Source layer memory descriptor.

memory::*desc* **src_iter_desc**() **const**
Returns source iteration memory descriptor.
Return Source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() **const**
Returns weights layer memory descriptor.
Return Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() **const**
Returns weights iteration memory descriptor.
Return Weights iteration memory descriptor.

memory::*desc* **bias_desc**() **const**
Returns bias memory descriptor.
Return Bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() **const**
Returns destination layer memory descriptor.
Return Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() **const**
Returns destination iteration memory descriptor.
Return Destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **diff_src_layer_desc**() **const**
Returns diff source layer memory descriptor.
Return Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() **const**
Returns diff source iteration memory descriptor.
Return Diff source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_weights_layer_desc**() **const**
Returns diff weights layer memory descriptor.
Return Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() **const**
Returns diff weights iteration memory descriptor.
Return Diff weights iteration memory descriptor.

memory::*desc* **diff_bias_desc**() **const**
Returns diff bias memory descriptor.
Return Diff bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() **const**
Returns diff destination layer memory descriptor.

Return Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() **const**
Returns diff destination iteration memory descriptor.
Return Diff destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

struct dnnl::**lbr_gru_forward** : **public** dnnl::*primitive*
LBR GRU forward propagation primitive.

Public Functions

lbr_gru_forward()
Default constructor. Produces an empty object.

lbr_gru_forward(**const** *primitive_desc* &*pd*)
Constructs an LBR GRU forward propagation primitive.

Parameters

- pd: Primitive descriptor for an LBR GRU forward propagation primitive.

struct desc
Descriptor for an LBR GRU forward propagation primitive.

Public Functions

desc(*prop_kind* *aprop_kind*, *rnn_direction* *direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, *rnn_flags* *flags* = *rnn_flags*::*undef*)
Constructs a descriptor for LBR GRU forward propagation primitive.

The following arguments may point to a zero memory descriptor:
- src_iter_desc,
- bias_desc,
- dst_iter_desc.

This would then indicate that the LBR GRU forward propagation primitive should not use them and should default to zero values instead.

Note All memory descriptors except src_iter_desc may be initialized with an *dnnl::memory::format_tag::any* value of format_tag.

Parameters
- aprop_kind: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- direction: RNN direction. See *dnnl::rnn_direction* for more info.
- src_layer_desc: Memory descriptor for the input vector.
- src_iter_desc: Memory descriptor for the input recurrent hidden state vector.
- weights_layer_desc: Memory descriptor for the weights applied to the layer input.
- weights_iter_desc: Memory descriptor for the weights applied to the recurrent input.
- bias_desc: Bias memory descriptor.
- dst_layer_desc: Memory descriptor for the output vector.
- dst_iter_desc: Memory descriptor for the output recurrent hidden state vector.
- flags: Unused.

struct primitive_desc : **public** dnnl::*rnn_primitive_desc_base*
Primitive descriptor for an LBR GRU forward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a LBR GRU forward propagation primitive.

Parameters
- `adesc`: Descriptor for a LBR GRU forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a LBR GRU forward propagation primitive.

Parameters
- `adesc`: Descriptor for a LBR GRU forward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() **const**
Returns source layer memory descriptor.
Return Source layer memory descriptor.

memory::*desc* **src_iter_desc**() **const**
Returns source iteration memory descriptor.
Return Source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() **const**
Returns weights layer memory descriptor.
Return Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() **const**
Returns weights iteration memory descriptor.
Return Weights iteration memory descriptor.

memory::*desc* **bias_desc**() **const**
Returns bias memory descriptor.
Return Bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() **const**
Returns destination layer memory descriptor.
Return Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() **const**
Returns destination iteration memory descriptor.
Return Destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.

Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

struct dnnl::**lbr_gru_backward** : **public** dnnl::*primitive*
LBR GRU backward propagation primitive.

Public Functions

lbr_gru_backward()
Default constructor. Produces an empty object.

lbr_gru_backward(**const** *primitive_desc* &*pd*)
Constructs an LBR GRU backward propagation primitive.

Parameters

- pd: Primitive descriptor for an LBR GRU backward propagation primitive.

struct desc
Descriptor for a LBR GRU backward propagation primitive.

Public Functions

desc(*prop_kind* *aprop_kind*, *rnn_direction* *direction*, **const** *memory*::*desc* &*src_layer_desc*, **const** *memory*::*desc* &*src_iter_desc*, **const** *memory*::*desc* &*weights_layer_desc*, **const** *memory*::*desc* &*weights_iter_desc*, **const** *memory*::*desc* &*bias_desc*, **const** *memory*::*desc* &*dst_layer_desc*, **const** *memory*::*desc* &*dst_iter_desc*, **const** *memory*::*desc* &*diff_src_layer_desc*, **const** *memory*::*desc* &*diff_src_iter_desc*, **const** *memory*::*desc* &*diff_weights_layer_desc*, **const** *memory*::*desc* &*diff_weights_iter_desc*, **const** *memory*::*desc* &*diff_bias_desc*, **const** *memory*::*desc* &*diff_dst_layer_desc*, **const** *memory*::*desc* &*diff_dst_iter_desc*, *rnn_flags* *flags* = *rnn_flags*::*undef*)
Constructs a descriptor for LBR GRU backward propagation primitive.

The following arguments may point to a zero memory descriptor:
- src_iter_desc together with diff_src_iter_desc,
- bias_desc together with diff_bias_desc,
- dst_iter_desc together with diff_dst_iter_desc.

This would then indicate that the LBR GRU backward propagation primitive should not use them and should default to zero values instead.

Note All memory descriptors may be initialized with *dnnl::memory::format_tag::any* value of format_tag.

Parameters
- aprop_kind: Propagation kind. Must be *dnnl::prop_kind::backward*.
- direction: RNN direction. See *dnnl::rnn_direction* for more info.
- src_layer_desc: Memory descriptor for the input vector.
- src_iter_desc: Memory descriptor for the input recurrent hidden state vector.
- weights_layer_desc: Memory descriptor for the weights applied to the layer input.
- weights_iter_desc: Memory descriptor for the weights applied to the recurrent input.
- bias_desc: Bias memory descriptor.
- dst_layer_desc: Memory descriptor for the output vector.
- dst_iter_desc: Memory descriptor for the output recurrent hidden state vector.
- diff_src_layer_desc: Memory descriptor for the diff of input vector.
- diff_src_iter_desc: Memory descriptor for the diff of input recurrent hidden state vector.

- `diff_weights_layer_desc`: Memory descriptor for the diff of weights applied to the layer input.
- `diff_weights_iter_desc`: Memory descriptor for the diff of weights applied to the recurrent input.
- `diff_bias_desc`: Diff bias memory descriptor.
- `diff_dst_layer_desc`: Memory descriptor for the diff of output vector.
- `diff_dst_iter_desc`: Memory descriptor for the diff of output recurrent hidden state vector.
- `flags`: Unused.

struct primitive_desc : **public** dnnl::*rnn_primitive_desc_base*

Primitive descriptor for an LBR GRU backward propagation primitive.

Public Functions

primitive_desc() = default

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *lbr_gru_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for an LBR GRU backward propagation primitive.

Parameters

- `adesc`: Descriptor for an LBR GRU backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an LBR GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *lbr_gru_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)

Constructs a primitive descriptor for an LBR GRU backward propagation primitive.

Parameters

- `adesc`: Descriptor for an LBR GRU backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an LBR GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_layer_desc**() **const**

Returns source layer memory descriptor.

Return Source layer memory descriptor.

memory::*desc* **src_iter_desc**() **const**

Returns source iteration memory descriptor.

Return Source iteration memory descriptor.

Return A zero memory descriptor if the primitive does not have a source iteration parameter.

memory::*desc* **weights_layer_desc**() **const**

Returns weights layer memory descriptor.

Return Weights layer memory descriptor.

memory::*desc* **weights_iter_desc**() **const**
Returns weights iteration memory descriptor.
Return Weights iteration memory descriptor.

memory::*desc* **bias_desc**() **const**
Returns bias memory descriptor.
Return Bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a bias parameter.

memory::*desc* **dst_layer_desc**() **const**
Returns destination layer memory descriptor.
Return Destination layer memory descriptor.

memory::*desc* **dst_iter_desc**() **const**
Returns destination iteration memory descriptor.
Return Destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination iteration parameter.

memory::*desc* **workspace_desc**() **const**
Returns the workspace memory descriptor.
Return Workspace memory descriptor.
Return A zero memory descriptor if the primitive does not require workspace parameter.

memory::*desc* **diff_src_layer_desc**() **const**
Returns diff source layer memory descriptor.
Return Diff source layer memory descriptor.

memory::*desc* **diff_src_iter_desc**() **const**
Returns diff source iteration memory descriptor.
Return Diff source iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff source iteration parameter.

memory::*desc* **diff_weights_layer_desc**() **const**
Returns diff weights layer memory descriptor.
Return Diff weights layer memory descriptor.

memory::*desc* **diff_weights_iter_desc**() **const**
Returns diff weights iteration memory descriptor.
Return Diff weights iteration memory descriptor.

memory::*desc* **diff_bias_desc**() **const**
Returns diff bias memory descriptor.
Return Diff bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::*desc* **diff_dst_layer_desc**() **const**
Returns diff destination layer memory descriptor.
Return Diff destination layer memory descriptor.

memory::*desc* **diff_dst_iter_desc**() **const**
Returns diff destination iteration memory descriptor.
Return Diff destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

5.5.17 Shuffle

The shuffle primitive shuffles data along the shuffle axis (here is designated as C) with the group parameter G. Namely, the shuffle axis is thought to be a 2D tensor of size $(\frac{C}{G} \times G)$ and it is being transposed to $(G \times \frac{C}{G})$. Variable names follow the standard *Conventions*.

The formal definition is shown below:

Forward

$$\mathrm{dst}(\overline{ou}, c, \overline{in}) = \mathrm{src}(\overline{ou}, c', \overline{in})$$

where

- c dimension is called a shuffle `axis`,
- G is a `group_size`,
- $\overline{ou}$ is the outermost indices (to the left from shuffle axis),
- $\overline{in}$ is the innermost indices (to the right from shuffle axis), and
- c' and c relate to each other as define by the system:

$$\begin{cases} c &= u + v \cdot \frac{C}{G}, \\ c' &= u \cdot G + v, \end{cases}$$

Here, $0 \leq u < \frac{C}{G}$ and $0 \leq v < G$.

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes $\mathrm{diff\_src}(ou, c, in)$, based on $\mathrm{diff\_dst}(ou, c, in)$.

Essentially, backward propagation is the same as forward propagation with g replaced by C/g.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. The memory format and data type for `src` and `dst` are assumed to be the same, and in the API are typically referred as `data` (e.g., see `data_desc` in *dnnl::shuffle_forward::desc::desc()*). The same holds for `diff_src` and `diff_dst`. The corresponding memory descriptors are referred to as `diff_data_desc`.

Data Types Support

The shuffle primitive supports the following combinations of data types:

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*f32*, *bf16*
forward	*s32*, *s8*, *u8*

Data Layouts

The shuffle primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions. However, the shuffle axis is typically referred to as channels (hence in formulas we use c).

Shuffle operation typically appear in CNN topologies. Hence, in the library the shuffle primitive is optimized for the corresponding memory formats:

Spatial	Logical tensor	Shuffle Axis	Implementations optimized for memory formats
2D	NCHW	1 (C)	*nchw* (*abcd*), *nhwc* (*acdb*), *optimized^*
3D	NCDHW	1 (C)	*ncdhw* (*abcde*), *ndhwc* (*acdeb*), *optimized^*

Here *optimized^* means the format that comes out of any preceding compute-intensive primitive.

Post-ops and Attributes

The shuffle primitive does not have to support any post-ops or attributes.

API

struct dnnl::**shuffle_forward** : **public** dnnl::*primitive*

Shuffle forward propagation primitive.

Public Functions

shuffle_forward()
Default constructor. Produces an empty object.

shuffle_forward(**const** *primitive_desc* &*pd*)
Constructs a shuffle forward propagation primitive.

Parameters

- pd: Primitive descriptor for a shuffle forward propagation primitive.

struct desc
Descriptor for a shuffle forward propagation primitive.

Public Functions

desc(*prop_kind aprop_kind*, **const** *memory*::*desc* &*data_desc*, int *axis*, int *group_size*)
Constructs a descriptor for a shuffle forward propagation primitive.

Parameters

- aprop_kind: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- data_desc: Source and destination memory descriptor.
- axis: The axis along which the data is shuffled.
- group_size: Shuffle group size.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for a shuffle forward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *primitive_attr* &*attr* = *primitive_attr*(), bool *allow_empty* = false)
Constructs a primitive descriptor for a shuffle forward propagation primitive.

Parameters

- adesc: Descriptor for a shuffle forward propagation primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.
- allow_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

struct dnnl::**shuffle_backward** : **public** dnnl::*primitive*

Shuffle backward propagation primitive.

Public Functions

shuffle_backward()

Default constructor. Produces an empty object.

shuffle_backward(**const** *primitive_desc* &*pd*)

Constructs a shuffle backward propagation primitive.

Parameters

- pd: Primitive descriptor for a shuffle backward propagation primitive.

struct desc

Descriptor for a shuffle primitive backward propagation primitive.

Public Functions

desc(**const** *memory*::*desc* &*diff_data_desc*, int *axis*, int *group_size*)

Constructs a descriptor for a shuffle backward propagation primitive.

Parameters

- diff_data_desc: Diff source and diff destination memory descriptor.
- axis: The axis along which the data is shuffled.
- group_size: Shuffle group size.

struct primitive_desc : **public** dnnl::*primitive_desc*

Primitive descriptor for a shuffle backward propagation primitive.

Public Functions

primitive_desc()

Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *shuffle_forward*::*primitive_desc* &*hint_fwd_pd*, **const** *primitive_attr* &*attr* = *primitive_attr*(), bool *allow_empty* = false)

Constructs a primitive descriptor for a shuffle backward propagation primitive.

Parameters

- adesc: Descriptor for a shuffle backward propagation primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.
- hint_fwd_pd: Primitive descriptor for a shuffle forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **diff_src_desc**() **const**

Returns a diff source memory descriptor.

Return Diff source memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc() const**

Returns a diff destination memory descriptor.

Return Diff destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff destination parameter.

5.5.18 Softmax

The softmax primitive performs softmax along a particular axis on data with arbitrary dimensions. All other axes are treated as independent (batch).

In general form, the operation is defined by the following formulas. The variable names follow the standard *Conventions*.

Forward

$$\mathrm{dst}(\overline{ou}, c, \overline{in}) = \frac{e^{\mathrm{src}(\overline{ou}, c, \overline{in}) - \nu(\overline{ou}, \overline{in})}}{\sum\limits_{ic} e^{\mathrm{src}(\overline{ou}, ic, \overline{in}) - \nu(\overline{ou}, \overline{in})}},$$

where

- c axis over which the softmax computation is computed on,
- $\overline{ou}$ is the outermost index (to the left of softmax axis),
- $\overline{in}$ is the innermost index (to the right of softmax axis), and
- ν is used to produce more accurate results and defined as:

$$\nu(\overline{ou}, \overline{in}) = \max_{ic} \mathrm{src}(\overline{ou}, ic, \overline{in})$$

Difference Between Forward Training and Forward Inference

There is no difference between the *forward_training* and *forward_inference* propagation kinds.

Backward

The backward propagation computes $\mathrm{diff\_src}(ou, c, in)$, based on $\mathrm{diff\_dst}(ou, c, in)$ and $\mathrm{dst}(ou, c, in)$.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	*DNNL_ARG_SRC*
dst	*DNNL_ARG_DST*
diff_src	*DNNL_ARG_DIFF_SRC*
diff_dst	*DNNL_ARG_DIFF_DST*

Operation Details

1. Both forward and backward propagation support in-place operations, meaning that `src` can be used as input and output for forward propagation, and `diff_dst` can be used as input and output for backward propagation. In case of in-place operation, the original data will be overwritten.

Post-ops and Attributes

The softmax primitive does not have to support any post-ops or attributes.

Data Types Support

The softmax primitive supports the following combinations of data types.

Note: Here we abbreviate data types names for readability. For example, *dnnl::memory::data_type::f32* is abbreviated to *f32*.

Propagation	Source / Destination
forward / backward	*bf16*, *f32*
forward	*f16*

Data Representation

Source, Destination, and Their Gradients

The softmax primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions. However, the softmax axis is typically referred to as channels (hence in formulas we use c).

API

struct dnnl::**softmax_forward** : **public** dnnl::*primitive*

Softmax forward propagation primitive.

Public Functions

softmax_forward()

Default constructor. Produces an empty object.

softmax_forward(**const** *primitive_desc* &*pd*)

Constructs a softmax forward propagation primitive.

Parameters

- `pd`: Primitive descriptor for a softmax forward propagation primitive.

struct desc

Descriptor for a softmax forward propagation primitive.

Public Functions

desc()
Default constructor. Produces an empty object.

desc(*prop_kind aprop_kind*, **const** *memory*::*desc* &*data_desc*, int *softmax_axis*)
Constructs a descriptor for a softmax forward propagation primitive.

Parameters
- `aprop_kind`: Propagation kind. Possible values are *dnnl::prop_kind::forward_training*, and *dnnl::prop_kind::forward_inference*.
- `data_desc`: Source and destination memory descriptor.
- `softmax_axis`: Axis over which softmax is computed.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for a softmax forward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a softmax forward propagation primitive.

Parameters
- `adesc`: descriptor for a softmax forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, bool *allow_empty* = false)
Constructs a primitive descriptor for a softmax forward propagation primitive.

Parameters
- `adesc`: Descriptor for a softmax forward propagation primitive.
- `aengine`: Engine to use.
- `attr`: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **src_desc**() **const**
Returns a source memory descriptor.
Return Source memory descriptor.
Return A zero memory descriptor if the primitive does not have a source parameter.

memory::*desc* **dst_desc**() **const**
Returns a destination memory descriptor.
Return Destination memory descriptor.
Return A zero memory descriptor if the primitive does not have a destination parameter.

struct dnnl::**softmax_backward** : **public** dnnl::*primitive*
Softmax backward propagation primitive.

Public Functions

softmax_backward()
Default constructor. Produces an empty object.

softmax_backward(**const** *primitive_desc* &*pd*)
Constructs a softmax backward propagation primitive.

Parameters

- `pd`: Primitive descriptor for a softmax backward propagation primitive.

struct desc
Descriptor for a softmax backward propagation primitive.

Public Functions

desc()
Default constructor. Produces an empty object.

desc(**const** *memory*::*desc* &*diff_data_desc*, **const** *memory*::*desc* &*data_desc*, int *softmax_axis*)
Constructs a descriptor for a softmax backward propagation primitive.

Parameters

- `diff_data_desc`: Diff source and diff destination memory descriptor.
- `data_desc`: Destination memory descriptor.
- `softmax_axis`: Axis over which softmax is computed.

struct primitive_desc : **public** dnnl::*primitive_desc*
Primitive descriptor for a softmax backward propagation primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *desc* &*adesc*, **const** *engine* &*aengine*, **const** *softmax_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for a softmax backward propagation primitive.

Parameters

- `adesc`: Descriptor for a softmax backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a softmax forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

primitive_desc(**const** *desc* &*adesc*, **const** *primitive_attr* &*attr*, **const** *engine* &*aengine*, **const** *softmax_forward*::*primitive_desc* &*hint_fwd_pd*, bool *allow_empty* = false)
Constructs a primitive descriptor for a softmax backward propagation primitive.

Parameters

- `adesc`: Descriptor for a softmax backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.

- `hint_fwd_pd`: Primitive descriptor for a softmax forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

memory::*desc* **dst_desc() const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

memory::*desc* **diff_src_desc() const**

Returns a diff source memory descriptor.

Return Diff source memory descriptor.

Return A zero memory descriptor if the primitive does not have a diff source memory with.

memory::*desc* **diff_dst_desc() const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

5.5.19 Sum

The sum primitive sums N tensors. The variable names follow the standard *Conventions*.

$$\mathrm{dst}(\overline{x}) = \sum_{i=1}^{N} scales(i) \cdot \mathrm{src}_i(\overline{x})$$

The sum primitive does not have a notion of forward or backward propagations. The backward propagation for the sum operation is simply an identity operation.

Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

primitive input/output	execution argument index
src	*DNNL_ARG_MULTIPLE_SRC*
dst	*DNNL_ARG_DST*

Operation Details

- The dst memory format can be either specified by a user or derived the most appropriate one by the primitive. The recommended way is to allow the primitive to choose the appropriate format.
- The sum primitive requires all source and destination tensors to have the same shape. Implicit broadcasting is not supported.

Post-ops and Attributes

The sum primitive does not support any post-ops or attributes.

Data Types Support

The sum primitive supports arbitrary data types for source and destination tensors.

Data Representation

Sources, Destination

The sum primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

API

struct dnnl::**sum** : **public** dnnl::*primitive*
Out-of-place summation (sum) primitive.

Public Functions

sum()
Default constructor. Produces an empty object.

sum(**const** *primitive_desc* &*pd*)
Constructs a sum primitive.

Parameters

- pd: Primitive descriptor for sum primitive.

struct primitive_desc : **public** dnnl::*primitive_desc_base*
Primitive descriptor for a sum primitive.

Public Functions

primitive_desc()
Default constructor. Produces an empty object.

primitive_desc(**const** *memory*::*desc* &*dst*, **const** std::vector<float> &*scales*, **const** std::vector<*memory*::*desc*> &*srcs*, **const** *engine* &*aengine*, **const** *primitive_attr* &*attr* = *primitive_attr*())
Constructs a primitive descriptor for a sum primitive.

Parameters

- dst: Destination memory descriptor.
- scales: Vector of scales to multiply data in each source memory by.
- srcs: Vector of source memory descriptors.
- aengine: Engine to perform the operation on.
- attr: Primitive attributes to use (optional).

primitive_desc(**const** std::vector<float> &*scales*, **const** std::vector<*memory*::*desc*> &*srcs*, **const** *engine* &*aengine*, **const** *primitive_attr* &*attr* = *primitive_attr*())

Constructs a primitive descriptor for a sum primitive.

This version derives the destination memory descriptor automatically.

Parameters

- `scales`: Vector of scales by which to multiply data in each source memory object.
- `srcs`: Vector of source memory descriptors.
- `aengine`: Engine on which to perform the operation.
- `attr`: Primitive attributes to use (optional).

memory::*desc* **src_desc**(int *idx* = 0) **const**

Returns a source memory descriptor.

Return Source memory descriptor.

Return A zero memory descriptor if the primitive does not have a source parameter with index `pdx`.

Parameters

- `idx`: Source index.

memory::*desc* **dst_desc**() **const**

Returns a destination memory descriptor.

Return Destination memory descriptor.

Return A zero memory descriptor if the primitive does not have a destination parameter.

5.6 Open Source Implementation

Intel has published an open source implementation with the Apache license.

5.7 Implementation Notes

This specification provides high-level descriptions for oneDNN operations and does not cover all the implementation-specific details of the open source implementation. Specifically, it does not cover highly-optimized memory formats and integration with profiling tools, etc. This is done intentionally to improve specification portability. Code that uses API defined in this specification is expected to be portable across open source implementation and any potential other implementations of this specification to a reasonable extent.

In the future this section will be extended with more details on how different implementations of this specification should cooperate and co-exist.

5.8 Testing

Intel's binary distribution of oneDNN contains example code that you can be used to test library functionality.

The open source implementation includes a comprehensive test suite. Consult the README for directions.

CHAPTER

SIX

ONECCL

6.1 Introduction

The oneAPI Collective Communications Library (oneCCL) provides primitives for the communication patterns that occur in deep learning applications. oneCCL supports both scale-up for platforms with multiple oneAPI devices and scale-out for clusters with multiple compute nodes.

oneCCL supports the following communication patterns used in deep learning (DL) algorithms:

- allgatherv
- allreduce
- alltoallv
- broadcast
- reduce
- reduce_scatter

oneCCL exposes controls over additional optimizations and capabilities such as:

- Prioritization for communication operations
- Persistent communication operations (enables decoupling one-time initialization and repetitive execution)

6.2 Namespaces

This section describes the oneCCL namespace conventions.

6.2.1 oneapi::ccl namespace

The `oneapi::ccl` namespace shall contains public identifiers defined by the library.

6.2.2 ccl namespace

The alternative `ccl` namespace shall be considered an alias for the `oneapi::ccl` namespace.

6.3 Current Version of this oneCCL Specification

This is the oneCCL specification version 1.0.

6.4 Definitions

6.4.1 oneCCL Concepts

oneCCL specification defines the following list of concepts:

- *Device*
- *Context*
- *Key-Value Store*
- *Communicator*
- *Stream*
- *Event*
- *Operation Attributes*

Device

Note: Here and below, a native device/context/stream/event are defined in the scope of SYCL device runtime

```
using native_device_type = sycl::device;
using native_context_type = sycl::context;
using native_stream_type = sycl::queue;
using native_event_type = sycl::event;
```

oneCCL specification defines `device` as an abstraction of a computational device: a CPU, a specific GPU card in the system, or any other device participating in a communication operation. `device` corresponds to the communicator's rank (addressable entity in a communication operation).

oneCCL specification defines the way to create an instance of the `device` class with a native object (`native_device_type`) and without a native object (corresponds to the host).

Creating a new device object:

```
device ccl::create_device(native_device_type& native_device);

device ccl::create_device();
```

native_device the existing native device object

return `device` a device object

`device` class shall provide ability to retrieve a native object.

Retrieving a native device object:

```
native_device_type device::get_native();
```

return `native_device_type`

a native device object
shall throw exception if a `device` object does not wrap the native object

Context

oneCCL specification defines `context` as an abstraction of a computational devices context that is responsible for managing resources and for executing of communication operations on one or more devices specified in the context.

oneCCL specification defines the way to create an instance of the `context` class with a native object (`native_context_type`) and without a native object.

Creating a new context object:

```
context ccl::create_context(native_context_type& native_context);

context ccl::create_context();
```

native_context the existing native context object

return `context` a context object

`context` class shall provide ability to retrieve a native object.

Retrieving a native context object:

```
native_context_type context::get_native();
```

return `native_context_type`

a native context object
shall throw exception if a `context` object does not wrap the native object

Key-Value Store

`kvs_interface` defines the key-value store (KVS) interface to be used to establish connection between ranks during the creation of oneCCL communicator. The interface shall include blocking `get` and `set` methods.

Getting a record from the key-value store:

```
virtual vector_class<char> kvs_interface::get(
    const string_class& key) = 0;
```

key the key of value to be retrieved

return `vector_class<char>` the value associated with the given key

Note: `get` operation with a non-existing key shall return empty result

Saving a record in the key-value store:

```
void kvs_interface::set(
    const string_class& key,
    const vector_class<char>& data) = 0;
```

key the key at which the value should be stored

data the value that should be associated with the given key

Note: `set` operation with empty `data` shall remove a record from the key-value store

---

oneCCL specification defines `kvs` class as a built-in KVS provided by oneCCL.

```
class kvs : public kvs_interface {

public:

static constexpr size_t address_max_size = 256;
using address_type = array_class<char, address_max_size>;

~kvs() override;

address_type get_address() const;

vector_class<char> get(
    const string_class& key) override;

void set(
    const string_class& key,
    const vector_class<char>& data) override;

}
```

Retrieving an address of built-in key-value store:

```
kvs::address_type kvs::get_address() const;
```

return `kvs::address_type`

the address of the key-value store
should be retrieved from the main built-in KVS and distributed to other processes for the built-in KVS creation

Creating a main built-in key-value store. Its address should be distributed using an out-of-band communication mechanism and be used to create key-value stores on other ranks:

```
shared_ptr_class<kvs> ccl::create_main_kvs();
```

return `shared_ptr_class<kvs>` the main key-value store object

Creating a new key-value store from main kvs address:

```
shared_ptr_class<kvs> ccl::create_kvs(const kvs::address_type& addr);
```

addr the address of the main kvs

return `shared_ptr_class<kvs>` key-value store object

Communicator

oneCCL specification defines `communicator` class that describes a group of communicating ranks, where a rank is an addressable entity in a communication operation and corresponds to single oneCCL device.

`communicator` defines communication operations on memory buffers between homogenous oneCCL devices, that is, all oneCCL devices either wrap native device objects of the same type (for example CPUs only or GPUs only) or do not wrap native objects.

Each process may correspond to multiple ranks.

Note: Support for multiple ranks per process is optional

Creating a new communicator(s) with user-supplied communicator size, rank-to-device mapping/rank, context and kvs:

Note: If `device` and `context` objects are omitted, then they are created with `ccl::create_device()` and `ccl::create_context()` functions without native objects

---

```
vector_class<communicator> ccl::create_communicators(
    int size,
    const map_class<int, device>& rank_device_map,
    const context& context,
    shared_ptr_class<kvs_interface> kvs);

communicator ccl::create_communicator(
    int size,
    int rank,
    shared_ptr_class<kvs_interface> kvs);
```

size user-supplied total number of ranks

rank_device_map user-supplied mapping of local ranks on devices

rank user-supplied local rank

context device context

kvs key-value store for ranks wire-up

return `vector_class<communicator>` / `communicator` a vector of communicator objects / a communicator object

`communicator` shall provide methods to retrieve the rank, the device, and the context that correspond to the communicator object as well as the total number of ranks in the communicator.

Retrieving the rank in a communicator:

```
int communicator::rank() const;
```

return `int` the rank that corresponds to the communicator object

Retrieving the total number of ranks in a communicator:

```
int communicator::size() const;
```

return `int` the total number of the ranks

Retrieving an underlying device, which was used as communicator construction argument:

```
device communicator::get_device() const;
```

return `device` the device that corresponds to the communicator object

Retrieving an underlying context, which was used as communicator construction argument:

```
context communicator::get_context() const;
```

return `context` the context that corresponds to the communicator object

---

Note: See also: *Collective Operations*

Stream

oneCCL specification defines `stream` as an abstraction that encapsulates execution context for `communicator` communication operations.

Stream shall be passed to `communicator` communication operation.

oneCCL specification defines the way to create an instance of the `stream` class with a native object (`native_stream_type`) and without a native object.

Creating a new stream object:

```
stream ccl::create_stream(native_stream_type& native_stream);

stream ccl::create_stream();
```

native_stream the existing native stream object

return `stream` a stream object

`stream` class shall provide ability to retrieve a native object.

Retrieving a native stream object:

```
native_stream_type stream::get_native();
```

return `native_stream_type`

a native stream object

shall throw exception if a `stream` object does not wrap the native object

Event

oneCCL specification defines `event` as an abstraction that encapsulates synchronization context for `communicator` communication operations.

Each communication operation of oneCCL shall return an event object for tracking the operation's progress. A vector of events may be passed to the `communicator` communication operation to designate input dependencies for the operation.

Note: Support for handling of input events is optional

---

oneCCL specification defines the way to create an instance of the `event` class with a native object (`native_event_type`).

Creating a new event object:

```
event ccl::create_event(native_event_type& native_event);
```

native_event the existing native event object

return `event` an event object

`event` class shall provide ability to retrieve a native object.

Retrieving a native event object:

```
native_event_type event::get_native();
```

return `native_event_type`

a native event object
shall throw exception if an `event` object does not wrap the native object

---

Note: See also: *Operation Progress Tracking*

Operation Attributes

Communication operation behavior may be controlled through operation attributes.

Operation Attributes

6.4.2 Communication Operations

This section covers communication operations defined by oneCCL specification.

Datatypes

oneCCL specification defines the following datatypes that may be used for communication operations:

```
enum class datatype : int
{
    int8            = /* unspecified */,
    uint8           = /* unspecified */,
    int16           = /* unspecified */,
    uint16          = /* unspecified */,
    int32           = /* unspecified */,
    uint32          = /* unspecified */,
    int64           = /* unspecified */,
    uint64          = /* unspecified */,

    float16         = /* unspecified */,
    float32         = /* unspecified */,
    float64         = /* unspecified */,
    bfloat16        = /* unspecified */,
```

(continues on next page)

(continued from previous page)

```
    last_predefined = /* unspecified, equal to the largest of all the values above */
};
```

datatype::int8 8 bits signed integer

datatype::uint8 8 bits unsigned integer

datatype::int16 16 bits signed integer

datatype::uint16 16 bits unsigned integer

datatype::int32 32 bits signed integer

datatype::uint32 32 bits unsigned integer

datatype::int64 64 bits signed integer

datatype::uint64 64 bits unsigned integer

datatype::float16 16-bit/half-precision floating point

datatype::float32 32-bit/single-precision floating point

datatype::float64 64-bit/double-precision floating point

datatype::bfloat16 non-standard 16-bit floating point with 7-bit mantissa

---

Note: Support for `datatype::float16` is optional

Custom Datatypes

oneCCL specification defines the way to register and deregister a custom datatype using the `datatype_attr` attribute object.

The list of identifiers that may be used to fill an attribute object:

```
enum class datatype_attr_id {
    size = /* unspecified */
};
```

datatype_attr_id::size the size of the datatype in bytes

Creating a datatype attribute object, which may used to register custom datatype:

```
datatype_attr ccl::create_datatype_attr();
```

return `datatype_attr` an object containing attributes for the custom datatype

Registering a custom datatype to be used in communication operations:

```
datatype ccl::register_datatype(const datatype_attr& attr);
```

attr the datatype's attributes

return `datatype` the handle for the custom datatype

Deregistering a custom datatype:

```
void ccl::deregister_datatype(datatype dtype);
```

dtype the handle for the custom datatype

Retrieving a datatype size in bytes:

```
size_t ccl::get_datatype_size(datatype dtype);
```

dtype the datatype's handle

return `size_t` datatype size in bytes

Reductions

oneCCL specification defines the following reduction operations for *Allreduce*, *Reduce* and *ReduceScatter* collective operations:

```
enum class reduction
{
    sum    = /* unspecified */,
    prod   = /* unspecified */,
    min    = /* unspecified */,
    max    = /* unspecified */,
    custom = /* unspecified */
};
```

reduction::sum elementwise summation

reduction::prod elementwise multiplication

reduction::min elementwise min

reduction::max elementwise max

reduction::custom

specify user-defined reduction operation
the actual reduction function must be passed through `reduction_fn` operation attribute

Operation Attributes

Collective Operations

oneCCL specification defines the following collective communication operations:

- *Allgatherv*
- *Allreduce*
- *Alltoallv*
- *Barrier*
- *Broadcast*
- *Reduce*
- *ReduceScatter*

These operations are collective, meaning that all participants (ranks) of oneCCL communicator should make a call. The order of collective operation calls should be the same across all ranks.

`communicator` shall provide the ability to perform communication operations either on host or device memory buffers depending on the `device` used to create the communicator. Additionally, communication operations shall

accept an execution context (stream) and may accept a vector of events that the communication operation should depend on, that is, input dependencies. The output `event` object shall provide the ability to track the progress of the operation.

Note: Support for handling of input events is optional

`BufferType` is used below to define the C++ type of elements in data buffers (`buf`, `send_buf` and `recv_buf`) of communication operations. At least the following types shall be supported: `[u]int{8/16/32/64}_t`, `float`, `double`. The explicit `datatype` parameter shall be used to enable data types which cannot be inferred from the function arguments.

Note: See also: *Custom Datatypes*

The communication operation accepts a `stream` object. If a communicator is created from `native_device_type`, then the stream shall translate to `native_stream_type` created from the corresponding device.

The communication operation may accept attribute object. If that parameter is missed, then the default attribute object is used (default_<operation_name>_attr). The default attribute object shall be provided by the library.

Note: See also: *Operation Attributes*

If the arguments provided to a communication operation call do not comply to the requirements of the operation, the behavior is undefined unless it is specified otherwise.

Allgatherv

Allgatherv is a collective communication operation that collects data from all the ranks within a communicator into a single buffer. Different ranks may contribute segments of different sizes. The resulting data in the output buffer must be the same for each rank.

```
template<class BufferType>
event ccl::allgatherv(const BufferType* send_buf,
                      size_t send_count,
                      BufferType* recv_buf,
                      const vector_class<size_t>& recv_counts,
                      const communicator& comm,
                      const stream& stream,
                      const allgatherv_attr& attr = default_allgatherv_attr,
                      const vector_class<event>& deps = {});

event ccl::allgatherv(const void* send_buf,
                      size_t send_count,
                      void* recv_buf,
                      const vector_class<size_t>& recv_counts,
                      datatype dtype,
                      const communicator& comm,
                      const stream& stream,
                      const allgatherv_attr& attr = default_allgatherv_attr,
                      const vector_class<event>& deps = {});
```

send_buf the buffer with `send_count` elements of `BufferType` that stores local data to be gathered

send_count the number of elements of type `BufferType` in `send_buf`

recv_buf [out] the buffer to store the gathered result, must be large enough to hold values from all ranks

recv_counts

an array with the number of elements of type `BufferType` to be received from each rank
the array's size must be equal to the number of ranks
the values in the array are expected to be the same for all ranks
the value at the position of the caller's rank must be equal to `send_count`

dtype

the datatype of elements in `send_buf` and `recv_buf`
must be skipped if `BufferType` can be inferred
otherwise must be passed explicitly

comm the communicator that defines a group of ranks for the operation

stream the stream associated with the operation

attr optional attributes to customize the operation

deps an optional vector of the events that the operation should depend on

return `event` an object to track the progress of the operation

Allreduce

Allreduce is a collective communication operation that performs the global reduction operation on values from all ranks of communicator and distributes the result back to all ranks.

```
template <class BufferType>
event ccl::allreduce(const BufferType* send_buf,
                     BufferType* recv_buf,
                     size_t count,
                     reduction rtype,
                     const communicator& comm,
                     const stream& stream,
                     const allreduce_attr& attr = default_allreduce_attr,
                     const vector_class<event>& deps = {});

event ccl::allreduce(const void* send_buf,
                     void* recv_buf,
                     size_t count,
                     reduction rtype,
                     datatype dtype,
                     const communicator& comm,
                     const stream& stream,
                     const allreduce_attr& attr = default_allreduce_attr,
                     const vector_class<event>& deps = {});
```

send_buf the buffer with `count` elements of `BufferType` that stores local data to be reduced

recv_buf [out] the buffer to store the reduced result, must have the same dimension as `send_buf`

count the number of elements of type `BufferType` in `send_buf` and `recv_buf`

rtype the type of the reduction operation to be applied

dtype

the datatype of elements in `send_buf` and `recv_buf`
must be skipped if `BufferType` can be inferred
otherwise must be passed explicitly

comm the communicator that defines a group of ranks for the operation

stream the stream associated with the operation

attr optional attributes to customize the operation

deps an optional vector of the events that the operation should depend on

return `event` an object to track the progress of the operation

Alltoallv

Alltoall is a collective communication operation in which each rank sends separate blocks of data to each rank. Block sizes may differ. The j-th block of send buffer sent from the i-th rank is received by the j-th rank and is placed in the i-th block of receive buffer.

```
template <class BufferType>
event ccl::alltoallv(const BufferType* send_buf,
                     const vector_class<size_t>& send_counts,
                     BufferType* recv_buf,
                     const vector_class<size_t>& recv_counts,
                     const communicator& comm,
                     const stream& stream,
                     const alltoallv_attr& attr = default_alltoallv_attr,
                     const vector_class<event>& deps = {});

event ccl::alltoallv(const void* send_buf,
                     const vector_class<size_t>& send_counts,
                     void* recv_buf,
                     const vector_class<size_t>& recv_counts,
                     datatype dtype,
                     const communicator& comm,
                     const stream& stream,
                     const alltoallv_attr& attr = default_alltoallv_attr,
                     const vector_class<event>& deps = {});
```

send_buf the buffer with elements of `BufferType` that stores local blocks to be sent to each rank

send_counts

an array with number of elements of type `BufferType` in the blocks sent for each rank
the array's size must be equal to the number of ranks
the values at the position of the caller's rank in `send_counts` and `recv_counts` must be equal

recv_buf [out] the buffer to store the received result, must be large enough to hold blocks from all ranks

recv_counts

an array with number of elements of type `BufferType` in the blocks received from each rank
the array's size must be equal to the number of ranks
the values at the position of the caller's rank in `send_counts` and `recv_counts` must be equal

dtype

the datatype of elements in `send_buf` and `recv_buf`
must be skipped if `BufferType` can be inferred

otherwise must be passed explicitly

comm the communicator that defines a group of ranks for the operation

stream the stream associated with the operation

attr optional attributes to customize the operation

deps an optional vector of the events that the operation should depend on

return `event` an object to track the progress of the operation

Barrier

Barrier synchronization is performed across all ranks of the communicator and it is completed only after all the ranks in the communicator have called it.

```
event ccl::barrier(const communicator& comm,
                   const stream& stream,
                   const barrier_attr& attr = default_barrier_attr,
                   const vector_class<event>& deps = {});
```

comm the communicator that defines a group of ranks for the operation

stream the stream associated with the operation

attr optional attributes to customize the operation

deps an optional vector of the events that the operation should depend on

return `event` an object to track the progress of the operation

Broadcast

Broadcast is a collective communication operation that broadcasts data from one rank of communicator (denoted as root) to all other ranks.

```
template <class BufferType>
event ccl::broadcast(BufferType* buf,
                     size_t count,
                     int root,
                     const communicator& comm,
                     const stream& stream,
                     const broadcast_attr& attr = default_broadcast_attr,
                     const vector_class<event>& deps = {});

event ccl::broadcast(void* buf,
                     size_t count,
                     datatype dtype,
                     int root,
                     const communicator& comm,
                     const stream& stream,
                     const broadcast_attr& attr = default_broadcast_attr,
                     const vector_class<event>& deps = {});
```

buf [in,out]

the buffer with `count` elements of `BufferType`
serves as `send_buf` for root and as `recv_buf` for other ranks

count the number of elements of type `BufferType` in `buf`

root the rank that broadcasts `buf`

dtype

the datatype of elements in `buf`
must be skipped if `BufferType` can be inferred
otherwise must be passed explicitly

comm the communicator that defines a group of ranks for the operation

stream the stream associated with the operation

attr optional attributes to customize the operation

deps an optional vector of the events that the operation should depend on

return `event` an object to track the progress of the operation

Reduce

Reduce is a collective communication operation that performs the global reduction operation on values from all ranks of the communicator and returns the result to the root rank.

```
template <class BufferType>
event ccl::reduce(const BufferType* send_buf,
                  BufferType* recv_buf,
                  size_t count,
                  reduction rtype,
                  int root,
                  const communicator& comm,
                  const stream& stream,
                  const reduce_attr& attr = default_reduce_attr,
                  const vector_class<event>& deps = {});

event ccl::reduce(const void* send_buf,
                  void* recv_buf,
                  size_t count,
                  datatype dtype,
                  reduction rtype,
                  int root,
                  const communicator& comm,
                  const stream& stream,
                  const reduce_attr& attr = default_reduce_attr,
                  const vector_class<event>& deps = {});
```

send_buf the buffer with `count` elements of `BufferType` that stores local data to be reduced

recv_buf [out]

the buffer to store the reduced result, must have the same dimension as `send_buf`.
Used by the `root` rank only, ignored by other ranks.

count the number of elements of type `BufferType` in `send_buf` and `recv_buf`

rtype the type of the reduction operation to be applied

root the rank that gets the result of the reduction

dtype

the datatype of elements in `send_buf` and `recv_buf`
must be skipped if `BufferType` can be inferred
otherwise must be passed explicitly

comm the communicator that defines a group of ranks for the operation

stream the stream associated with the operation

attr optional attributes to customize the operation

deps an optional vector of the events that the operation should depend on

return `event` an object to track the progress of the operation

ReduceScatter

Reduce-scatter is a collective communication operation that performs the global reduction operation on values from all ranks of the communicator and scatters the result in blocks back to all ranks.

```
template <class BufferType>
event ccl::reduce_scatter(const BufferType* send_buf,
                          BufferType* recv_buf,
                          size_t recv_count,
                          reduction rtype,
                          const communicator& comm,
                          const stream& stream,
                          const reduce_scatter_attr& attr = default_reduce_scatter_
↪attr,
                          const vector_class<event>& deps = {});

event ccl::reduce_scatter(const void* send_buf,
                          void* recv_buf,
                          size_t recv_count,
                          datatype dtype,
                          reduction rtype,
                          const communicator& comm,
                          const stream& stream,
                          const reduce_scatter_attr& attr = default_reduce_scatter_
↪attr,
                          const vector_class<event>& deps = {});
```

send_buf the buffer with `comm_size` * `count` elements of `BufferType` that stores local data to be reduced

recv_buf [out] the buffer to store the result block containing `recv_count` elements of type `BufferType`

recv_count the number of elements of type `BufferType` in the received block

rtype the type of the reduction operation to be applied

dtype

the datatype of elements in `send_buf` and `recv_buf`
must be skipped if `BufferType` can be inferred
otherwise must be passed explicitly

comm the communicator that defines a group of ranks for the operation

stream the stream associated with the operation

attr optional attributes to customize the operation

deps an optional vector of the events that the operation should depend on

return `event` an object to track the progress of the operation

Note: See also:

- *Communicator*
- *Stream*
- *Event*
- *Operation Progress Tracking*

Operation Attributes

oneCCL specification defines communication operation attributes that serve as modifiers of an operation's behavior. Optionally, they may be passed to the corresponding communication operations.

oneCCL specification defines the following operation attribute classes:

- `allgatherv_attr`
- `allreduce_attr`
- `alltoallv_attr`
- `barrier_attr`
- `broadcast_attr`
- `reduce_attr`
- `reduce_scatter_attr`

oneCCL specification defines attribute identifiers that may be used to fill operation attribute objects.

The list of common attribute identifiers that may be used for any communication operation:

```
enum class operation_attr_id {
    priority    = /* unspecified */,
    to_cache    = /* unspecified */,
    synchronous = /* unspecified */,
    match_id    = /* unspecified */

    last_value  = /* unspecified, equal to the largest of all the values above */
};
```

operation_attr_id::priority the priority of the communication operation

operation_attr_id::to_cache

persistent/non-persistent communication operation
should be used in conjunction with `match_id`

operation_attr_id::synchronous synchronous/asynchronous communication operation

operation_attr_id::match_id

the unique identifier of the operation
in conjunction with `to_cache`, it enables the caching of the communication operation

The communication operation specific attribute identifiers may extend the list of common identifiers.

The list of attribute identifiers that may be used for *Allreduce*, *Reduce* and *ReduceScatter* collective operations:

```
enum class allreduce_attr_id {
    reduction_fn = /* unspecified */
};

enum class reduce_attr_id {
    reduction_fn = /* unspecified */
};

enum class reduce_scatter_attr_id {
    reduction_fn = /* unspecified */
};
```

allreduce_attr_id::reduction_fn / reduce_attr_id::reduction_fn / reduce_scatter_attr_id::reduction_fn a function pointer for the custom reduction operation that follows the signature:

```
typedef void (*reduction_fn)
(
    const void*,       /* in_buf     */
    size_t,            /* in_count   */
    void*,             /* inout_buf  */
    size_t*,           /* out_count  */
    datatype,          /* datatype   */
    const fn_context* /* context    */
);

typedef struct {
    const char* match_id;
    const size_t offset;
} fn_context;
```

Creating an operation attribute object, which may be used in a corresponding communication operation:

```
template <class OpAttrType>
OpAttrType ccl::create_operation_attr();
```

return `OpAttrType` an object to hold attributes for a specific communication operation

The operation attribute classes shall provide `get` and `set` methods for getting and setting of values with specific attribute identifiers.

Operation Progress Tracking

oneCCL communication operation shall return an event object to be used for tracking the operation's progress.

The `event` class shall provide the ability to wait for completion of an operation in a blocking manner, the ability to check the completion status in a non-blocking manner, and the ability to retrieve the underlying native object that is signaled when the operation completes.

Event

Waiting for the completion of an operation in a blocking manner:

```
void event::wait();
```

Checking for the completion of an operation in a non-blocking manner:

```
bool event::test();
```

return **bool** `true` if the operation has been completed `false` if the operation has not been completed

Retrieving a native object that is signaled when the operation completes:

```
native_event_type event::get_native();
```

return **native_event_type**

a native object that is signaled when the operation completes
shall throw an exception if an `event` object does not wrap the native object

6.4.3 Error handling

oneCCL error handling relies on the mechanism of C++ exceptions. If an error occurs, it shall be propagated at the point of a function call where it is caught using standard C++ error handling mechanism.

Exception classification

Exception classification in oneCCL is aligned with C++ Standard Library classification. oneCCL introduces class that defines the base class in the hierarchy of oneCCL exception classes. All oneCCL routines throw exceptions inherited from this base class.

In the hierarchy of oneCCL exceptions, `ccl::exception` is the base class inherited from `std::exception` class. All other oneCCL exception classes are derived from this base class.

This specification does not require implementations to perform error-checking. However, if an implementation does provide error-checking, it shall use the following exception classes. Additional implementation-specific exception classes can be used for exceptional conditions not fitting any of these classes.

Common exceptions

Exception class	Description
`ccl::exception`	Reports general unspecified error
`ccl::invalid_argument`	Reports an error when arguments to the operation were rejected
`ccl::host_bad_alloc`	Reports an error that occurred during memory allocation on the host
`ccl::unimplemented`	Reports an error when the requested operation is not implemented
`ccl::unsupported`	Reports an error when the requested operation is not supported

6.5 Programming Model

6.5.1 Generic Workflow

Below is a generic workflow with oneCCL API

1. Create a main built-in key-value store. Its address should be distributed using an out-of-band communication mechanism and be used to create key-value stores on other processes:

```
using namespace std;

/* for example use MPI as an out-of-band communication mechanism */

int mpi_rank, mpi_size;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

ccl::shared_ptr_class<ccl::kvs> kvs;
ccl::kvs::address_type kvs_addr;

if (mpi_rank == 0) {
    kvs = ccl::create_main_kvs();
    kvs_addr = kvs->get_address();
    MPI_Bcast((void*)kvs_addr.data(), ccl::kvs::address_max_size, MPI_BYTE, 0, MPI_
↪COMM_WORLD);
}
else {
    MPI_Bcast((void*)kvs_addr.data(), ccl::kvs::address_max_size, MPI_BYTE, 0, MPI_
↪COMM_WORLD);
    kvs = ccl::create_kvs(kvs_addr);
}
```

2. Create communicator(s):

```
/* host communications */
auto comm = ccl::create_communicator(mpi_size, mpi_rank, kvs);
```

```
/* SYCL devices communications, for example with multiple devices per process */

/* sycl_context -> sycl::context */
/* sycl_devices -> vector<sycl::device> */
/* sycl_queues  -> vector<sycl::queue> */

/* create ccl::context object from sycl::context object */
auto ccl_context = ccl::create_context(sycl_context);

/* create ccl::device objects from sycl::device objects */
vector<ccl::device> ccl_devices;
for (size_t idx = 0; idx < sycl_devices.size(); idx++) {
    ccl_devices.push_back(ccl::create_device(sycl_devices[idx]));
}

map<int, ccl::device> r2d_map;
for (auto& dev : ccl_devices) {
```

(continues on next page)

(continued from previous page)

```
    int rank = /* generate a globally unique rank for a specific device */
    r2d_map[rank] = dev;
}

/* create ccl::stream objects from sycl::queue objects */
vector<ccl::stream> ccl_streams;
for (size_t idx = 0; idx < sycl_queues.size(); idx++) {
    ccl_streams.push_back(ccl::create_stream(sycl_queues[idx]));
}

auto comms = ccl::create_communicators(mpi_size * r2d_map.size(),
                                       r2d_map,
                                       ccl_context,
                                       kvs);
```

3. Execute a communication operation of choice on the communicator(s):

```
/* host communications */
allreduce(..., comm).wait();
```

```
/* SYCL devices communications */
vector<ccl::event> events;
for (auto& comm : comms) {
    events.push_back(allreduce(..., comm, ccl_streams[comm.rank()]));
}

for (auto& e : events) {
    e.wait();
}
```

CHAPTER

SEVEN

LEVEL ZERO

The oneAPI Level Zero (Level Zero) provides low-level direct-to-metal interfaces that are tailored to the devices in a oneAPI platform. Level Zero supports broader language features such as function pointers, virtual functions, unified memory, and I/O capabilities while also providing fine-grain explicit controls needed by higher-level runtime APIs including:

- Device discovery
- Memory allocation
- Peer-to-peer communication
- Inter-process sharing
- Kernel submission
- Asynchronous execution and scheduling
- Synchronization primitives
- Metrics reporting

The API architecture exposes both physical and logical abstractions of the underlying oneAPI platform devices and their capabilities. The device, sub-device, and memory are exposed at a physical level while command queues, events, and synchronization methods are defined as logical entities. All logical entities are bound to device-level physical capabilities. The API provides a scheduling model that is tailored to multiple uses including a low-latency submission model to the devices as well as one that is tailored to the construction and submission of work across simultaneous host threads. While heavily influenced by other low-level APIs, such as OpenCL, Level Zero is designed to evolve independently. While heavily influenced by GPU architecture, Level Zero is supportable across different oneAPI compute device architectures, such as FPGAs.

7.1 Detailed API Descriptions

The detailed specification can be found online in the specification.

CHAPTER

EIGHT

ONEDAL

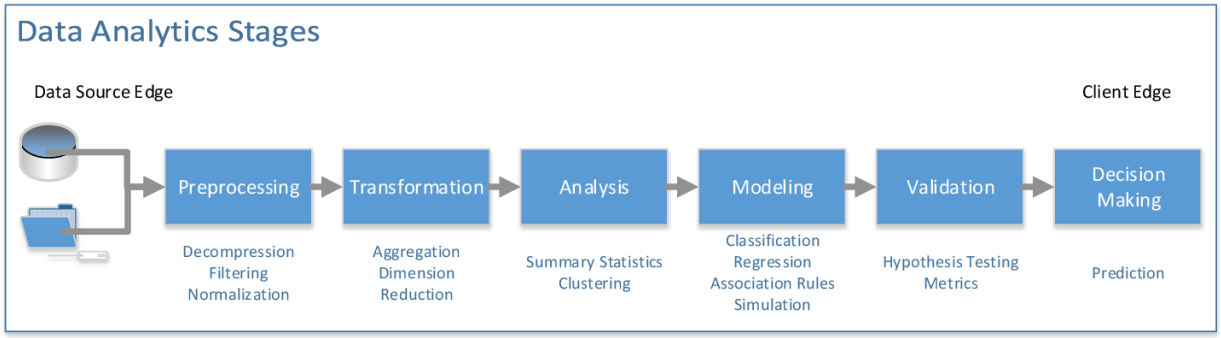
This document specifies requirements for implementations of oneAPI Data Analytics Library (oneDAL).

oneDAL is a library that helps speed up big data analysis by providing highly optimized algorithmic building blocks for all stages of data analytics (preprocessing, transformation, analysis, modeling, validation, and decision making) in batch, online, and distributed processing modes of computation. The current version of oneDAL provides Data Parallel C++ (DPC++) API extensions to the traditional C++ interface.

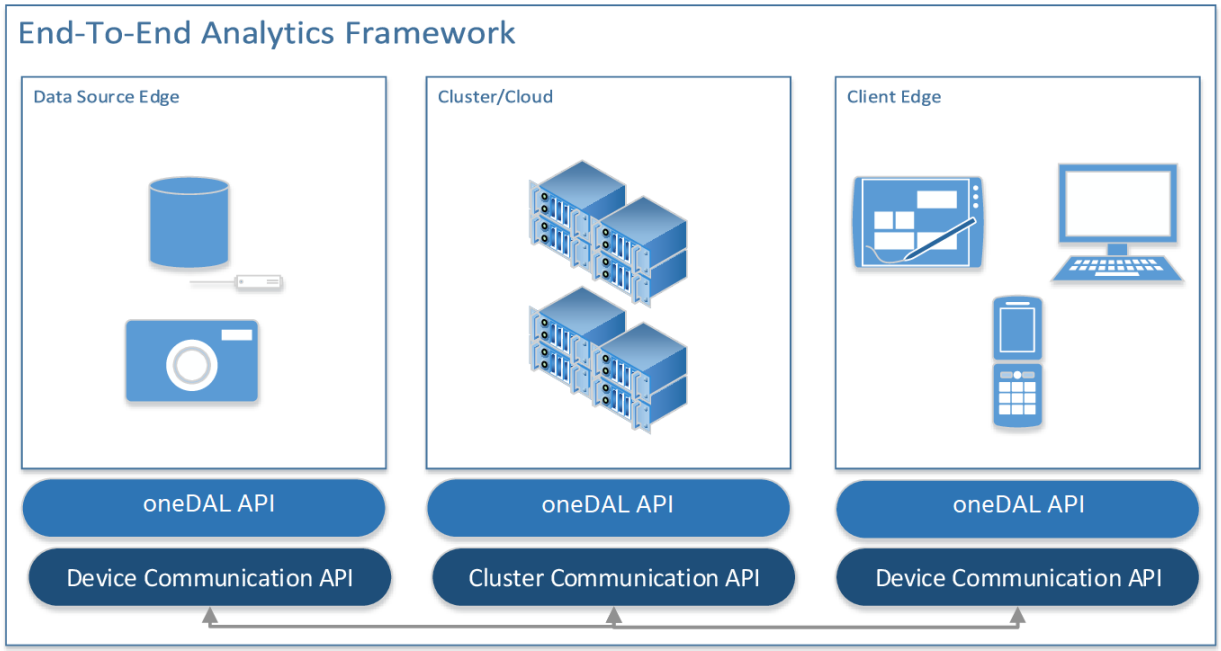
For general information, visit oneDAL GitHub* page.

8.1 Introduction

oneAPI Data Analytics Library (oneDAL) is a library that provides building blocks covering all stages of data analytics: data acquisition from a data source, preprocessing, transformation, data mining, modeling, validation, and decision making.



oneDAL supports the concept of the end-to-end analytics when some of data analytics stages are performed on the edge devices (close to where the data is generated and where it is finally consumed). Specifically, oneDAL Application Programming Interfaces (APIs) are agnostic about a particular cross-device communication technology and, therefore, can be used within different end-to-end analytics frameworks.



oneDAL consists of the following major components:

- The *Data Management* component includes classes and utilities for data acquisition, initial preprocessing and normalization, for data conversion into numeric formats (performed by one of supported Data Sources), and for model representation.
- The *Algorithms* component consists of classes that implement algorithms for data analysis (data mining) and data modeling (training and prediction). These algorithms include clustering, classification, regression, and recommendation algorithms. Algorithms support the following computation modes:
  - *Batch processing*: algorithms work with the entire data set to produce the final result
  - *Online processing*: algorithms process a data set in blocks streamed into the device's memory
  - *Distributed processing*: algorithms operate on a data set distributed across several devices (compute nodes)

    Distributed algorithms in oneDAL are abstracted from underlying cross-device communication technology, which enables use of the library in a variety of multi-device computing and data transfer scenarios.

  Depending on the usage, algorithms operate both on actual data (data set) and data models:
  - Analysis algorithms typically operate on data sets.
  - Training algorithms typically operate on a data set to train an appropriate data model.
  - Prediction algorithms typically work with the trained data model and with a working data set.
- The **Utilities** component includes auxiliary functionality intended to be used for design of classes and implementation of methods such as memory allocators or type traits.
- The **Miscellaneous** component includes functionality intended to be used by oneDAL algorithms and applications for algorithm customization and optimization on various stages of the analytical pipeline. Examples of such algorithms include solvers and random number generators.

Classes in Data Management, Algorithms, Utilities, and Miscellaneous components cover the most important usage scenarios and allow seamless implementation of complex data analytics workflows through direct API calls. At the same time, the library is an object-oriented framework that helps customize the API by redefining particular classes and methods of the library.

8.2 Glossary

8.2.1 Machine learning terms

Categorical feature A *feature* with a discrete domain. Can be *nominal* or *ordinal*.

Synonyms: discrete feature, qualitative feature

Classification A *supervised machine learning problem* of assigning *labels* to *feature vectors*.

Examples: predict what type of object is on the picture (a dog or a cat?), predict whether or not an email is spam

Clustering An *unsupervised machine learning problem* of grouping *feature vectors* into bunches, which are usually encoded as *nominal* values.

Example: find big star clusters in the space images

Continuous feature A *feature* with values in a domain of real numbers. Can be *interval* or *ratio*

Synonyms: quantitative feature, numerical feature

Examples: a person's height, the price of the house

CSV file A comma-separated values file (csv) is a type of a text file. Each line in a CSV file is a record containing fields that are separated by the delimiter. Fields can be of a numerical or a text format. Text usually refers to categorical values. By default, the delimiter is a comma, but, generally, it can be any character. For more details, see.

Dataset A collection of *observations*.

Dimensionality reduction A problem of transforming a set of *feature vectors* from a high-dimensional space into a low-dimensional space while retaining meaningful properties of the original feature vectors.

Feature A particular property or quality of a real object or an event. Has a defined type and domain. In machine learning problems, features are considered as input variable that are independent from each other.

Synonyms: attribute, variable, input variable

Feature vector A vector that encodes information about real object, an event or a group of objects or events. Contains at least one *feature*.

Example: A rectangle can be described by two features: its width and height

Inference A process of applying a *trained model* to the *dataset* in order to predict *response* values based on input *feature vectors*.

Synonym: prediction

Inference set A *dataset* used at the *inference* stage. Usually without *responses*.

Interval feature A *continuous feature* with values that can be compared, added or subtracted, but cannot be multiplied or divided.

Examples: a time frame scale, a temperature in Celsius or Fahrenheit

Label A *response* with *categorical* or *ordinal* values. This is an output in *classification* and *clustering* problems.

Example: the spam-detection problem has a binary label indicating whether the email is spam or not

Model An entity that stores information necessary to run *inference* on a new *dataset*. Typically a result of a *training* process.

Example: in linear regression algorithm, the model contains weight values for each input feature and a single bias value

Nominal feature A *categorical feature* without ordering between values. Only equality operation is defined for nominal features.

Examples: a person's gender, color of a car

Observation A *feature vector* and zero or more *responses*.

Synonyms: instance, sample

Ordinal feature A *categorical feature* with defined operations of equality and ordering between values.

Example: student's grade

Outlier *Observation* which is significantly different from the other observations.

Ratio feature A *continuous feature* with defined operations of equality, comparison, addition, subtraction, multiplication, and division. Zero value element means the absence of any value.

Example: the height of a tower

Regression A *supervised machine learning problem* of assigning *continuous responses* for *feature vectors*.

Example: predict temperature based on weather conditions

Response A property of some real object or event which dependency from *feature vector* need to be defined in *supervised learning* problem. While a *feature* is an input in the machine learning problem, the response is one of the outputs can be made by the *model* on the *inference* stage.

Synonym: dependent variable

Supervised learning *Training* process that uses a *dataset* with information about dependencies between *features* and *responses*. The goal is to get a *model* of dependencies between input *feature vector* and *responses*.

Training A process of creating a *model* based on information extracted from a *training set*. Resulting *model* is selected in accordance with some quality criteria.

Training set A *dataset* used at the *training* stage to create a *model*.

Unsupervised learning *Training* process that uses a *training set* with no *responses*. The goal is to find hidden patters inside *feature vectors* and dependencies between them.

8.2.2 oneDAL terms

Accessor A oneDAL concept for an object that provides access to the data of another object in the special *data format*. It abstracts data access from interface of an object and provides uniform access to the data stored in objects of different types.

Batch mode The computation mode for an algorithm in oneDAL, where all the data needed for computation is available at the start and fits the memory of the device on which the computations are performed.

Builder A oneDAL concept for an object that encapsulates the creation process of another object and enables its iterative creation.

Contiguous data Data that are stored as one contiguous memory block. One of the characteristics of a *data format*.

Data format Representation of the internal structure of the data.

Examples: data can be stored in array-of-structures or compressed-sparse-row format

Data layout A characteristic of *data format* which describes the order of elements in a *contiguous data* block.

Example: row-major format, where elements are stored row by row

Data type An attribute of data used by a compiler to store and access them. Includes size in bytes, encoding principles, and available operations (in terms of a programming language).

Examples: `int32_t`, `float`, `double`

Flat data A block of *contiguous homogeneous* data.

Getter A method that returns the value of the private member variable.

Example:

```
std::int64_t get_row_count() const;
```

Heterogeneous data Data which contain values either of different *data types* or different sets of operations defined on them. One of the characteristics of a *data format*.

Example: A *dataset* with 100 *observations* of three *interval features*. The first two features are of float32 *data type*, while the third one is of float64 data type.

Homogeneous data Data with values of single *data type* and the same set of available operations defined on them. One of the characteristics of a *data format*.

Example: A *dataset* with 100 *observations* of three *interval features*, each of type float32

Immutability The object is immutable if it is not possible to change its state after creation.

Metadata Information about logical and physical structure of an object. All possible combinations of metadata values present the full set of possible objects of a given type. Metadata do not expose information that is not a part of a type definition, e.g. implementation details.

Example: *table* object can contain three *nominal features* with 100 *observations* (logical part of metadata). This object can store data as sparse csr array and provides direct access to them (physical part)

Online mode The computation mode for an algorithm in oneDAL, where the data needed for computation becomes available in parts over time.

Reference-counted object A copy-constructible and copy-assignable oneDAL object which stores the number of references to the unique implementation. Both copy operations defined for this object are lightweight, which means that each time a new object is created, only the number of references is increased. An implementation is automatically freed when the number of references becomes equal to zero.

Setter A method that accepts the only parameter and assigns its value to the private member variable.

Example:

```
void set_row_count(std::int64_t row_count);
```

Table A oneDAL concept for a *dataset* that contains only numerical data, *categorical* or *continuous*. Serves as a transfer of data between user's application and computations inside oneDAL. Hides details of *data format* and generalizes access to the data.

Workload A problem of applying a oneDAL algorithm to a *dataset*.

8.2.3 Common oneAPI terms

API Application Programming Interface

DPC++ Data Parallel C++ (DPC++) is a high-level language designed for data parallel programming productivity. DPC++ is based on *SYCL** from the Khronos* Group to support data parallelism and heterogeneous programming.

Host/Device OpenCL [OpenCLSpec] refers to CPU that controls the connected GPU executing kernels.

JIT Just in Time Compilation — compilation during execution of a program.

Kernel Code written in OpenCL [OpenCLSpec] or *SYCL* and executed on a GPU device.

SPIR-V Standard Portable Intermediate Representation - V is a language for intermediate representation of compute kernels.

SYCL SYCL(TM) [SYCLSpec] — high-level programming model for OpenCL(TM) that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++.

8.3 Mathematical Notations

Notation	Definition
n or m	The number of *observations* in a *dataset*. Typically n is used, but sometimes m is required to distinguish two datasets, e.g., the *training set* and the *inference set*.
p or r	The number of features in a dataset. Typically p is used, but sometimes r is required to distinguish two datasets.
$a \times b$	The dimensionality of a matrix (dataset) has a rows (observations) and b columns (features).
$\|A\|$	Depending on the context may be interpreted as follows: • If A is a set, this denotes its cardinality, i.e., the number of elements in the set A. • If A is a real number, this denotes an absolute value of A.
$\|\|x\|\|$	The L_2-norm of a vector $x \in \mathbb{R}^d$, $$\|\|x\|\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_d^2}.$$
$\operatorname{sgn}(x)$	Sign function for $x \in \mathbb{R}$, $$\operatorname{sgn}(x) = \begin{cases} -1, x < 0, \\ 0, x = 0, \\ 1, x > 0. \end{cases}$$
x_i	In the description of an algorithm, this typically denotes the i-th *feature vector* in the training set.
x'_i	In the description of an algorithm, this typically denotes the i-th feature vector in the inference set.
y_i	In the description of an algorithm, this typically denotes the i-th *response* in the training set.
y'_i	In the description of an algorithm, this typically denotes the i-th response that needs to be predicted by the inference algorithm given the feature vector x'_i from the inference set.

8.4 Programming model

oneDAL primarily targets algorithms that are extensively used in data analytics. These algorithms typically have many parameters, i.e. knobs to control its internal behavior and produced result. In machine learning, those parameters are often referred as *meta-parameters* to distinguish them from the model parameters learned during the training. Some algorithms define a dozen meta-parameters, while others depend on another algorithm as, for example, the logistic regression training procedure depends on an optimization algorithm.

Besides meta-parameters, machine learning algorithms may have different *stages*, such as *training* and *inference*. Moreover, the stages of an algorithm may be implemented in a variety of *computational methods*. For instance, a linear regression model could be trained by solving a system of linear equations [Friedman17] or by applying an iterative optimization solver directly to the empirical risk function [Zhang04].

The same machine learning techniques are often applied for solving problems of different types. In the example with linear regression, the same mathematical model used for solving *regression* problem is generalized for solving a *classification* problem, for example, logistic regression. Such techniques differ only in few problem-specific aspects, but share the same subset of meta-parameters and have a common computational flow. oneDAL does not distinguish these techniques into different algorithms. Instead, from oneDAL perspective, the same algorithm may perform different *computational tasks*.

From computational perspective, algorithm implementation may rely on different *floating-point types*, such as `float`, `double` or `bfloat16`. Having a capability to specify what type is needed is important for the end user as their precision requirements vary depending on a workload.

To best tackle the mentioned challenges, each algorithm is decomposed into *descriptors* and *operations*.

8.4.1 End-to-end example

Below you can find a typical workflow of using oneDAL algorithm on GPU. The example is provided for *Principal Component Analysis algorithm (PCA)*.

The following steps depict how to:

- Read the data from CSV file
- Run the training and inference operations for PCA
- Access intermediate results obtained at the training stage

1. Include the following header that makes all oneDAL declarations available.

```
#include "oneapi/dal.hpp"

/* Standard library headers required by this example */
#include <cassert>
#include <iostream>
```

2. Create a SYCL* queue with the desired device selector. In this case, GPU selector is used:

```
const auto queue = sycl::queue{ sycl::gpu_selector{} };
```

3. Since all oneDAL declarations are in the `oneapi::dal` namespace, import all declarations from the `oneapi` namespace to use `dal` instead of `oneapi::dal` for brevity:

```
using namespace oneapi;
```

4. Use *CSV data source* to read the data from the CSV file into a *table*:

```
const auto data = dal::read<dal::table>(queue, dal::csv::data_source{"data.csv"});
```

5. Create a *PCA* descriptor, configure its parameters, and run the training algorithm on the data loaded from CSV.

```
const auto pca_desc = dal::pca::descriptor<float>
   .set_component_count(3)
   .set_deterministic(true);

const dal::pca::train_result train_res = dal::train(queue, pca_desc, data);
```

6. Print the learned eigenvectors:

```
const dal::table eigenvectors = train_res.get_eigenvectors();

const auto acc = dal::row_accessor<const float>{eigenvectors};
for (std::int64_t i = 0; i < eigenvectors.row_count(); i++) {

   /* Get i-th row from the table, the eigenvector stores pointer to USM */
   const dal::array<float> eigenvector = acc.pull(queue, {i, i + 1});
   assert(eigenvector.get_count() == eigenvectors.get_column_count());

   std::cout << i << "-th eigenvector: ";
   for (std::int64_t j = 0; j < eigenvector.get_count(); j++) {
      std::cout << eigenvector[j] << " ";
   }
   std::cout << std::endl;
}
```

7. Use the trained model for inference to reduce dimensionality of the data:

```
const dal::pca::model model = train_res.get_model();

const dal::table data_transformed =
   dal::infer(queue, pca_desc, data).get_transformed_data();

assert(data_transformed.column_count() == 3);
```

8.4.2 Descriptors

A descriptor is an object that represents an algorithm including all its meta-parameters, dependencies on other algorithms, floating-point types, computational methods and tasks. A descriptor serves as:

- A dispatching mechanism for *operations*. Based on a descriptor type, an operation executes a particular algorithm implementation.
- An aggregator of meta-parameters. It provides an interface for setting up meta-parameters at either compile-time or run-time.
- An object that stores the state of the algorithm. In the general case, a descriptor is a stateful object whose state changes after an operation is applied.

Each oneDAL algorithm has its own dedicated namespace, where the corresponding descriptor is defined (for more details, see *Namespaces*). Descriptor, in its turn, defines the following:

- **Template parameters.** A descriptor is allowed to have any number of template parameters, but shall support at least three:

  - `Float` is a *floating-point type* that the algorithm uses for computations. This parameter is defined first and has the `oneapi::dal::default_float_t` default value.
  - `Method` is a tag-type that specifies the *computational method*. This parameter is defined second and has the `method::by_default` default value.
  - `Task` is a tag-type that specifies the *computational task*. This parameter is defined third and has the `task::by_default` default value.

- **Properties.** A property is a run-time parameter that can be accessed by means of the corresponding *getter* and *setter* methods.

The following code sample shows the common structure of a descriptor's definition for an abstract algorithm. To define a particular algorithm, the following strings shall be substituted:

- `%ALGORITHM%` is the name of an algorithm and its namespace. All classes and structures related to that algorithm are defined within the namespace.
- `%PROPERTY_NAME%` and `%PROPERTY_TYPE%` are the name and the type of one of the algorithm's properties.

```
namespace oneapi::dal::%ALGORITHM% {

template <typename Float  = default_float_t,
          typename Method = method::by_default,
          typename Task   = task::by_default,
          /* more template parameters */>
class descriptor {
public:
   /* Constructor */
   descriptor(const %PROPERTY_TYPE%& %PROPERTY_NAME%,
              /* more properties */)

   /* Getter & Setter for the property called `%PROPERTY_NAME%` */
   descriptor& set_%PROPERTY_NAME%(%PROPERTY_TYPE% value);
   %PROPERTY_TYPE% get_%PROPERTY_NAME%() const;

   /* more properties */
};

} // namespace oneapi::dal::%ALGORITHM%
```

Each meta-parameter of an algorithm is mapped to a property that shall satisfy the following requirements:

- Properties are defined with getter and setter methods. The underlying class member variable that stores the property's value is never exposed in the descriptor interface.
- The getter returns the value of the underlying class member variable.
- The setter accepts only one parameter of the property's type and assigns it to the underlying class member variable.
- Most of the properties are preset with default values, others are initialized by passing the required parameters to the constructor.
- The setter returns a reference to the descriptor object to allow chaining calls as shown in the example below.

```
auto desc = descriptor{}
   .set_property_name_1(value_1)
   .set_property_name_2(value_2)
   .set_property_name_3(value_3);
```

Floating-point Types

It is required for each algorithm to support at least one implementation-defined floating-point type. Other floating-point types are optional, for example `float`, `double`, `float16`, and `bfloat16`. It is up to a specific oneDAL implementation whether or not to support these types.

The floating-point type used as a default in descriptors is implementation-defined and shall be declared within the top-level namespace.

```
namespace oneapi::dal {
   using default_float_t = /* implementation defined */;
} // namespace oneapi::dal
```

Computational Methods

The supported computational methods are declared within the `%ALGORITHM%::method` namespace using tag-types. Algorithm shall support at least one method and declare the `by_default` type alias that refers to one of the methods as shown in the example below.

```
namespace oneapi::dal::%ALGORITHM% {
   namespace method {
      struct x {};
      struct y {};
      using by_default = x;
   } // namespace method
} // namespace oneapi::dal::%ALGORITHM%
```

Computational Tasks

The supported computational tasks are declared within the `%ALGORITHM%::task` namespace using tag-types. Algorithm shall support at least one task and declare the `by_default` type alias that refers to one of the tasks as shown in the example below.

If an algorithm assumes both `classification` and `regression` tasks, the default task shall be `classification`. In some cases where an algorithm does not have the well-defined training and inference stages an algorithm may define only one task.

```
namespace oneapi::dal::%ALGORITHM% {
   namespace task {
      struct classification {};
      struct regression {};
      using by_default = classification;
   } // namespace task
} // namespace oneapi::dal::%ALGORITHM%
```

8.4.3 Operations

An operation is a function that transforms *a descriptor* and other arguments represented via *an input* object to *a result* object. An operation is responsible for:

- Executing all of an algorithm's computational routines represented by the descriptor.
- Passing SYCL* queue to computational routines.
- Verifying preconditions and postconditions before and after the execution of computational routines.

General operation definition

The following code sample shows the declaration of an abstract operation. To declare a particular operation, the `%OPERATION%` shall be substituted with the name of the operation.

```
namespace oneapi::dal {

template <typename Descriptor>
using %OPERATION%_input_t = /* implementation defined */;

template <typename Descriptor>
using %OPERATION%_result_t = /* implementation defined */;

template <typename Descriptor>
%OPERATION%_result_t<Descriptor> %OPERATION%(
   sycl::queue& queue,
   const Descriptor& desc,
   const %OPERATION%_input_t<Descriptor>& input);

} // namespace oneapi::dal
```

Each operation shall satisfy the following requirements:

- An operation shall accept three parameters in the following order:
  - The SYCL* queue object
  - The descriptor of the algorithm
  - The *input object*
- An operation shall return the *result object*.
- The `%OPERATION%_input_t` and `%OPERATION%_result_t` alias templates shall be used for inference of the input and return types.
- If a precondition is violated, an operation shall throw an exception derived from `oneapi::dal::logic_error`.
- If a postcondition is violated, an operation shall throw an exception derived from `oneapi::dal::runtime_error`.
- If the descriptor is incompatible with some operation, an error shall be reported at compile-time.
- The exact list of compatible operations and pre-/post- conditions shall be defined by *a particular algorithm specification*.

Operation shortcuts

In order to make the code on user side less verbose, oneDAL defines the following overloaded functions called *shortcuts* for each operation in addition to the general one described in section *General operation definition*.

- A shortcut for execution on *host* that performs the same operation as the general function on host, but does not require the queue to be passed explicitly.

```
template <typename Descriptor>
%OPERATION%_result_t<Descriptor> %OPERATION%(
   const Descriptor& desc,
   const %OPERATION%_input_t<Descriptor>& input);
```

- A shortcut that allows omitting explicit input creation.

```
template <typename Descriptor, typename... Args>
%OPERATION%_result_t<Descriptor> %OPERATION%(
   sycl::queue& queue,
   const Descriptor& desc,
   Args&&... args);
```

- A shortcut that allows omitting explicit queue and input creation. This is a combination of two previous shortcuts.

```
template <typename Descriptor, typename... Args>
%OPERATION%_result_t<Descriptor> %OPERATION%(
   const Descriptor& desc,
   Args&&... args);
```

Input

An input object aggregates all the data that the algorithm requires for performing a specific operation. The data is represented via *tables*, so, typically, an input is a collection of tables, but not limited to them and can aggregate objects of an arbitrary type.

In general, input class definition is similar to *descriptor*. An input defines properties that can be accessed by means of the corresponding *getter* and *setter* methods. Requirements to the input's properties are the same as *requirements for descriptor's properties*.

The following code sample shows the common structure of a inputs's definition. To define an input for particular algorithm and operation, the following strings shall be substituted:

- `%ALGORITHM%` is the name of an algorithm and its namespace.
- `%OPERATION%` is the name of operation.
- `%PROPERTY_NAME%` and `%PROPERTY_TYPE%` are the name and the type of one of the input's properties.

```
namespace oneapi::dal::%ALGORITHM% {

template <typename Task = task::by_default>
class OPERATION_input {
public:
   /* Constructor */
   %OPERATION%_input(const %PROPERTY_TYPE%& %PROPERTY_NAME%,
                     /* more properties */)

   /* Getter & Setter for the property called `%PROPERTY_NAME%` */
```

(continues on next page)

(continued from previous page)

```
   descriptor& set_%PROPERTY_NAME%(%PROPERTY_TYPE% value);
   %PROPERTY_TYPE% get_%PROPERTY_NAME%() const;

   /* more properties */
};

} // namespace oneapi::dal::%ALGORITHM%
```

Note: An input is specific to algorithm and operation, so each `%ALGORITHM%-%OPERATION%` pair shall define its own set of the properties.

Result

A result object aggregates all output values computed by the algorithm. All assumptions about *an input* are applied to a result as well.

```
namespace oneapi::dal::%ALGORITHM% {

template <typename Task = task::by_default>
class OPERATION_result {
public:
   /* Constructor */
   %OPERATION%_result(const %PROPERTY_TYPE%& %PROPERTY_NAME%,
                      /* more properties */)

   /* Getter & Setter for the property called `%PROPERTY_NAME%` */
   descriptor& set_%PROPERTY_NAME%(%PROPERTY_TYPE% value);
   %PROPERTY_TYPE% get_%PROPERTY_NAME%() const;

   /* more properties */
};

} // namespace oneapi::dal::%ALGORITHM%
```

Supported operation

Refer to the *Supported operations* section for more information about particular operations.

Supported operations

This section describes all operations supported by oneDAL. For more information about general operation definition, refer to *Operations* section.

The table bellow specifies whether an algorithm's descriptor can be used together with each operation.

Algorithm	Operations		
	Train	*Infer*	*Compute*
K-Means	Yes	Yes	No
K-Means Initialization	No	No	Yes
k-NN	Yes	Yes	No
PCA	Yes	Yes	No

Train

The `train` operation performs *training* procedure of a machine learning algorithm. The result obtained after the training contains a *model* that can be passed to the `infer` operation.

```
namespace oneapi::dal {

template <typename Descriptor>
using train_input_t = /* implementation defined */;

template <typename Descriptor>
using train_result_t = /* implementation defined */;

template <typename Descriptor>
train_result_t<Descriptor> train(
   sycl::queue& queue,
   const Descriptor& desc,
   const train_input_t<Descriptor>& input);

} // namespace oneapi::dal
```

Infer

The `infer` operation performs *inference* procedure of a machine learning algorithm based on the model obtained as a result of training.

```
namespace oneapi::dal {

template <typename Descriptor>
using infer_input_t = /* implementation defined */;

template <typename Descriptor>
using infer_result_t = /* implementation defined */;

template <typename Descriptor>
infer_result_t<Descriptor> infer(
   sycl::queue& queue,
   const Descriptor& desc,
   const infer_input_t<Descriptor>& input);

} // namespace oneapi::dal
```

Compute

The `compute` operation is used if an algorithm does not have the well-defined training and inference stages.

```
namespace oneapi::dal {

template <typename Descriptor>
using compute_input_t = /* implementation defined */;

template <typename Descriptor>
using compute_result_t = /* implementation defined */;

template <typename Descriptor>
compute_result_t<Descriptor> compute(
   sycl::queue& queue,
   const Descriptor& desc,
   const compute_input_t<Descriptor>& input);

} // namespace oneapi::dal
```

8.4.4 Computational modes

Batch

In the batch processing mode, the algorithm works with the entire data set to produce the final result. A more complex scenario occurs when the entire data set is not available at the moment or the data set does not fit into the device memory.

Online

In the online processing mode, the algorithm processes a data set in blocks streamed into the device's memory. Partial results are updated incrementally and finalized when the last data block is processed.

Distributed

In the distributed processing mode, the algorithm operates on a data set distributed across several devices (compute nodes). On each node, the algorithm produces partial results that are later merged into the final result on the main node.

8.5 Common Interface

8.5.1 Current Version of this oneDAL Specification

This is the oneDAL specification which is part of the oneAPI specification version 1.0.

8.5.2 Header files

oneDAL public identifiers are represented in the following header files:

Header file	Description
`oneapi/dal.hpp`	The main header file of oneDAL library.
`oneapi/dal/%FILE%.hpp`	The common type definitions used in other oneDAL layers. For example, `data_type` or `range`.
`oneapi/dal/algo/%ALGO%.hpp`	A header file for a particular algorithm. The folder for the algorithm itself is `oneapi/dal/algo/%ALGO%/`. The string %ALGO% should be substituted with the name of the algorithm, for example, `kmeans` or `knn`.
`oneapi/dal/algo/misc/%FUNC%.hpp`	A header file for miscellaneous data types and functionality that is intended to be used by oneDAL algorithms and applications of the analytical pipeline. The string %FUNC% should be substituted with the name of the functionality, for example, `mt19937` or `cross_enthropy_loss`.
`oneapi/dal/table/%FILE%.hpp`	A header file for the types related to the *table* concept. The string %FILE% should be substituted with the name of the functionality, for example, `common` for key concepts related to table types (e.g., `table`, `table_metadata`, `data_layout` classes). For entities that have the `_table` suffix in their names, the related header file shall not contain this suffix in its name, for example, `homogen` for `homogen_table` class.
`oneapi/dal/io/%FILE%.hpp`	A header file for the types and entities of input-output functionality. The string %FILE% should be substituted with the name of the functionality, for example, `csv` for reading and writing csv files.
`oneapi/dal/util/%UTIL%.hpp``	A header file for auxiliary functionality, such as memory allocators or type traits, that is intended to be used for the design of classes and implementation of various methods. The string %UTIL% should be substituted with the name of the auxiliary functionality, for example, `usm_allocator` or `type_traits`.

8.5.3 Namespaces

oneDAL functionality is represented with a system of C++ namespaces described below:

Namespace	oneDAL content
`oneapi::dal`	The namespace of the library that contains externally visible data types, data management entities, processing and service functionality of oneDAL.
`oneapi::dal::ALGORITHM`	The namespace of the algorithm. All classes and structures related to that algorithm shall be defined within this particular namespace. To define a namespace for a specific algorithm, the string %ALGORITHM% should be substituted with its name, for example, `oneapi::dal::kmeans` or `oneapi::dal::knn`.
`oneapi::dal::DATA_SOURCE`	The namespace of the data source. All classes and structures related to that data source shall be defined within a particular namespace. To define a specific data source, the string %DATA_SOURCE% should be substituted with its name, for example, `oneapi::dal::csv`.
`oneapi::dal::inner`	The namespace that contains miscellaneous data types and functionality intended to be used by oneDAL algorithms and applications for algorithm customization and optimization on various stages of the analytical pipeline.
`%PARENT%::detail`	The namespace that contains implementation details of the data types and functionality for the parent namespace. The namespace can be on any level in the namespace hierarchy. To define a specific namespace, the string %PARENT% should be substituted with the namespace for which the details are provided, for example, `oneapi::dal::detail` or `oneapi::dal::kmeans::detail`. The application shall not use any data types nor call any functionality located in the `detail` namespaces.

8.5.4 Error handling

oneDAL error handling relies on the mechanism of C++ exceptions. If an error occurs, it shall be propagated at the point of a function call where it is caught using standard C++ error handling mechanism.

Exception classification

Exception classification in oneDAL is aligned with C++ Standard Library classification. oneDAL shall introduce abstract classes that define the base class in the hierarchy of exception classes. Non-abstract exception classes are derived from the respective C++ Standard Library exception classes. oneDAL shall throw exceptions represented with non-abstract classes.

In the hierarchy of oneDAL exceptions, `oneapi::dal::exception` is the base abstract class that all other exception classes are derived from.

```
class oneapi::dal::exception;
```

Exception	Description	Abstract
`oneapi::dal::exception`	The base class of oneDAL exception hierarchy.	Yes

All oneDAL exceptions shall be divided into three groups:

- logic errors
- runtime errors
- errors with allocation

```
class oneapi::dal::logic_error : public oneapi::dal::exception;
class oneapi::dal::runtime_error : public oneapi::dal::exception;
class oneapi::dal::bad_alloc :  public oneapi::dal::exception, public std::bad_alloc;
```

Exception	Description	Abstract
`oneapi::dal::logic_error`	Reports violations of preconditions and invariants.	Yes
`oneapi::dal::runtime_error`	Reports violations of postconditions and other errors happened during the execution of oneDAL functionality.	Yes
`oneapi::dal::bad_alloc`	Reports failure to allocate storage.	Yes

All precondition and invariant errors represented by `oneapi::dal::logic_error` shall be divided into the following groups:

- invalid argument errors
- domain errors
- out of range errors
- errors with an unimplemented method or algorithm
- unsupported device

```
class oneapi::dal::invalid_argument :  public oneapi::dal::logic_error, public␣
↪std::invalid_argument;
class oneapi::dal::domain_error :  public oneapi::dal::logic_error, public␣
↪std::domain_error;
class oneapi::dal::out_of_range :  public oneapi::dal::logic_error, public std::out_
↪of_range;
class oneapi::dal::unimplemented :  public oneapi::dal::logic_error, public␣
↪std::logic_error;
class oneapi::dal::unsupported_device :  public oneapi::dal::logic_error, public␣
↪std::logic_error;
```

Exception	Description	Abstract
`oneapi::dal::invalid_argument`	Reports situations when the argument was not accepted.	No
`oneapi::dal::domain_error`	Reports situations when the argument is outside of the domain on which the operation is defined. Higher priority than `oneapi::dal::invalid_argument`.	No
`oneapi::dal::out_of_range`	Reports situations when the index is out of range. Higher priority than `oneapi::dal::invalid_argument`.	No
`oneapi::dal::unimplemented`	Reports errors that arise because an algorithm or a method is not implemented.	No
`oneapi::dal::unsupported_device`	Reports situations when a device is not supported.	No

Errors that occur during the execution of oneDAL functionality are represented with `oneapi::dal::runtime_error`. Two main groups of errors shall be distinguished:

- errors in the destination type range
- errors in the OS facilities interaction

All other errors are reported via `oneapi::dal::internal_error`.

```
class oneapi::dal::range_error :  public oneapi::dal::runtime_error, public␣
↪std::range_error;
class oneapi::dal::system_error :  public oneapi::dal::runtime_error, public␣
↪std::system_error;
class oneapi::dal::internal_error :  public oneapi::dal::runtime_error, public␣
↪std::runtime_error;
```

Exception	Description	Abstract
`oneapi::dal::range_error`	Reports situations where a result of a computation cannot be represented by the destination type.	No
`oneapi::dal::system_error`	Reports errors occurred during interaction with OS facilities.	No
`oneapi::dal::internal_error`	Reports all runtime errors that could not be assigned to other inheritors.	No

All memory allocation errors are represented by `oneapi::dal::bad_alloc`. They shall be divided into two groups based on where they occur:

- Host memory allocation error
- Device memory allocation error

```
class oneapi::dal::host_bad_alloc :  public oneapi::dal::bad_alloc;
class oneapi::dal::device_bad_alloc :  public oneapi::dal::bad_alloc;
```

Exception	Description	Abstract
`oneapi::dal::host_bad_alloc`	Reports failure to allocate storage on the host.	No
`oneapi::dal::device_bad_alloc`	Reports failure to allocate storage on the device.	No

8.5.5 Common type definitions

This section describes common types used in oneDAL.

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/common.hpp` header file.

Scalar types

oneDAL relies on the use of integral types defined in `<cstdint>`. This file shall be included in `oneapi/dal/common.hpp` and all oneDAL types shall use these data types.

The interfaces of the library shall use `std::int64_t` data type to represent dimensionality (for example, the number of rows and columns in the table).

It is recommended to use standard C++ types for applications as well.

Enum classes

Which base type to use when defining `enum` or `enum class` representing a oneDAL concept is up to the implementer unless specification requires a specific base type.

Data type

The implementation of *data type* concept. It shall enumerate all the data types supported by oneDAL to perform computations. The `data_type` class shall contain all the base *scalar types* and can also extend them. Base scalar types include the types whose names follow the pattern `std::int_XX_t` or `std::uint_XX_t`, where XX is 8, 16, 32, or 64.

```
enum class data_type {
    int8,
    int16,
    int32,
    int64,
    uint8,
    uint16,
    uint32,
    uint64,
    float32,
    float64,
    bfloat16
};
```

enum class data_type

data_type::int8 8-bit signed integer value type.

data_type::int16 16-bit signed integer value type.

data_type::int32 32-bit signed integer value type.

data_type::int64 64-bit signed integer value type.

data_type::uint8 8-bit unsigned integer value type.

data_type::uint16 16-bit unsigned integer value type.

data_type::uint32 32-bit unsigned integer value type.

data_type::uint64 64-bit unsigned integer value type.

data_type::float32 32-bit floating-point value type.

data_type::float64 64-bit floating-point value type.

data_type::bfloat16 bi-float value type.

Range

A range `[start_index, end_index)` in an array or any other container that supports value indexing.

```
struct range {
public:
    range(std::int64_t start, std::int64_t end);

    std::int64_t get_element_count(std::int64_t max_end_index) const noexcept;

    std::int64_t start_idx;

    std::int64_t end_idx;
};
```

struct range

Constructors

range (std::int64_t *start*, std::int64_t *end*)

Constructs a range of elements from the given start and end indices.

Parameters

- **start** – The first index in the range. The value shall be greater than or equal to 0.
- **end** – The relative end index in the range. Indicates the next index after the last one in the range. If positive, shall be greater than *start*. If negative, indicates the offset of the last element from the end of the range. For example, `start = 1` and `end = -2` specify the range of elements [1, 2, 3] in the set [0, 1, 2, 3, 4].

Properties

std::int64_t **element_count**

The number of elements in the range. The *max_end_index* value specifies the last maximal index in the sequence.

Getter & Setter

```
std::int64_t get_element_count(std::int64_t max_end_index) const
noexcept
```

8.6 Data management

This section includes concepts and objects that operate on data. For oneDAL, such set of operations, or **data management**, is distributed between different stages of the *data analytics pipeline*. From a perspective of data management, this pipeline contains three main steps of data acquisition, preparation, and computation (see *the picture below*):

1. Raw data acquisition

- Transfer out-of-memory data from various sources (databases, files, remote storage) into an in-memory representation.

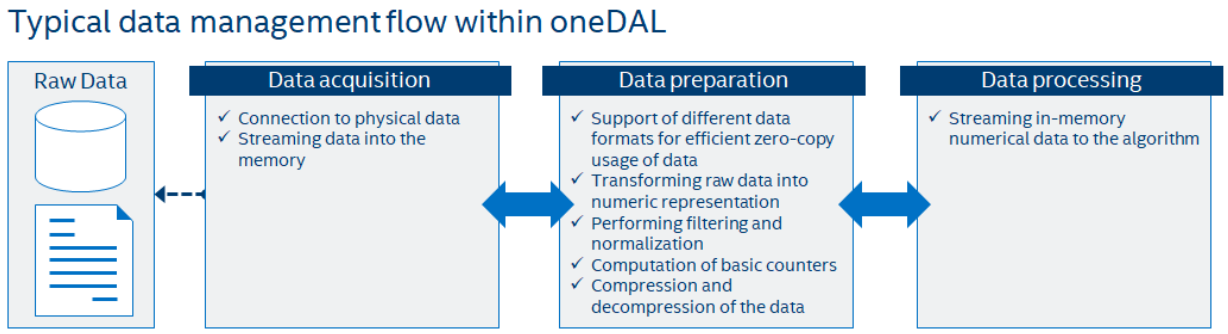
2. Data preparation

- Support different in-memory *data formats*.
- Compress and decompress the data.
- Convert the data into numeric representation.
- Recover missing values.
- Filter the data and perform data normalization.
- Compute various statistical metrics for numerical data, such as mean, variance, and covariance.

3. Algorithm computation

- Stream in-memory numerical data to the algorithm.

In complex usage scenarios, data flow goes through these three stages back and forth. For example, when the data are not fully available at the start of the computation, it can be done step-by-step using blocks of data. After the computation on the current block is completed, the next block should be obtained and prepared.

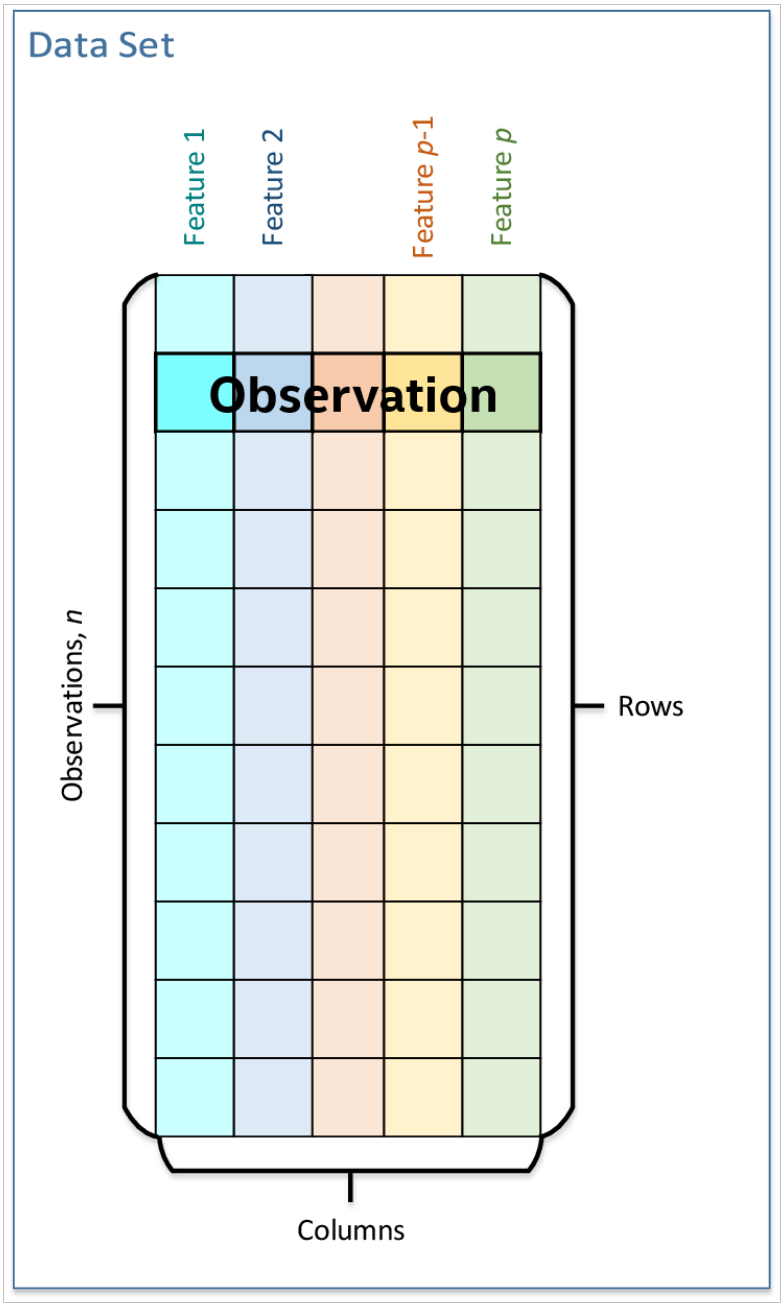


8.6.1 Key concepts

oneDAL provides a set of concepts to operate on out-of-memory and in-memory data during different stages of the *data analytics pipeline*.

Dataset

The main data-related concept that oneDAL works with is a *dataset*. It is a tabular view of data, where table rows represent the *observations* and columns represent the *features*.



The dataset is used across all stages of the data analytics pipeline. For example:

1. At the acquisition stage, it is downloaded into the local memory.
2. At the preparation stage, it is converted into a numerical representation.
3. At the computation stage, it is used as one of the *inputs* or *results* of an algorithm or a *descriptor* properties.

Data source

Data source is a concept of an out-of-memory storage for a *dataset*. It is used at the data acquisition and data preparation stages to:

- Extract datasets from external sources such as databases, files, remote storage.
- Load datasets into the device's local memory. Data do not always fit the local memory, especially when processing with accelerators. A data source provides the ability to load data by batches and extracts it directly into the device's local memory. Therefore, a data source enables complex data analytics scenarios, such as *online computations*.
- Transform datasets into their numerical representation. Data source shall automatically transform non-numeric *categorical* and *continuous* data values into one of the numeric *data formats*.

For details, see *data sources* section.

Table

Table is a concept of in-memory numerical data that are organized in a tabular view with several rows and columns. It is used at the data preparation and data processing stages to:

- Be an in-memory representation of a *dataset* or another tabular data (for example, matrices, vectors, and scalars).
- Store heterogeneous data in various *data formats*, such as dense, sparse, chunked, contiguous.
- Avoid unnecessary data copies during conversion from external data representations.
- Transfer memory ownership of the data from user application to the table, or share it between them.
- Connect with the *data source* to convert data from an out-of-memory into an in-memory representation.
- Support streaming of the data to the algorithm.
- Access the underlying data on a device in a required *data format*, e.g. by blocks with the defined *data layout*.

Note: For thread-safety reasons and better integration with external entities, a table provides a read-only access to the data within it, thus, table object shall be *immutable*.

This concept has different logical organization and physical *format of the data*:

- Logically, a table contains n rows and p columns. Every column may have its own type of data values and a set of allowed operations.
- Physically, a table can be organized in different ways: as a *homogeneous*, *contiguous* array of bytes, as a *heterogeneous* list of arrays of different *data types*, in a compressed-sparse-row format. The number of bytes needed to store the data differs from the number of elements $n \times p$ within a table.

For details, see *tables* section.

Table metadata

Table metadata concept provides an additional information about data in the table:

1. The *data types* of the columns.
2. The logical types of data in the columns: *nominal*, *ordinal*, *interval*, or *ratio*.

Only the properties of data that do not affect table concept definition shall be the part of metadata concept.

> **Warning:** While extending the table concept, specification implementer shall distinguish whether a new property they are adding is a property of a particular `table` sub-type or a property of table metadata.
>
> For example, *data layout* and *data format* are properties of table objects since they affect the structure of a table, its contract, and behavior. The list of names of features or columns inside the table is the example of metadata property.

Accessor

Accessor is a concept that defines a single way to extract the data from a *table*. It allows to:

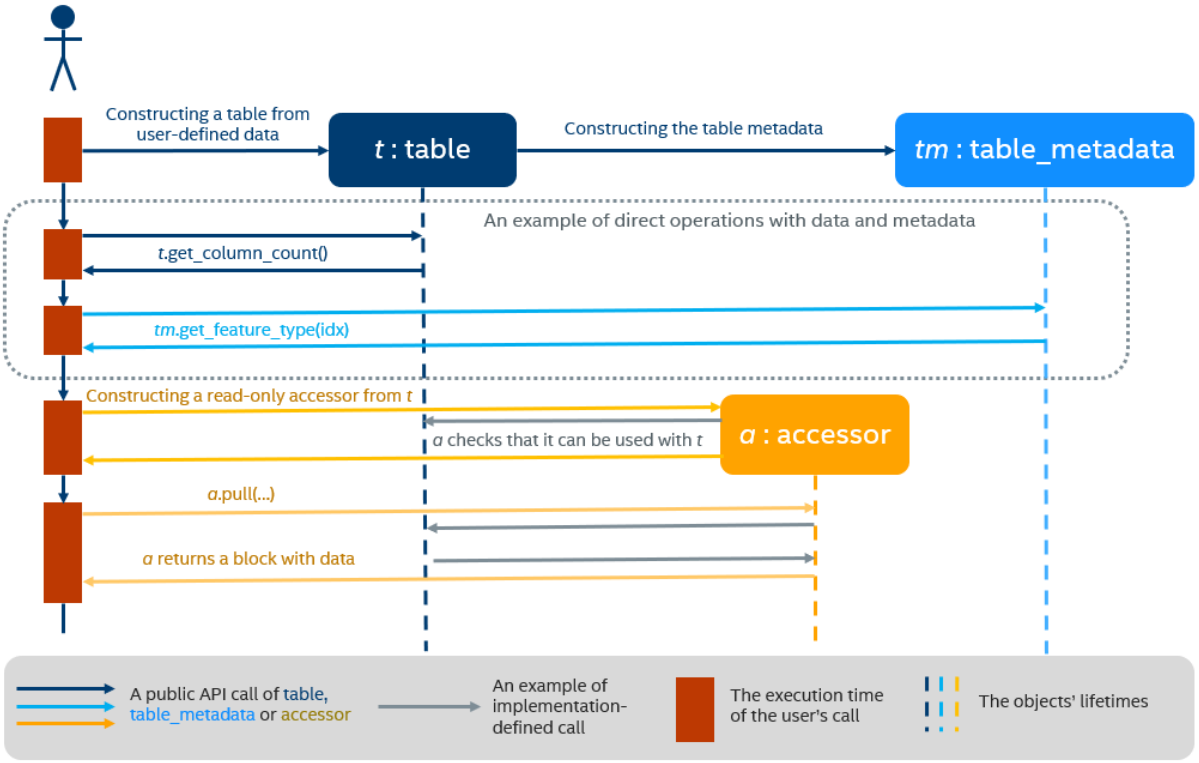
- Have unified access to the data from *table* objects of different types, without exposing their implementation details.
- Provide a *flat* view on the data blocks of a *table* for better data locality. For example, the accessor returns a column of the table stored in row-major format as a contiguous array.
- Acquire data in a desired *data format* for which a specific set of operations is defined.
- Have read-only access to the data.

For details, see *accessors* section.

Example of interaction between table and accessor objects

This section provides a basic usage scenario of the *table* and *accessor* concepts and demonstrates the relations between them. *The following diagram* shows objects of these concepts, which are highlighted by colors:

- *table* object is dark blue
- *accessor* is orange
- *table metadata* is light blue



To perform computations on a dataset, one shall create a *table* object first. It can be done either using a *data source* or directly from user-defined memory. The diagram shows the creation of a *table* object *t* from the data provided by user (not shown on the diagram). During a table creation, an object *tm* of table metadata is constructed and initialized using the data.

Once a table object is created, it can be used as an input in computations or as a parameter of some algorithm. The data in the table can be accessed via its own interface or via read-only accessor as shown on the diagram.

8.6.2 Details

This section includes the detailed descriptions of all data management objects in oneDAL.

Array

The array is a simple concept over the data in oneDAL. It represents a storage that:

1. Holds the data allocated inside it or references to the external data. The data are organized as one *homogeneous* and *contiguous* memory block.
2. Contains information about the memory block's size.
3. Supports both *immutable* and mutable data.
4. Provides an ability to change the data state from immutable to mutable one.
5. Holds ownership information on the data (see the *data ownership requirements* section).
6. Ownership information on the data can be shared between several arrays. It is possible to create a new array from another one without any data copies.

Usage example

The following listing provides a brief introduction to the array API and an example of basic usage scenario:

```
#include <CL/sycl.hpp>
#include <iostream>
#include <string>
#include "oneapi/dal/array.hpp"

using namespace oneapi;

void print_property(const std::string& description, const auto& property) {
   std::cout << description << ": " << property << std::endl;
}

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr std::int64_t data_count = 4;
   const float data[] = { 1.0f, 2.0f, 3.0f, 4.0f };

   // Creating an array from immutable user-defined memory
   auto arr_data = dal::array<float>::wrap(data, data_count);

   // Creating an array from internally allocated memory filled by ones
   auto arr_ones = dal::array<float>::full(queue, data_count, 1.0f);

   print_property("Is arr_data mutable", arr_data.has_mutable_data()); // false
   print_property("Is arr_ones mutable", arr_ones.has_mutable_data()); // true

   // Creating new array from arr_data without data copy - they share ownership␣
↪information.
   dal::array<float> arr_mdata = arr_data;

   print_property("arr_mdata elements count", arr_mdata.get_count()); // equal to␣
↪data_count
   print_property("Is arr_mdata mutable", arr_mdata.has_mutable_data()); // false

   /// Copying data inside arr_mdata to new mutable memory block.
   /// arr_data still refers to the original data pointer.
   arr_mdata.need_mutable_data(queue);

   print_property("Is arr_data mutable", arr_data.has_mutable_data()); // false
   print_property("Is arr_mdata mutable", arr_mdata.has_mutable_data()); // true

   queue.submit([&](sycl::handler& cgh){
      auto mdata = arr_mdata.get_mutable_data();
      auto cones = arr_ones.get_data();
      cgh.parallel_for<class array_addition>(sycl::range<1>(data_count), [=](sycl::id
↪<1> idx) {
         mdata[idx[0]] += cones[idx[0]];
      });
   }).wait();

   std::cout << "arr_mdata values: ";
   for(std::int64_t i = 0; i < arr_mdata.get_count(); i++) {
      std::cout << arr_mdata[i] << ", ";
```

(continues on next page)

(continued from previous page)

```
    }
    std::cout << std::endl;

    return 0;
}
```

Data ownership requirements

The array shall support the following requirements on the internal data management:

1. An array shall own two properties representing raw pointers to the data:
    - `data` for a pointer to immutable data block
    - `mutable_data` for a pointer to mutable data block (see the *programming interface*)
2. If an array owns mutable data, both properties shall point to the same memory block.
3. If an array owns immutable data, `mutable_data` shall be `nullptr`.
4. An array shall store the number of elements in the block it owns and shall update the `count` property when a new memory block is assigned to the array.
5. An array shall store a pointer to the **ownership structure** of the data:
    - The **reference count** indicating how many array objects refer to the same memory block.
    - The **deleter** object used to free the memory block when reference count is zero.
6. An array shall create the ownership structure for a new memory block not associated with such structure.
7. An array shall decrement the number of references to the memory block when the array goes out of the scope. If the number of references is zero, the array shall call the deleter on this memory block and free the ownership structure.
8. An array shall store the pointer to the ownership structure created by another array when they share the data. An array shall increment the reference count for it to be equal to the number of array objects sharing the same data.

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/array.hpp` header file.

All the `array` class methods can be divided into several groups:

1. Constructors that are used to create an array from external, mutable or immutable memory.
2. Constructors and assignment operators that are used to create an array that shares its data with another one.
3. The group of `reset()` methods that are used to re-assign an array to another external memory block.
4. The group of `reset()` methods that are used to re-assign an array to an internally allocated memory block.
5. The methods that are used to access the data.
6. Static methods that provide simplified ways to create an array either from external memory or by allocating it within a new object.

```
template <typename Data>
class array {
public:
    using data_t = Data;

public:
    static array<Data> empty(const sycl::queue& queue,
                             std::int64_t count,
                             const sycl::usm::alloc& alloc =␣
↪sycl::usm::alloc::shared);

    template <typename Element>
    static array<Data> full(sycl::queue& queue,
                            std::int64_t count,
                            Element&& element,
                            const sycl::usm::alloc& alloc = sycl::usm::alloc::shared);

    static array<Data> zeros(sycl::queue& queue,
                             std::int64_t count,
                             const sycl::usm::alloc& alloc =␣
↪sycl::usm::alloc::shared);

    static array<Data> wrap(Data* data,
                            std::int64_t count,
                            const sycl::vector_class<sycl::event>& dependencies = {});

    static array<Data> wrap(const Data* data,
                            std::int64_t count,
                            const sycl::vector_class<sycl::event>& dependencies = {});

public:
    array();

    array(const array<Data>& other);

    array(array<Data>&& other);

    template <typename Deleter>
    explicit array(const sycl::queue& queue,
                   Data* data,
                   std::int64_t count,
                   Deleter&& deleter,
                   const sycl::vector_class<sycl::event>& dependencies = {});

    template <typename ConstDeleter>
    explicit array(const sycl::queue& queue,
                   const Data* data,
                   std::int64_t count,
                   ConstDeleter&& deleter,
                   const sycl::vector_class<sycl::event>& dependencies = {});

    template <typename RefData, typename ExtData>
    explicit array(const array<RefData>& ref, ExtData* data, std::int64_t count);

    array<Data> operator=(const array<Data>& other);

    array<Data> operator=(array<Data>&& other);
```

(continues on next page)

(continued from previous page)

```
    Data* get_mutable_data() const;

    const Data* get_data() const noexcept;

    bool has_mutable_data() const noexcept;

    array& need_mutable_data(sycl::queue& queue,
                             const sycl::usm::alloc& alloc =␣
↪sycl::usm::alloc::shared);

    std::int64_t get_count() const noexcept;

    std::int64_t get_size() const noexcept;

    void reset();

    void reset(const sycl::queue& queue,
               std::int64_t count,
               const sycl::usm::alloc& alloc = sycl::usm::alloc::shared);

    template <typename Deleter>
    void reset(Data* data,
               std::int64_t count,
               Deleter&& deleter,
               const sycl::vector_class<sycl::event>& dependencies = {});

    template <typename ConstDeleter>
    void reset(const Data* data,
               std::int64_t count,
               ConstDeleter&& deleter,
               const sycl::vector_class<sycl::event>& dependencies = {});


    template <typename RefData>
    void reset(const array<RefData>& ref, Data* data, std::int64_t count);

    template <typename RefData>
    void reset(const array<RefData>& ref, const Data* data, std::int64_t count);

    const Data& operator[](std::int64_t index) const noexcept;
};
```

template<typename **Data**>
class array

Template Parameters **Data** – The type of the memory block elements within the array. *Data* can represent any type.

Public Static Methods

static array<Data> **empty**(**const** sycl::queue &*queue*, std::int64_t *count*, **const** sycl::usm::alloc &*alloc* = sycl::usm::alloc::shared)

Allocates a new memory block for mutable data, does not initialize it, creates a new array instance by passing a pointer to the memory block. The array shall own the memory block (for details, see *data ownership requirements*).

Parameters

- **queue** – The SYCL* queue object.
- **count** – The number of elements of type $Data$ to allocate memory for.
- **alloc** – The kind of USM to be allocated.

Preconditions

count > 0

template<typename **Element**>
static array<Data> **full**(sycl::queue *&queue*, std::int64_t *count*, *Element* *&&element*, **const** sycl::usm::alloc *&alloc* = sycl::usm::alloc::shared)

Allocates a new memory block for mutable data, fills it with a scalar value, creates a new array instance by passing a pointer to the memory block. The array shall own the memory block (for details, see *data ownership requirements*).

Template Parameters **Element** – The type from which array elements of type $Data$ can be constructed.

Parameters

- **queue** – The SYCL* queue object.
- **count** – The number of elements of type $Data$ to allocate memory for.
- **element** – The value that is used to fill a memory block.
- **alloc** – The kind of USM to be allocated.

Preconditions

count > 0

Elements of type *Data* are constructible from the *Element* type.

static array<Data> **zeros**(sycl::queue *&queue*, std::int64_t *count*, **const** sycl::usm::alloc *&alloc* = sycl::usm::alloc::shared)

Allocates a new memory block on mutable data, fills it with zeros, creates a new array instance by passing a pointer to the memory block. The array shall own the memory block (for details, see *data ownership requirements*).

Parameters

- **queue** – The SYCL* queue object.
- **count** – The number of elements of type $Data$ to allocate memory for.
- **alloc** – The kind of USM to be allocated.

Preconditions

count > 0

static array<Data> **wrap**(Data **data*, std::int64_t *count*, **const** sycl::vector_class<sycl::event> *&dependencies* = {})

Creates a new array instance by passing the pointer to externally-allocated memory block for mutable data. It is the responsibility of the calling application to free the memory block as the array shall not free it when the reference count is zero.

Parameters

- **data** – The pointer to externally-allocated memory block.

- **count** – The number of elements of type $Data$ in the memory block.
- **dependencies** – Events indicating availability of the $data$ for reading or writing.

Preconditions

```
data != nullptr
count > 0
```

static array<Data> **wrap**(**const** Data **data*, std::int64_t *count*, **const** sycl::vector_class<sycl::event> &*dependencies* = {})

Creates a new array instance by passing the pointer to externally-allocated memory block for immutable data. It is the responsibility of the calling application to free the memory block as the array shall not free it when the reference count is zero.

Parameters

- **data** – The pointer to externally-allocated memory block.
- **count** – The number of elements of type $Data$ in the memory block.
- **dependencies** – Events indicating availability of the $data$ for reading or writing.

Preconditions

```
data != nullptr
count > 0
```

Constructors

array()

Creates a new instance of the class without memory allocation: `mutable_data` and `data` pointers shall be set to *nullptr*, `count` shall be to zero; the pointer to the ownership structure shall be set to *nullptr*.

array(**const** *array*<Data> &*other*)

Creates a new array instance that shares an ownership with $other$ on its memory block.

array(*array*<Data> &&*other*)

Moves `data`, `mutable_data` pointers, `count`, and pointer to the ownership structure in $other$ to the new array instance.

template<typename **Deleter**>
array(**const** sycl::queue &*queue*, Data **data*, std::int64_t *count*, *Deleter* &&*deleter*, **const** sycl::vector_class<sycl::event> &*dependencies* = {})

Creates a new array instance which owns a memory block of externally-allocated mutable data. The ownership structure shall be created for a block, the input $deleter$ shall be assigned to it.

Template Parameters **Deleter** – The type of a deleter used to free the $data$. The deleter shall provide *void operator()(Data*)* member function.

Parameters

- **queue** – The SYCL* queue object.
- **data** – The pointer to externally-allocated memory block.
- **count** – The number of elements of type $Data$ in the memory block.
- **deleter** – The object used to free $data$.
- **dependencies** – Events that indicate when $data$ becomes ready to be read or written.

template<typename **ConstDeleter**>

array(**const** sycl::queue &*queue*, **const** Data **data*, std::int64_t *count*, *ConstDeleter* &&*deleter*, **const** sycl::vector_class<sycl::event> &*dependencies* = {})

Creates a new array instance which owns a memory block of externally-allocated immutable data. The ownership structure shall be created for a block, the input $deleter$ shall be assigned to it.

Template Parameters **ConstDeleter** – The type of a deleter used to free the $data$. The deleter shall implement *void operator()(const Data*)* member function.

Parameters

- **queue** – The SYCL* queue object.
- **data** – The pointer to externally-allocated memory block.
- **count** – The number of elements of type $Data$ in the $data$.
- **deleter** – The object used to free $data$.
- **dependencies** – Events indicating availability of the $data$ for reading or writing.

template<typename **RefData**, typename **ExtData**>
array(**const** *array*<*RefData*> &*ref*, *ExtData* **data*, std::int64_t *count*)

An aliasing constructor: creates a new array instance that stores $data$ pointer, assigns the pointer to the ownership structure of ref to the new instance. Array shall return $data$ pointer as its mutable or immutable block depending on the $ExtData$ type.

Template Parameters

- **RefData** – The type of elements in the referenced array.
- **ExtData** – Either $Data$ or $const Data$ type.

Parameters

- **ref** – The array which shares ownership structure with created one.
- **data** – Mutable or immutable unmanaged pointer hold by created array.
- **count** – The number of elements of type $Data$ in the $data$.

Preconditions

```
std::is_same_v<ExtData, const Data> || std::is_same_v<ExtData,
Data>
```

Public Methods

array<Data> **operator=**(**const** array<Data> &*other*)

Replaces the `data`, `mutable_data` pointers, `count`, and pointer to the ownership structure in the array instance by the values in $other$.

Postconditions

```
data == other.data
mutable_data == other.mutable_data
count == other.count
```

array<Data> **operator=**(array<Data> &&*other*)

Swaps the values of `data`, `mutable_data` pointers, `count`, and pointer to the ownership structure in the array instance and $other$.

bool **has_mutable_data**() **const noexcept**

Returns whether array contains `mutable_data` or not.

array &**need_mutable_data**(sycl::queue *&queue*, **const** sycl::usm::alloc *&alloc* = sycl::usm::alloc::shared)

Returns mutable_data, if array contains it. Otherwise, allocates a memory block for mutable data and fills it with the data stored at `data`. Creates the ownership structure for allocated memory block and stores the pointer.

Parameters

- **queue** – The SYCL* queue object.
- **alloc** – The kind of USM to be allocated.

Postconditions

has_mutable_data() == true

void **reset**()

Resets ownership structure pointer to *nullptr*, sets `count` to zero, `data` and `mutable_data` to `nullptr`.

void **reset**(**const** sycl::queue *&queue*, std::int64_t *count*, **const** sycl::usm::alloc *&alloc* = sycl::usm::alloc::shared)

Allocates a new memory block for mutable data, does not initialize it, creates ownership structure for this block, assigns the structure inside the array. The array shall own allocated memory block.

Parameters

- **queue** – The SYCL* queue object.
- **count** – The number of elements of type $Data$ to allocate memory for.
- **alloc** – The kind of USM to be allocated.

template<typename **Deleter**>
void **reset**(Data **data*, std::int64_t *count*, *Deleter* &&*deleter*, **const** sycl::vector_class<sycl::event> &*dependencies* = {})

Creates the ownership structure for memory block of externally-allocated mutable data, assigns input $deleter$ object to it, sets *data* and `mutable_data` pointers to this block.

Template Parameters **Deleter** – The type of a deleter used to free the $data$. The deleter shall implement *void operator()(Data*)* member function.

Parameters

- **data** – The mutable memory block pointer to be assigned inside the array.
- **count** – The number of elements of type $Data$ into the block.
- **deleter** – The object used to free $data$.
- **dependencies** – Events indicating availability of the $data$ for reading or writing.

template<typename **ConstDeleter**>
void **reset**(**const** Data **data*, std::int64_t *count*, *ConstDeleter* &&*deleter*, **const** sycl::vector_class<sycl::event> &*dependencies* = {})

Creates the ownership structure for memory block of externally-allocated immutable data, assigns input $deleter$ object to it, sets *data* pointer to this block.

Template Parameters **ConstDeleter** – The type of a deleter used to free. The deleter shall implement *void operator()(const Data*)* ` member function.

Parameters

- **data** – The immutable memory block pointer to be assigned inside the array.

- **count** – The number of elements of type $Data$ into the block.
- **deleter** – The object used to free $data$.
- **dependencies** – Events indicating availability of the $data$ for reading or writing.

template<typename **RefData**>
void **reset**(**const** array<*RefData*> &*ref*, Data **data*, std::int64_t *count*)

Initializes *data* and `mutable_data` with data pointer, *count* with input $count$ value, initializes the pointer to ownership structure with the one from ref. Array shall return $data$ pointer as its mutable block.

Template Parameters **RefData** – The type of elements in the referenced array.

Parameters

- **ref** – The array which is used to share ownership structure with current one.
- **data** – Mutable unmanaged pointer to be assigned to the array.
- **count** – The number of elements of type $Data$ in the $data$.

template<typename **RefData**>
void **reset**(**const** array<*RefData*> &*ref*, **const** Data **data*, std::int64_t *count*)

Initializes *data* with data pointer, *count* with input $count$ value, initializes the pointer to ownership structure with the one from ref. Array shall return $data$ pointer as its immutable block.

Template Parameters **RefData** – The type of elements in the referenced array.

Parameters

- **ref** – The array which is used to share ownership structure with current one.
- **data** – Immutable unmanaged pointer to be assigned to the array.
- **count** – The number of elements of type $Data$ in the $data$.

const Data &**operator[]**(std::int64_t *index*) **const noexcept**

Provides a read-only access to the elements of array. Shall not perform boundary checks.

Properties

Data ***mutable_data**

The pointer to the memory block holding mutable data.

Getter & Setter

```
Data * get_mutable_data() const
```

Invariants

mutable_data `!= nullptr` if `has_mutable_data() &&` *count* `> 0`

const Data ***data**

The pointer to the memory block holding immutable data.

Getter & Setter

```
const Data * get_data() const noexcept
```

Invariants

data `!= nullptr` if *count* `> 0`
if `has_mutable_data() == true` then *data* `==` *mutable_data*

std::int64_t **count**

The number of elements of type $Data$ in a memory block.

Getter & Setter

```
std::int64_t get_count() const noexcept
```

std::int64_t **size**
: The size of memory block in bytes.

  Getter & Setter

```
std::int64_t get_size() const noexcept
```

  Invariants

```
size == count * sizeof(Data)
```

Accessors

This section defines *requirements* to an *accessor* implementation and introduces several *accessor types*.

Requirements

Each accessor implementation shall:

1. Define a single *format of the data* for the access. Every accessor type shall return and use only one data format.
2. Provide read-only access to the data in the *table* types.
3. Provide the `pull()` method for obtaining the values from the table.
4. Be lightweight. Its constructors shall not have computationally intensive operations such data copy, reading, or conversion. These operations shall be performed by method `pull()`. Support of copy- and move- constructors by the accessor is not required since it shall be designed for use in a local scope - directly in a place when it is created.
5. The `pull()` method shall avoid data copy and conversion when it is possible to return the pointer to the memory block in the table. This is applicable for cases such as when the *data format* and *data types* of the data within the table are the same as the *data format* and *data type* for the access.

Accessor Types

oneDAL defines a set of accessor classes. Each class supports one specific way of obtaining data from the *table*.

All accessor classes in oneDAL are listed below:

Accessor type	Description	List of supported types
row accessor	Provides access to the range of rows as one *contiguous homogeneous* block of memory.	*homogen table*
column accessor	Provides access to the range of values within a single column as one *contiguous homogeneous* block of memory.	*homogen table*

Details

Column accessor

The `column_accessor` class provides a read-only access to the column values of the *table* as *contiguous homogeneous* array.

Usage example

```
#include <CL/sycl.hpp>
#include <iostream>

#include "oneapi/dal/table/homogen.hpp"
#include "oneapi/dal/table/column_accessor.hpp"

using namespace oneapi;

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr float host_data[] = {
      1.0f, 1.5f, 2.0f,
      2.1f, 3.2f, 3.7f,
      4.0f, 4.9f, 5.0f,
      5.2f, 6.1f, 6.2f
   };

   constexpr std::int64_t row_count = 4;
   constexpr std::int64_t column_count = 3;

   auto shared_data = sycl::malloc_shared<float>(row_count * column_count, queue);
   auto event = queue.memcpy(shared_data, host_data, sizeof(float) * row_count *␣
↪column_count);
   auto t = dal::homogen_table::wrap(queue, data, row_count, column_count, { event });

   // Accessing whole elements in a first column
   dal::column_accessor<const float> acc { t };

   auto block = acc.pull(queue, 0);
   for(std::int64_t i = 0; i < block.get_count(); i++) {
      std::cout << block[i] << ", ";
   }
   std::cout << std::endl;

   sycl::free(shared_data, queue);
   return 0;
}
```

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/column_accessor.hpp` header file.

```
template <typename Data>
class column_accessor {
public:
    using data_t = std::remove_const_t<Data>;

public:
    column_accessor(const table& obj);

    array<data_t> pull(sycl::queue& queue,
                       std::int64_t column_index,
                       const range& rows              = { 0, -1 },
                       const sycl::usm::alloc& alloc = sycl::usm::alloc::shared)␣
↪const;

    Data* pull(sycl::queue& queue,
               array<data_t>& block,
               std::int64_t column_index,
               const range& rows              = { 0, -1 },
               const sycl::usm::alloc& alloc = sycl::usm::alloc::shared) const;
};
```

template<typename **Data**>
class column_accessor

Template Parameters **Data** – The type of data values in blocks returned by the accessor. Shall be const-qualified for read-only access. An accessor shall support at least `float`, `double`, and `std::int32_t` types of *Data*.

Constructors

column_accessor(**const** *table* &*obj*)

Creates a new read-only accessor object from the table. The check that the accessor supports the table kind of *obj* shall be performed. The reference to the *obj* table shall be stored within the accessor to obtain data from the table.

Public Methods

array<data_t> **pull** (sycl::queue &*queue*, std::int64_t *column_index*, **const** *range* &*rows* = {0, -1}, **const** sycl::usm::alloc &*alloc* = sycl::usm::alloc::shared) **const**

Provides access to the column values of the table. The method shall return an array that directly points to the memory within the table if it is possible. In that case, the array shall refer to the memory as to immutable data. Otherwise, the new memory block shall be allocated, the data from the table rows shall be converted and copied into this block. The array shall refer to the block as to mutable data.

Parameters

- **queue** – The SYCL* queue object.
- **column_index** – The index of the column from which the data shall be returned by the accessor.
- **rows** – The range of rows that should be read in the $column_index$ block.
- **alloc** – The requested kind of USM in the returned block.

Preconditions

rows are within the range of *[0, obj.row_count)*.
column_index is within the range of *[0, obj.column_count)*.

Data *__pull__(sycl::queue &*queue*, *array*<data_t> &*block*, std::int64_t *column_index*, __const__ *range* &*rows* = {0, -1}, __const__ sycl::usm::alloc &*alloc* = sycl::usm::alloc::shared) __const__
Provides access to the column values of the table. The method shall return the `block.data` pointer.

Parameters

- **queue** – The SYCL* queue object.
- **block** – The block which memory is reused (if it is possible) to obtain the data from the table. The block memory shall be reset either when its size is not big enough, or when it contains immutable data, or when direct memory from the table can be used. If the block is reset to use a direct memory pointer from the object, it shall refer to this pointer as to immutable memory block.
- **column_index** – The index of the column from which the data shall be returned by the accessor.
- **rows** – The range of rows that should be read in the $column_index$ block.
- **alloc** – The requested kind of USM in the returned block.

Preconditions

rows are within the range of *[0, obj.row_count)*.
column_index is within the range of *[0, obj.column_count)*.

Row accessor

The `row_accessor` class provides a read-only access to the rows of the *table* as *contiguous homogeneous* array.

Usage example

```
#include <CL/sycl.hpp>
#include <iostream>

#include "oneapi/dal/table/homogen.hpp"
#include "oneapi/dal/table/row_accessor.hpp"

using namespace oneapi;

int main() {
   sycl::queue queue { sycl::default_selector() };

   constexpr float host_data[] = {
      1.0f, 1.5f, 2.0f,
      2.1f, 3.2f, 3.7f,
      4.0f, 4.9f, 5.0f,
      5.2f, 6.1f, 6.2f
   };

   constexpr std::int64_t row_count = 4;
   constexpr std::int64_t column_count = 3;
```

(continues on next page)

(continued from previous page)

```
    auto shared_data = sycl::malloc_shared<float>(row_count * column_count, queue);
    auto event = queue.memcpy(shared_data, host_data, sizeof(float) * row_count *␣
↪column_count);
    auto t = dal::homogen_table::wrap(queue, data, row_count, column_count, { event });

    // Accessing second and third rows of the table
    dal::row_accessor<const float> acc { t };

    auto block = acc.pull(queue, {1, 3});
    for(std::int64_t i = 0; i < block.get_count(); i++) {
        std::cout << block[i] << ", ";
    }
    std::cout << std::endl;

    sycl::free(shared_data, queue);
    return 0;
}
```

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/row_accessor.hpp` header file.

```
template <typename Data>
class row_accessor {
public:
    using data_t = std::remove_const_t<Data>;

public:
    row_accessor(const table& obj);

    array<data_t> pull(sycl::queue& queue,
                       const range& rows             = { 0, -1 },
                       const sycl::usm::alloc& alloc = sycl::usm::alloc::shared)␣
↪const;

    Data* pull(sycl::queue& queue,
               array<data_t>& block,
               const range& rows             = { 0, -1 },
               const sycl::usm::alloc& alloc = sycl::usm::alloc::shared) const;
};
```

template<typename **Data**>
class row_accessor

Template Parameters **Data** – The type of data values in blocks returned by the accessor. Shall be const-qualified for read-only access. An accessor shall support at least `float`, `double`, and `std::int32_t` types of *Data*.

Constructors

row_accessor(**const** *table* &*obj*)

Creates a new read-only accessor object from the table. The check that the accessor supports the table kind of *obj* shall be performed. The reference to the *obj* table shall be stored within the accessor to obtain data from the table.

Public Methods

array<data_t> **pull** (sycl::queue &*queue*, **const** *range* &*rows* = {0, -1}, **const** sycl::usm::alloc &*alloc* = sycl::usm::alloc::shared) **const**

Provides access to the rows of the table. The method shall return an array that directly points to the memory within the table if it is possible. In that case, the array shall refer to the memory as to immutable data. Otherwise, the new memory block shall be allocated, the data from the table rows shall be converted and copied into this block. The array shall refer to the block as to mutable data.

Parameters

- **queue** – The SYCL* queue object.
- **rows** – The range of rows that data shall be returned from the accessor.
- **alloc** – The requested kind of USM in the returned block.

Preconditions

rows are within the range of *[0, obj.row_count)*.

Data ***pull** (sycl::queue &*queue*, *array*<data_t> &*block*, **const** *range* &*rows* = {0, -1}, **const** sycl::usm::alloc &*alloc* = sycl::usm::alloc::shared) **const**

Provides access to the rows of the table. The method shall return the *block*.data pointer.

Parameters

- **queue** – The SYCL* queue object.
- **block** – The block which memory is reused (if it is possible) to obtain the data from the table. The block memory shall be reset either when its size is not big enough, or when it contains immutable data, or when direct memory from the table can be used. If the block is reset to use a direct memory pointer from the object, it shall refer to this pointer as to immutable memory block.
- **rows** – The range of rows that data shall be returned from the accessor.
- **alloc** – The requested kind of USM in the returned block.

Preconditions

rows are within the range of *[0, obj.row_count)*.

Data Sources

This section describes the types related to the *data source* concept.

Read

Read operation is a function that transforms a data source and other arguments represented via *an args* object to *a result* object. The operation is responsible for:

- Executing all of the data retrieval and transformation routines of the data source.
- Passing a SYCL* queue to the data retrieval and transformation routines.

Read operation definition

The following code sample shows the declaration for a read operation.

```
namespace oneapi::dal {

template <typename Object, typename DataSource>
using read_args_t = /* implementation defined */;

template <typename Object, typename DataSource>
using read_result_t = Object;

template <typename Object, typename DataSource>
read_result_t<Object, DataSource> read(
   sycl::queue& queue,
   const DataSource& data_source,
   const read_args_t<Object, DataSource>& args);

} // namespace oneapi::dal
```

Each operation shall satisfy the following requirements:

- An operation shall accept three parameters in the following order:
  - The SYCL* queue object.
  - The data source.
  - The *args object*.
- An operation shall return the *result object*.
- The `read_args_t` and `read_result_t` alias templates shall be used for inference of the args and return types.

Read operation shortcuts

In order to make the code on user side less verbose, oneDAL defines the following overloaded functions called *shortcuts* for a read operation in addition to the general one described in section *Read operation definition*.

- A shortcut for execution on *host*. Performs the same operation as the general function on host, but does not require passing the queue explicitly.

```
template <typename Object, typename DataSource>
read_result_t<Object, DataSource> read(
   const DataSource& data_source,
   const read_args_t<Object, DataSource>& args);
```

- A shortcut that allows omitting explicit args creation.

```
template <typename Object, typename DataSource, typename... Args>
read_result_t<Object, DataSource> read(
   sycl::queue& queue,
   const DataSource& data_source,
   Args&&... args);
```

- A shortcut that allows omitting explicit queue and args creation. This is a combination of two previous shortcuts.

```
template <typename Object, typename DataSource, typename... Args>
read_result_t<Object, DataSource> read(
   const DataSource& data_source,
   Args&&... args);
```

Args

- The string `%DATA_SOURCE%` should be substituted with the name of the data source, for example, `csv`.
- `%PROPERTY_NAME%` and `%PROPERTY_TYPE%` should be substituted with the name and the type of one of the data source args properties.

```
namespace oneapi::dal::%DATA_SOURCE% {

template <typename Object, typename DataSource>
class read_args {
public:
   read_args(
      const %PROPERTY_TYPE_1%& property_name_1,
      const %PROPERTY_TYPE_2%& property_name_2,
      /* more properties */
   )
   /* Getter & Setter for the property called `%PROPERTY_NAME_1%` */
   descriptor& set_%PROPERTY_NAME_1%(%PROPERTY_TYPE_1% value);
   %PROPERTY_TYPE_1% get_%PROPERTY_NAME_1%() const;
   /* Getter & Setter for the property called `%PROPERTY_NAME_2%` */
   descriptor& set_%PROPERTY_NAME_2%(%PROPERTY_TYPE_2% value);
   %PROPERTY_TYPE_2% get_%PROPERTY_NAME_2%() const;
   /* more properties */
};
} // namespace oneapi::dal::%DATA_SOURCE%
```

Result

The result of a `read` operation is an instance of an in-memory object with `Object` type.

Data Source Types

oneDAL defines a set of classes.

Data source type	Description
CSV data source	Data source that allows reading data from a text file into a *table*.

Details

CSV data source

Class `csv::data_source` is an API for accessing the data source represented as a *csv file*. CSV data source shall be used with `read` operation to extract data in text format from the given input file, process it using provided parameters (such as delimiter and read options), transform it into numerical representation, and store it as an in-memory *dataset* of a chosen type.

Supported type of in-memory object for `read` operation with CSV data source is `oneapi::dal::table`.

CSV data source requires input file name to be set in the constructor, while the other parameters of the constructor such as delimiter and read options rely on default values.

Usage example

```
using namespace oneapi;

const auto data_source = dal::csv::data_source("data.csv", ',');

const auto table = dal::read<dal::table>(data_source);
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::csv` namespace and be available via inclusion of the `oneapi/dal/io/csv.hpp` header file.

```
enum class read_options : std::uint64_t {
   none = 0,
   parse_header = 1 << 0
};

constexpr char default_delimiter = ',';
constexpr read_options default_read_options = read_options::none;

class data_source {
public:
   data_source(const char *file_name,
               char delimiter = default_delimiter,
               read_options opts = default_read_options);

   data_source(const std::string &file_name,
               char delimiter = default_delimiter,
               read_options opts = default_read_options);

   std::string get_file_name() const;
   char get_delimiter() const;
   read_options get_read_options() const;
};
```

class data_source

data_source(**const** char **file_name*, char *delimiter* = default_delimiter, read_options *opts* = default_read_options)
Creates a new instance of a CSV data source with the given *file_name*, *delimiter* and read options *opts* flag.

data_source(**const** std::string &*file_name*, char *delimiter* = default_delimiter, read_options *opts* = default_read_options)
Creates a new instance of a CSV data source with the given *file_name*, *delimiter* and read options *opts* flag.

std::string **file_name** = ""
A string that contains the name of the file with the dataset to read.

Getter

```
std::string get_filename() const
```

char **delimiter** = default_delimiter
A character that represents the delimiter between separate features in the input file.

Getter

```
char get_delimter() const
```

read_options **options** = default_read_options
Value that stores read options to be applied during reading of the input file. Enabled `parse_header` option indicates that the first line in the input file shall be processed as a header record with features names.

Getter

```
read_options get_read_options() const
```

Reading `oneapi::dal::read<Object>...`

Args

```
template <typename Object>
class read_args {
public:
   read_args();
};
```

template<typename **Object**>
class read_args

read_args()
Creates args for the read operation with the default attribute values.

Operation

`oneapi::dal::table` is the only supported value of the `Object` template parameter for *read* operation with CSV data source.

template<typename **Object**, typename **DataSource**>
Object **read**(**const** *DataSource* &*ds*)

Template Parameters

- **Object** – oneDAL object type that shall be produced as a result of reading from the data source.
- **DataSource** – CSV data source `csv::data_source`.

Tables

This section describes the types related to the *table* concept.

Type	Description
table	A common implementation of the table concept. Base class for other table types.
table_metadata	An implementation of *table metadata* concept.
data_layout	An enumeration of *data layouts* used to store contiguous data blocks inside the table.
feature_type	An enumeration of *feature* types used in oneDAL to define set of available operations onto the data.

Requirements on table types

Each implementation of *table* concept shall:

1. Follow the definition of the *table* concept and its restrictions (e.g., *immutability*).
2. Be derived from the `oneapi::dal::table` class. The behavior of this class can be extended, but cannot be weaken.
3. Be *reference-counted*.
4. Every new `oneapi::dal::table` sub-type shall define a unique id number - the "kind" that represents objects of that type in runtime.

The following listing provides an example of table API to illustrate table kinds and copy-assignment operation:

```
using namespace onedal;

// Creating homogen_table sub-type.
dal::homogen_table table1 = homogen_table::wrap(queue, data_ptr, row_count, column_
↪count);

// table1 and table2 share the same data (no data copy is performed)
dal::table table2 = table1;

// Creating an empty table
dal::table table3;

std::cout << table1.get_kind()      == table2.get_kind() << std::endl; // true
std::cout << homogen_table::kind() == table2.get_kind() << std::endl; // true
std::cout << table2.get_kind()      == table3.get_kind() << std::endl; // false
```

(continues on next page)

(continued from previous page)

```
// Referring table3 to the table2.
table3 = table2;
std::cout << table2.get_kind() == table3.get_kind() << std::endl; // true
```

Table types

oneDAL defines a set of classes that implement the *table* concept for a specific data format:

Table type	Description
homogen table	A dense table that contains *contiguous homogeneous* data.

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/common.hpp` header file.

Table

A base implementation of the *table* concept. The `table` type and all of its subtypes shall be *reference-counted*:

1. The instance shall store a pointer to table implementation that holds all property values and data
2. The reference count indicating how many table objects refer to the same implementation.
3. The table shall increment the reference count for it to be equal to the number of table objects sharing the same implementation.
4. The table shall decrement the reference count when the table goes out of the scope. If the reference count is zero, the table shall free its implementation.

```
class table {
public:
    table();

    table(const table& other);

    table(table&& other);

    table& operator=(const table& other);

    table& operator=(table&& other);

    bool has_data() const noexcept;

    std::int64_t get_column_count() const;

    std::int64_t get_row_count() const;

    const table_metadata& get_metadata() const;

    std::int64_t get_kind() const;
```

(continues on next page)

(continued from previous page)

```
    data_layout get_data_layout() const;
};
```

class table

Constructors

table()
An empty table constructor: creates the table instance with zero number of rows and columns. Implementation shall be set to the special "empty" object that returns all the property values set to default (see Properties section).

table(**const** *table* &*other*)
Creates a new table instance which shares implementation with $other$.

table(*table* &&*other*)
Creates a new table instance and moves implementation from $other$ into it.

Public Methods

table &**operator=**(**const** *table* &*other*)
Replaces the implementation by another one from $other$.

table &**operator=**(*table* &&*other*)
Swaps the implementation of this object and $other$.

bool **has_data**() **const noexcept**
Indicates whether a table contains non-zero number of rows and columns.

Properties

std::int64_t **column_count** = 0
The number of columns in the table.

Getter & Setter

```
std::int64_t get_column_count() const
```

std::int64_t **row_count** = 0
The number of rows in the table.

Getter & Setter

```
std::int64_t get_row_count() const
```

const table_metadata &**metadata** = table_metadata()
The metadata object that holds additional information about the data within the table.

Getter & Setter

```
const table_metadata & get_metadata() const
```

std::int64_t **kind** = empty_table_kind
The runtime id of the table type. Each table sub-type shall have its unique *kind*. An empty table (see the default constructor) shall have a unique *kind* value as well.

Getter & Setter

```
std::int64_t get_kind() const
```

data_layout **data_layout** = *data_layout*::unknown
The layout of the data within the table.

Getter & Setter

```
data_layout get_data_layout() const
```

Table metadata

An implementation of the *table metadata* concept. Holds additional information about data within the table. The objects of `table_metadata` shall be *reference-counted*.

```
class table_metadata {
public:
    table_metadata();

    table_metadata(const array<data_type>& dtypes, const array<feature_type>& ftypes);

    std::int64_t get_feature_count() const;

    const feature_type& get_feature_type(std::int64_t feature_index) const;

    const data_type& get_data_type(std::int64_t feature_index) const;
};
```

class table_metadata

Constructors

table_metadata()

Creates the metadata instance without information about the features. The `feature_count` shall be set to zero. The *data_type* and *feature_type* properties shall not be initialized.

table_metadata(**const** *array*<*data_type*> &*dtypes*, **const** *array*<*feature_type*> &*ftypes*)

Creates the metadata instance from external information about the data types and the feature types.

Parameters

- **dtypes** – The data types of the features. Shall be assigned into the *data_type* property.
- **ftypes** – The feature types. Shall be assigned into the *feature_type* property.

Preconditions

dtypes.get_count() == *ftypes*.get_count()

Properties

std::int64_t **feature_count**

The number of features that metadata contains information about.

Getter & Setter

```
std::int64_t get_feature_count() const
```

const *feature_type* &**feature_type**

Feature types in the metadata object. Shall be within the range *[0, feature_count)*.

Getter & Setter

```
const feature_type & get_feature_type(std::int64_t feature_index)
const
```

const *data_type* &**data_type**

Data types of the features in the metadata object. Shall be within the range *[0, feature_count)*.

Getter & Setter

```
const data_type & get_data_type(std::int64_t feature_index) const
```

Data layout

An implementation of the *data layout* concept.

```
enum class data_layout { unknown, row_major, column_major };
```

enum class data_layout

data_layout::unknown Represents the *data layout* that is undefined or unknown at this moment.

data_layout::row_major The data block elements are stored in raw-major layout.

data_layout::column_major The data block elements are stored in column_major layout.

Feature type

An implementation of the logical data types.

```
enum class feature_type { nominal, ordinal, interval, ratio };
```

enum class feature_type

feature_type::nominal Represents the type of *Nominal feature*.

feature_type::ordinal Represents the type of *Ordinal feature*.

feature_type::interval Represents the type of *Interval feature*.

feature_type::ratio Represents the type of *Ratio feature*.

Homogeneous table

Class `homogen_table` is an implementation of a table type for which the following is true:

- The data within the table are dense and stored as one contiguous memory block.
- All the columns have the same *data type*.

Programming interface

All types and functions in this section shall be declared in the `oneapi::dal` namespace and be available via inclusion of the `oneapi/dal/table/homogen.hpp` header file.

```
class homogen_table : public table {
public:
    static std::int64_t kind();

    template <typename Data>
    static homogen_table wrap(const sycl::queue& queue,
                              const Data* data_pointer,
                              std::int64_t row_count,
                              std::int64_t column_count,
```

(continues on next page)

(continued from previous page)

```
                                const sycl::vector_class<sycl::event>& dependencies = {}
↪,
                                data_layout layout = data_layout::row_major);

public:
    homogen_table();

    template <typename Data, typename ConstDeleter>
    homogen_table(const sycl::queue& queue,
                  const Data* data_pointer,
                  std::int64_t row_count,
                  std::int64_t column_count,
                  ConstDeleter&& data_deleter,
                  const sycl::vector_class<sycl::event>& dependencies = {},
                  data_layout layout                                  = data_
↪layout::row_major);

    template <typename Data>
    const Data* get_data() const {
        return reinterpret_cast<const Data*>(this->get_data());
    }

    const void* get_data() const;

    std::int64_t get_kind() const {
        return kind();
    }
};
```

class homogen_table

Public Static Methods

static std::int64_t **kind**()

Returns the unique id of *homogen_table* class.

template<typename **Data**>

static *homogen_table* **wrap**(**const** sycl::queue &*queue*, **const** *Data* *data_pointer*, std::int64_t *row_count*, std::int64_t *column_count*, **const** sycl::vector_class<sycl::event> &*dependencies* = {}, *data_layout* *layout* = *data_layout*::row_major)

Creates a new *homogen_table* instance from externally-defined data block. Table object refers to the data but does not own it. The responsibility to free the data remains on the user side. The `data` shall point to the *data_pointer* memory block.

Template Parameters **Data** – The type of elements in the data block that will be stored into the table. The table shall initialize data types of metadata with this data type. The feature types shall be set to default values for *Data* type: contiguous for floating-point, ordinal for integer types. The *Data* type shall be at least `float`, `double` or `std::int32_t`.

Parameters

- **queue** – The SYCL* queue object.
- **data_pointer** – The pointer to a homogeneous data block.
- **row_count** – The number of rows in the table.
- **column_count** – The number of columns in the table.
- **dependencies** – Events indicating availability of the *data* for reading or writing.

- **layout** – The layout of the data. Shall be *data_layout::row_major* or *data_layout::column_major*.

Constructors

homogen_table()
Creates a new *homogen_table* instance with zero number of rows and columns. The `kind` shall be set to`homogen_table::kind()`. All the properties shall be set to default value (see the Properties section).

template<typename **Data**, typename **ConstDeleter**>
homogen_table(**const** sycl::queue &*queue*, **const** *Data* *data_pointer, std::int64_t *row_count*, std::int64_t *column_count*, *ConstDeleter* &&*data_deleter*, **const** sycl::vector_class<sycl::event> &*dependencies* = {}, *data_layout* *layout* = *data_layout*::row_major)
Creates a new *homogen_table* instance from externally-defined data block. Table object owns the data pointer. The `data` shall point to the *data_pointer* memory block.

Template Parameters

- **Data** – The type of elements in the data block that will be stored into the table. The $Data$ type shall be at least `float`, `double` or `std::int32_t`.
- **ConstDeleter** – The type of a deleter called on *data_pointer* when the last table that refers it is out of the scope.

Parameters

- **queue** – The SYCL* queue object.
- **data_pointer** – The pointer to a homogeneous data block.
- **row_count** – The number of rows in the table.
- **column_count** – The number of columns in the table.
- **data_deleter** – The deleter that is called on the *data_pointer* when the last table that refers it is out of the scope.
- **dependencies** – Events indicating availability of the $data$ for reading or writing.
- **layout** – The layout of the data. Shall be *data_layout::row_major* or *data_layout::column_major*.

Public Methods

template<typename **Data**>
const *Data* ***get_data**() **const**
Returns the `data` pointer cast to the $Data$ type. No checks are performed that this type is the actual type of the data within the table.

Properties

const void ***data**
The pointer to the data block within the table. Shall be equal to *nullptr* when `row_count == 0` and `column_count == 0`.

Getter & Setter

```
const void * get_data() const
```

std::int64_t **kind**
The unique id of the homogen table type.

Getter & Setter

```
std::int64_t get_kind() const
```

8.7 Algorithms

The Algorithms component consists of classes that implement algorithms for data analysis (data mining) and data modeling (training and prediction). These algorithms include matrix decompositions, clustering, classification, and regression algorithms, as well as association rules discovery.

8.7.1 Clustering

K-Means

The K-Means algorithm solves *clustering* problem by partitioning n feature vectors into k clusters minimizing some criterion. Each cluster is characterized by a representative point, called *a centroid*.

Operation	Computational methods	Programming Interface		
Training	*Lloyd's*	*train(...)*	*train_input*	*train_result*
Inference	*Lloyd's*	*infer(...)*	*infer_input*	*infer_result*

Mathematical formulation

Training

Given the training set $X = \{x_1, \ldots, x_n\}$ of p-dimensional feature vectors and a positive integer k, the problem is to find a set $C = \{c_1, \ldots, c_k\}$ of p-dimensional centroids that minimize the objective function

$$\Phi_X(C) = \sum_{i=1}^{n} d^2(x_i, C),$$

where $d^2(x_i, C)$ is the squared Euclidean distance from x_i to the closest centroid in C,

$$d^2(x_i, C) = \min_{1 \le j \le k} \|x_i - c_j\|^2, \quad 1 \le i \le n.$$

Expression $\|\cdot\|$ denotes L_2 norm.

Note: In the general case, d may be an arbitrary distance function. Current version of the oneDAL spec defines only Euclidean distance case.

Training method: *Lloyd's*

The Lloyd's method [Lloyd82] consists in iterative updates of centroids by applying the alternating *Assignment* and *Update* steps, where t denotes a index of the current iteration, e.g., $C^{(t)} = \{c_1^{(t)}, \ldots, c_k^{(t)}\}$ is the set of centroids at the t-th iteration. The method requires the initial centroids $C^{(1)}$ to be specified at the beginning of the algorithm ($t = 1$).

(1) Assignment step: Assign each feature vector x_i to the nearest centroid. $y_i^{(t)}$ denotes the assigned label (cluster index) to the feature vector x_i.

$$y_i^{(t)} = \arg \min_{1 \le j \le k} \|x_i - c_j^{(t)}\|^2, \quad 1 \le i \le n.$$

Each feature vector from the training set X is assigned to exactly one centroid so that X is partitioned to k disjoint sets (clusters)

$$S_j^{(t)} = \left\{ x_i \in X : y_i^{(t)} = j \right\}, \quad 1 \le j \le k.$$

(2) Update step: Recalculate centroids by averaging feature vectors assigned to each cluster.

$$c_j^{(t+1)} = \frac{1}{|S_j^{(t)}|} \sum_{x \in S_j^{(t)}} x, \quad 1 \le j \le k.$$

The steps (1) and (2) are performed until the following **stop condition**,

$$\sum_{j=1}^{k} \left\| c_j^{(t)} - c_j^{(t+1)} \right\|^2 < \varepsilon,$$

is satisfied or number of iterations exceeds the maximal value T defined by the user.

Inference

Given the inference set $X' = \{x'_1, \ldots, x'_m\}$ of p-dimensional feature vectors and the set $C = \{c_1, \ldots, c_k\}$ of centroids produced at the training stage, the problem is to predict the index $y'_j \in \{0, \ldots, k-1\}$, $1 \le j \le m$, of the centroid in accordance with a method-defined rule.

Inference method: *Lloyd's*

Lloyd's inference method computes the y'_j as an index of the centroid closest to the feature vector x'_j,

$$y'_j = \arg\min_{1 \le l \le k} \|x'_j - c_l\|^2, \quad 1 \le j \le m.$$

Usage example

Training

```
kmeans::model<> run_training(const table& data,
                             const table& initial_centroids) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(10)
      .set_max_iteration_count(50)
      .set_accuracy_threshold(1e-4);

   const auto result = train(kmeans_desc, data, initial_centroids);

   print_table("labels", result.get_labels());
   print_table("centroids", result.get_model().get_centroids());
   print_value("objective", result.get_objective_function_value());

   return result.get_model();
}
```

Inference

```
table run_inference(const kmeans::model<>& model,
                    const table& new_data) {
   const auto kmeans_desc = kmeans::descriptor<float>{}
      .set_cluster_count(model.get_cluster_count());

   const auto result = infer(kmeans_desc, model, new_data);

   print_table("labels", result.get_labels());
}
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::kmeans` namespace and be available via inclusion of the `oneapi/dal/algo/kmeans.hpp` header file.

Descriptor

```
template <typename Float = float,
          typename Method = method::by_default,
          typename Task = task::by_default>
class descriptor {
public:
   explicit descriptor(std::int64_t cluster_count = 2);

   int64_t get_cluster_count() const;
   descriptor& set_cluster_count(int64_t);

   int64_t get_max_iteration_count() const;
   descriptor& set_max_iteration_count(int64_t);

   double get_accuracy_threshold() const;
   descriptor& set_accuracy_threshold(double);
};
```

template<typename **`Float`** = float, typename **`Method`** = method::*by_default*, typename **`Task`** = task::*by_default*>
`class descriptor`

Template Parameters

- **`Float`** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **`Method`** – Tag-type that specifies an implementation of algorithm. Can be `method::`*`lloyd`*.
- **`Task`** – Tag-type that specifies the type of the problem to solve. Can be `task::`*`clustering`*.

Constructors

`descriptor`(std::int64_t *cluster_count* = 2)
Creates a new instance of the class with the given `cluster_count`.

Properties

int64_t **cluster_count** = 2
: The number of clusters k.

Getter & Setter

```
int64_t get_cluster_count() const
descriptor & set_cluster_count(int64_t)
```

Invariants

```
cluster_count > 0
```

int64_t **max_iteration_count** = 100
: The maximum number of iterations T.

Getter & Setter

```
int64_t get_max_iteration_count() const
descriptor & set_max_iteration_count(int64_t)
```

Invariants

```
max_iteration_count >= 0
```

double **accuracy_threshold** = 0.0
: The threshold ε for the stop condition.

Getter & Setter

```
double get_accuracy_threshold() const
descriptor & set_accuracy_threshold(double)
```

Invariants

```
accuracy_threshold >= 0.0
```

Method tags

```
namespace method {
   struct lloyd {};
   using by_default = lloyd;
} // namespace method
```

struct lloyd
: Tag-type that denotes *Lloyd's* computational method.

using by_default = *lloyd*
: Alias tag-type for *Lloyd's* computational method.

Task tags

```
namespace task {
   struct clustering {};
   using by_default = clustering;
} // namespace task
```

struct clustering
: Tag-type that parameterizes entities used for solving *clustering problem*.

using by_default = *clustering*
Alias tag-type for the clustering task.

Model

```
template <typename Task = task::by_default>
class model {
public:
   model();

   const table& get_centroids() const;

   int64_t get_cluster_count() const;
};
```

template<typename **Task** = task::*by_default*>
class model

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Constructors

model()
Creates a new instance of the class with the default property values.

Properties

const table &**centroids** = table{}
A $k \times p$ table with the cluster centroids. Each row of the table stores one centroid.

Getter & Setter

```
const table & get_centroids() const
```

int64_t **cluster_count** = 0
Number of clusters k in the trained model.

Getter & Setter

```
int64_t get_cluster_count() const
```

Invariants

```
cluster_count == centroids.row_count
```

Training `train...`

Input

```
template <typename Task = task::by_default>
class train_input {
public:
   train_input(const table& data = table{},
               const table& initial_centroids = table{});

   const table& get_data() const;
   train_input& set_data(const table&);
```

(continues on next page)

(continued from previous page)

```
   const table& get_initial_centroids() const;
   train_input& set_initial_centroids(const table&);
};
```

template<typename **Task** = task::*by_default*>
class train_input

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Constructors

train_input(**const** *table* &*data* = *table*{}, **const** *table* &*initial_centroids* = *table*{})
Creates a new instance of the class with the given `data` and `initial_centroids`.

Properties

const table &**data**
An $n \times p$ table with the data to be clustered, where each row stores one feature vector.

Getter & Setter

```
const table & get_data() const
train_input & set_data(const table &)
```

const table &**initial_centroids**
A $k \times p$ table with the initial centroids, where each row stores one centroid.

Getter & Setter

```
const table & get_initial_centroids() const
train_input & set_initial_centroids(const table &)
```

Result

```
template <typename Task = task::by_default>
class train_result {
public:
   train_result();

   const model<Task>& get_model() const;

   const table& get_labels() const;

   int64_t get_iteration_count() const;

   double get_objective_function_value() const;
};
```

template<typename **Task** = task::*by_default*>
class train_result

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Constructors

train_result()
Creates a new instance of the class with the default property values.

Properties

const *model*<Task> &**model** = *model*<Task>{}
The trained K-means model.

Getter & Setter

```
const model< Task > & get_model() const
```

const table &**labels** = table{}
An $n \times 1$ table with the labels y_i assigned to the samples x_i in the input data, $1 \leq 1 \leq n$.

Getter & Setter

```
const table & get_labels() const
```

int64_t **iteration_count** = 0
The number of iterations performed by the algorithm.

Getter & Setter

```
int64_t get_iteration_count() const
```

Invariants

```
iteration_count >= 0
```

double **objective_function_value**
The value of the objective function $\Phi_X(C)$, where C is *model*.centroids (see kmeans::model::centroids).

Getter & Setter

```
double get_objective_function_value() const
```

Invariants

```
objective_function_value >= 0.0
```

Operation

template<typename **Float**, typename **Method**, typename **Task**>
train_result<*Task*> **train**(**const** *descriptor*<*Float*, *Method*, *Task*> &*desc*, **const** *train_input*<*Task*> &*input*)
Runs the training operation for K-Means clustering. For more details see oneapi::dal::train.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be method::*lloyd*.
- **Task** – Tag-type that specifies type of the problem to solve. Can be task::*clustering*.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the training operation.

Preconditions

```
input.data.has_data == true
input.initial_centroids.row_count == desc.cluster_count
input.initial_centroids.column_count == input.data.column_count
```

Postconditions

```
result.labels.row_count == input.data.row_count
result.labels.column_count == 1
result.labels[i] >= 0
result.labels[i] < desc.cluster_count
result.iteration_count <= desc.max_iteration_count
result.model.centroids.row_count == desc.cluster_count
result.model.centroids.column_count == input.data.column_count
```

Inference `infer...`

Input

```
template <typename Task = task::by_default>
class infer_input {
public:
   infer_input(const model<Task>& m = model<Task>{},
               const table& data = table{});

   const model<Task>& get_model() const;
   infer_input& set_model(const model<Task>&);

   const table& get_data() const;
   infer_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class infer_input

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Constructors

infer_input(**const** *model*<Task> &*m* = *model*<Task>{}, **const** *table* &*data* = *table*{})
Creates a new instance of the class with the given `model` and `data`.

Properties

const *model*<Task> &**model** = *model*<Task>{}
An $n \times p$ table with the data to be assigned to the clusters, where each row stores one feature vector.

Getter & Setter

```
const model< Task > & get_model() const
infer_input & set_model(const model< Task > &)
```

const table &**data** = table{}
The trained K-Means model.

Getter & Setter

```
const table & get_data() const
infer_input & set_data(const table &)
```

Result

```
template <typename Task = task::by_default>
class infer_result {
public:
   infer_result();

   const table& get_labels() const;

   double get_objective_function_value() const;
};
```

template<typename **Task** = task::*by_default*>
class infer_result

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Constructors

infer_result()
Creates a new instance of the class with the default property values.

Properties

const table &**labels** = table{}
An $n \times 1$ table with assignments labels to feature vectors in the input data.

Getter & Setter

```
const table & get_labels() const
```

double **objective_function_value** = 0.0
The value of the objective function $\Phi_X(C)$, where C is defined by the corresponding `infer_input::model::centroids`.

Getter & Setter

```
double get_objective_function_value() const
```

Invariants

```
objective_function_value >= 0.0
```

Operation

template<typename **Float**, typename **Method**, typename **Task**>
infer_result<*Task*> **infer**(**const** *descriptor*<*Float*, *Method*, *Task*> &*desc*, **const** *infer_input*<*Task*> &*input*)
Runs the inference operation for K-Means clustering. For more details see `oneapi::dal::infer`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.

- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::lloyd`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the inference operation.

Preconditions

```
input.data.has_data == true
input.model.centroids.has_data == true
input.model.centroids.row_count == desc.cluster_count
input.model.centroids.column_count == input.data.column_count
```

Postconditions

```
result.labels.row_count == input.data.row_count
result.labels.column_count == 1
result.labels[i] >= 0
result.labels[i] < desc.cluster_count
```

K-Means initialization

The K-Means initialization algorithm receives n feature vectors as input and chooses k initial centroids. After initialization, K-Means algorithm uses the initialization result to partition input data into k clusters.

Operation	Computational methods	Programming Interface		
Computing	*Dense*	*compute(. . .)*	*compute_input*	*compute_result*

Mathematical formulation

Computing

Given the training set $X = \{x_1, \ldots, x_n\}$ of p-dimensional feature vectors and a positive integer k, the problem is to find a set $C = \{c_1, \ldots, c_k\}$ of p-dimensional initial centroids.

Computing method: *dense*

The method chooses first k feature vectors from the training set X.

Usage example

Computing

```
table run_compute(const table& data) {
   const auto kmeans_desc = kmeans_init::descriptor<float,
                                                    kmeans_init::method::dense>{}
      .set_cluster_count(10)

   const auto result = compute(kmeans_desc, data);

   print_table("centroids", result.get_centroids());

   return result.get_centroids();
}
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::kmeans_init` namespace and be available via inclusion of the `oneapi/dal/algo/kmeans_init.hpp` header file.

Descriptor

```
template <typename Float = float,
          typename Method = method::by_default,
          typename Task = task::by_default>
class descriptor {
public:

   explicit descriptor(std::int64_t cluster_count = 2);

   std::int64_t get_cluster_count() const;
   descriptor& set_cluster_count(std::int64_t);

};
```

template<typename **Float** = float, typename **Method** = method::*by_default*, typename **Task** = task::*by_default*>
class descriptor

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of K-Means Initialization algorithm.
- **Task** – Tag-type that specifies the type of the problem to solve. Can be `task::init`.

Constructors

descriptor(std::int64_t *cluster_count* = 2)
Creates a new instance of the class with the given `cluster_count`.

Properties

std::int64_t **cluster_count** = 2
The number of clusters k.

Getter & Setter

```
std::int64_t get_cluster_count() const
descriptor & set_cluster_count(std::int64_t)
```

Invariants

```
cluster_count > 0
```

Method tags

```
namespace method {
   struct dense {};
   using by_default = dense;
} // namespace method
```

struct dense
Tag-type that denotes *dense* computational method.

using by_default = *dense*

Task tags

```
namespace task {
   struct init {};
   using by_default = init;
} // namespace task
```

struct init
Tag-type that parameterizes entities used for obtaining the initial K-Means centroids.

using by_default = *init*
Alias tag-type for the initialization task.

Computing `compute...`

Input

```
template <typename Task = task::by_default>
class compute_input {
public:

   compute_input(const table& data = table{});

   const table& get_data() const;
   compute_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class compute_input

Template Parameters **`Task`** – Tag-type that specifies type of the problem to solve. Can be `task::init`.

Constructors

`compute_input`(**`const`** *table* &*data* = *table*{})
Creates a new instance of the class with the given `data`.

Properties

`const` table &**`data`** = table{}
An $n \times p$ table with the data to be clustered, where each row stores one feature vector.

Getter & Setter

```
const table & get_data() const
compute_input & set_data(const table &)
```

Result

```
template <typename Task = task::by_default>
class compute_result {
public:

   compute_result();

   const table& get_centroids() const;
};
```

template<typename **`Task`** = task::*by_default*>
`class compute_result`

Template Parameters **`Task`** – Tag-type that specifies type of the problem to solve. Can be `task::clustering`.

Constructors

`compute_result`()
Creates a new instance of the class with the default property values.

Properties

`const` table &**`centroids`** = table{}
A $k \times p$ table with the initial centroids. Each row of the table stores one centroid.

Getter & Setter

```
const table & get_centroids() const
```

Operation

template<typename **`Float`**, typename **`Method`**, typename **`Task`**>
compute_result<*Task*> **`compute`**(**`const`** *descriptor*<*Float*, *Method*, *Task*> &*desc*, **`const`** *compute_input*<*Task*> &*input*)
Runs the computing operation for K-Means initialization. For more details, see `oneapi::dal::compute`.

Template Parameters

- **`Float`** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.

- **Method** – Tag-type that specifies an implementation of K-Means Initialization algorithm.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::init`.

Parameters

- **desc** – The descriptor of the algorithm.
- **input** – Input data for the computing operation.

Preconditions

```
input.data.has_data == true
input.data.row_count == desc.cluster_count
```

Postconditions

```
result.centroids.has_data == true
result.centroids.row_count == desc.cluster_count
result.centroids.column_count == input.data.column_count
```

8.7.2 Nearest Neighbors (kNN)

k-Nearest Neighbors Classification (k-NN)

k-NN *classification* algorithm infers the class for the new feature vector by computing majority vote of the k nearest observations from the training set.

Operation	Computational methods		Programming Interface		
Training	*Brute-force*	*k-d tree*	*train(...)*	*train_input*	*train_result*
Inference	*Brute-force*	*k-d tree*	*infer(...)*	*infer_input*	*infer_result*

Mathematical formulation

Training

Let $X = \{x_1, \ldots, x_n\}$ be the training set of p-dimensional feature vectors, let $Y = \{y_1, \ldots, y_n\}$ be the set of class labels, where $y_i \in \{0, \ldots, c-1\}$, $1 \leq i \leq n$. Given X, Y and the number of nearest neighbors k, the problem is to build a model that allows distance computation between the feature vectors in training and inference sets at the inference stage.

Training method: *brute-force*

The training operation produces the model that stores all the feature vectors from the initial training set X.

Training method: *k-d tree*

The training operation builds a k-d tree that partitions the training set X (for more details, see *k-d Tree*).

Inference

Let $X' = \{x'_1, \ldots, x'_m\}$ be the inference set of p-dimensional feature vectors. Given X', the model produced at the training stage and the number of nearest neighbors k, the problem is to predict the label y'_j for each x'_j, $1 \le j \le m$, by performing the following steps:

1. Identify the set $N(x'_j) \subseteq X$ of the k feature vectors in the training set that are nearest to x'_j with respect to the Euclidean distance.
2. Estimate the conditional probability for the l-th class as the fraction of vectors in $N(x'_j)$ whose labels y_j are equal to l:

$$P_{jl} = \frac{1}{|N(x'_j)|}\Big|\big\{x_r \in N(x'_j) : y_r = l\big\}\Big|, \quad 1 \le j \le m,\ 0 \le l < c. \tag{8.1}$$

3. Predict the class that has the highest probability for the feature vector x'_j:

$$y'_j = \arg\max_{0 \le l < c} P_{jl}, \quad 1 \le j \le m. \tag{8.2}$$

Inference method: *brute-force*

Brute-force inference method determines the set $N(x'_j)$ of the nearest feature vectors by iterating over all the pairs (x'_j, x_i) in the implementation defined order, $1 \le i \le n$, $1 \le j \le m$. The final prediction is computed according to the equations (8.1) and (8.2).

Inference method: *k-d tree*

K-d tree inference method traverses the k-d tree to find feature vectors associated with a leaf node that are closest to x'_j, $1 \le j \le m$. The set $\tilde{n}(x'_j)$ of the currently-known nearest k-th neighbors is progressively updated during tree traversal. The search algorithm limits exploration of the nodes for which the distance between the x'_j and respective part of the feature space is not less than the distance between x'_j and the most distant feature vector from $\tilde{n}(x'_j)$. Once tree traversal is finished, $\tilde{n}(x'_j) \equiv N(x'_j)$. The final prediction is computed according to the equations (8.1) and (8.2).

Usage example

Training

```
knn::model<> run_training(const table& data,
                          const table& labels) {
   const std::int64_t class_count = 10;
   const std::int64_t neighbor_count = 5;
   const auto knn_desc = knn::descriptor<float>{class_count, neighbor_count};

   const auto result = train(knn_desc, data, labels);

   return result.get_model();
}
```

Inference

```
table run_inference(const knn::model<>& model,
                    const table& new_data) {
   const std::int64_t class_count = 10;
   const std::int64_t neighbor_count = 5;
   const auto knn_desc = knn::descriptor<float>{class_count, neighbor_count};

   const auto result = infer(knn_desc, model, new_data);

   print_table("labels", result.get_labels());
}
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::knn` namespace and be available via inclusion of the `oneapi/dal/algo/knn.hpp` header file.

Descriptor

```
template <typename Float = float,
          typename Method = method::by_default,
          typename Task = task::by_default>
class descriptor {
public:
   explicit descriptor(std::int64_t class_count,
                       std::int64_t neighbor_count);

   std::int64_t get_class_count() const;
   descriptor& set_class_count(std::int64_t);

   std::int64_t get_neighbor_count() const;
   descriptor& set_neighbor_count(std::int64_t);
};
```

template<typename **`Float`** = float, typename **`Method`** = method::*by_default*, typename **`Task`** = task::*by_default*>
`class descriptor`

Template Parameters

- **`Float`** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **`Method`** – Tag-type that specifies an implementation of algorithm. Can be `method::bruteforce` or `method::kd_tree`.
- **`Task`** – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Constructors

`descriptor`(std::int64_t *class_count*, std::int64_t *neighbor_count*)
Creates a new instance of the class with the given `class_count` and `neighbor_count` property values.

Properties

std::int64_t **class_count**
: The number of classes c.

Getter & Setter

```
std::int64_t get_class_count() const
descriptor & set_class_count(std::int64_t)
```

Invariants

```
class_count > 1
```

std::int64_t **neighbor_count**
: The number of neighbors k.

Getter & Setter

```
std::int64_t get_neighbor_count() const
descriptor & set_neighbor_count(std::int64_t)
```

Invariants

```
neighbor_count > 0
```

Method tags

```
namespace method {
   struct bruteforce {};
   struct kd_tree {};
   using by_default = bruteforce;
} // namespace method
```

struct bruteforce
: Tag-type that denotes *brute-force* computational method.

struct kd_tree
: Tag-type that denotes k-d tree computational method.

using by_default = *bruteforce*
: Alias tag-type for *brute-force* computational method.

Task tags

```
namespace task {
   struct classification {};
   using by_default = classification;
} // namespace task
```

struct classification
: Tag-type that parameterizes entities used for solving *classification problem*.

using by_default = *classification*
: Alias tag-type for classification task.

Model

```
template <typename Task = task::by_default>
class model {
public:
   model();
};
```

template<typename **Task** = task::*by_default*>
class model

> **Template Parameters** **Task** – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Constructors

model()
> Creates a new instance of the class with the default property values.

Training `train...`

Input

```
template <typename Task = task::by_default>
class train_input {
public:
   train_input(const table& data = table{},
               const table& labels = table{});

   const table& get_data() const;
   train_input& set_data(const table&);

   const table& get_labels() const;
   train_input& set_labels(const table&);
};
```

template<typename **Task** = task::*by_default*>
class train_input

> **Template Parameters** **Task** – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Constructors

train_input(**const** *table* &*data* = *table*{}, **const** *table* &*labels* = *table*{})
> Creates a new instance of the class with the given `data` and `labels` property values.

Properties

const table &**data** = table{}
> The training set X.

Getter & Setter

```
const table & get_data() const
train_input & set_data(const table &)
```

const table &**labels** = table{}
> Vector of labels y for the training set X.

Getter & Setter

```
const table & get_labels() const
train_input & set_labels(const table &)
```

Result

```
template <typename Task = task::by_default>
class train_result {
public:
   train_result();

   const model<Task>& get_model() const;
};
```

template<typename **Task** = task::*by_default*>
class train_result

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Constructors

train_result()
Creates a new instance of the class with the default property values.

Properties

const *model*<Task> &**model** = *model*<Task>{}
The trained k-NN model.

Getter & Setter

```
const model< Task > & get_model() const
```

Operation

template<typename **Float**, typename **Method**, typename **Task**>
train_result<*Task*> **train**(**const** *descriptor*<*Float*, *Method*, *Task*> &*desc*, **const** *train_input*<*Task*> &*input*)
Runs the training operation for k-NN classifier. For more details see `oneapi::dal::train`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::bruteforce` or `method::kd_tree`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the training operation.

Preconditions

```
input.data.has_data == true
input.labels.has_data == true
input.data.row_count == input.labels.row_count
input.labels.column_count == 1
input.labels[i] >= 0
input.labels[i] < desc.class_count
```

Inference `infer...`

Input

```
template <typename Task = task::by_default>
class infer_input {
public:
   infer_input(const model<Task>& m = model<Task>{},
               const table& data = table{});

   const model<Task>& get_model() const;
   infer_input& set_model(const model&);

   const table& get_data() const;
   infer_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class infer_input

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Constructors

infer_input(**const** *model*<Task> &*m* = *model*<Task>{}, **const** *table* &*data* = *table*{})
Creates a new instance of the class with the given `model` and `data` property values.

Properties

const *model*<Task> &**model** = *model*<Task>{}
The trained k-NN model.

Getter & Setter

```
const model< Task > & get_model() const
infer_input & set_model(const model &)
```

const table &**data** = table{}
The dataset for inference X'.

Getter & Setter

```
const table & get_data() const
infer_input & set_data(const table &)
```

Result

```
template <typename Task = task::by_default>
class infer_result {
public:
   infer_result();

   const table& get_labels() const;
};
```

template<typename **Task** = task::*by_default*>
class infer_result

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Constructors

infer_result()
Creates a new instance of the class with the default property values.

Properties

const table &**labels** = table{}
The predicted labels.

Getter & Setter

```
const table & get_labels() const
```

Operation

template<typename **Float**, typename **Method**, typename **Task**>
infer_result<*Task*> **infer**(**const** *descriptor*<*Float*, *Method*, *Task*> &*desc*, **const** *infer_input*<*Task*> &*input*)
Runs the inference operation for k-NN classifier. For more details see `oneapi::dal::infer`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::bruteforce` or `method::kd_tree`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::classification`.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the inference operation.

Preconditions

```
input.data.has_data == true
```

Postconditions

```
result.labels.row_count == input.data.row_count
```

```
result.labels.column_count == 1
result.labels[i] >= 0
result.labels[i] < desc.class_count
```

8.7.3 Decomposition

Principal Components Analysis (PCA)

Principal Component Analysis (PCA) is an algorithm for exploratory data analysis and *dimensionality reduction*. PCA transforms a set of feature vectors of possibly correlated features to a new set of uncorrelated features, called principal components. Principal components are the directions of the largest variance, that is, the directions where the data is mostly spread out.

Operation	**Computational methods**		**Programming Interface**		
Training	*Covariance*	*SVD*	*train(...)*	*train_input*	*train_result*
Inference	*Covariance*	*SVD*	*infer(...)*	*infer_input*	*infer_result*

Mathematical formulation

Training

Given the training set $X = \{x_1, \ldots, x_n\}$ of p-dimensional feature vectors and the number of principal components r, the problem is to compute r principal directions (p-dimensional eigenvectors [Lang87]) for the training set. The eigenvectors can be grouped into the $r \times p$ matrix T that contains one eigenvector in each row.

Training method: *Covariance*

This method uses eigenvalue decomposition of the covariance matrix to compute the principal components of the datasets. The method relies on the following steps:

1. Computation of the covariance matrix
2. Computation of the eigenvectors and eigenvalues
3. Formation of the matrices storing the results

Covariance matrix computation shall be performed in the following way:

1. Compute the vector-column of sums $s_i = \sum_{j=1}^{n} x_{i,j}, \quad 1 \leq i \leq p$.
2. Compute the cross-product $P = X^T X - s^T s$.
3. Compute the covariance matrix $\Sigma = \frac{1}{n-1} P$.

To compute eigenvalues λ_i and eigenvectors υ_i, the implementer can choose an arbitrary method such as [Ping14].

The final step is to sort the set of pairs (λ_i, υ_i) in the descending order by λ_i and form the resulting matrix $T = (\upsilon_{i,1}, \cdots, \upsilon_{i,r}), \quad 1 \leq i \leq p$. Additionally, the means and variances of the initial dataset shall be returned.

Training method: *SVD*

This method uses singular value decomposition of the dataset to compute its principal components. The method relies on the following steps:

1. Computation of the singular values and singular vectors
2. Formation of the matrices storing the results

To compute singular values λ_i and singular vectors u_i and v_i, the implementer can choose an arbitrary method such as [Demmel90].

The final step is to sort the set of pairs (λ_i, v_i) in the descending order by λ_i and form the resulting matrix $T = (v_{i,1}, \cdots, v_{i,r}), \quad 1 \leq i \leq p$. Additionally, the means and variances of the initial dataset shall be returned.

Sign-flip technique

Eigenvectors computed by some eigenvalue solvers are not uniquely defined due to sign ambiguity. To get the deterministic result, a sign-flip technique should be applied. One of the sign-flip techniques proposed in [Bro07] requires the following modification of matrix T:

$$\hat{T_i} = T_i \cdot \operatorname{sgn}(\max_{1 \leq j \leq p} |T_{ij}|), \quad 1 \leq i \leq r,$$

where T_i is i-th row, T_{ij} is the element in the i-th row and j-th column, $\operatorname{sgn}(\cdot)$ is the signum function,

$$\operatorname{sgn}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$$

Note: The sign-flip technique described above is an example. oneDAL spec does not require implementation of this sign-flip technique. Implementer can choose an arbitrary technique that modifies the eigenvectors' signs.

Inference

Given the inference set $X' = \{x'_1, \ldots, x'_m\}$ of p-dimensional feature vectors and the $r \times p$ matrix T produced at the training stage, the problem is to transform X' to the set $X'' = \{x''_1, \ldots, x''_m\}$, where x''_j is an r-dimensional feature vector, $1 \leq j \leq m$.

The feature vector x''_j is computed through applying linear transformation [Lang87] defined by the matrix T to the feature vector x'_j,

$$x''_j = T x'_j, \quad 1 \leq j \leq m. \tag{8.3}$$

Inference methods: *Covariance* and *SVD*

Covariance and SVD inference methods compute x''_j according to (8.3).

Usage example

Training

```
pca::model<> run_training(const table& data) {
   const auto pca_desc = pca::descriptor<float>{}
      .set_component_count(5)
      .set_deterministic(true);

   const auto result = train(pca_desc, data);

   print_table("means", result.get_means());
   print_table("variances", result.get_variances());
   print_table("eigenvalues", result.get_eigenvalues());
   print_table("eigenvectors", result.get_eigenvectors());

   return result.get_model();
}
```

Inference

```
table run_inference(const pca::model<>& model,
                    const table& new_data) {
   const auto pca_desc = pca::descriptor<float>{}
      .set_component_count(model.get_component_count());

   const auto result = infer(pca_desc, model, new_data);

   print_table("labels", result.get_transformed_data());
}
```

Programming Interface

All types and functions in this section shall be declared in the `oneapi::dal::pca` namespace and be available via inclusion of the `oneapi/dal/algo/pca.hpp` header file.

Descriptor

```
template <typename Float = float,
          typename Method = method::by_default,
          typename Task = task::by_default>
class descriptor {
public:
   explicit descriptor(std::int64_t component_count = 0);

   int64_t get_component_count() const;
   descriptor& set_component_count(int64_t);

   bool get_deterministic() const;
   descriptor& set_deterministic(bool);
};
```

template<typename **Float** = float, typename **Method** = method::*by_default*, typename **Task** = task::*by_default*>
class descriptor

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::cov` or `method::svd`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

descriptor(std::int64_t *component_count* = 0)
Creates a new instance of the class with the given `component_count` property value.

Properties

int64_t **component_count** = 0
The number of principal components r. If it is zero, the algorithm computes the eigenvectors for all features, $r = p$.

Getter & Setter

```
int64_t get_component_count() const
descriptor & set_component_count(int64_t)
```

Invariants

```
component_count >= 0
```

bool **deterministic** = true
Specifies whether the algorithm applies the *Sign-flip technique*. If it is *true*, the directions of the eigenvectors must be deterministic.

Getter & Setter

```
bool get_deterministic() const
descriptor & set_deterministic(bool)
```

Method tags

```
namespace method {
   struct cov {};
   struct svd {};
   using by_default = cov;
} // namespace method
```

struct cov
Tag-type that denotes *Covariance* computational method.

struct svd
Tag-type that denotes *SVD* computational method.

using by_default = *cov*
Alias tag-type for *Covariance* computational method.

Task tags

```
namespace task {
   struct dim_reduction {};
   using by_default = dim_reduction;
} // namespace task
```

struct dim_reduction
Tag-type that parameterizes entities used for solving *dimensionality reduction problem*.

using by_default = *dim_reduction*
Alias tag-type for dimensionality reduction task.

Model

```
template <typename Task = task::by_default>
class model {
public:
   model();

   const table& get_eigenvectors() const;

   int64_t get_component_count() const;
};
```

template<typename **Task** = task::*by_default*>
class model

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

model()
Creates a new instance of the class with the default property values.

Properties

const table &**eigenvectors** = table{}
An $r \times p$ table with the eigenvectors. Each row contains one eigenvector.

Getter & Setter

```
const table & get_eigenvectors() const
```

int64_t **component_count** = 0
The number of components r in the trained model.

Getter & Setter

```
int64_t get_component_count() const
```

Invariants

```
component_count == eigenvectors.row_count
```

Training `train...`

Input

```
template <typename Task = task::by_default>
class train_input {
public:
   train_input(const table& data = table{});

   const table& get_data() const;
   train_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class train_input

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

train_input(**const** *table* &*data* = *table*{})
Creates a new instance of the class with the given `data` property value.

Properties

const table &**data** = table{}
An $n \times p$ table with the training data, where each row stores one feature vector.

Getter & Setter

```
const table & get_data() const
train_input & set_data(const table &)
```

Result

```
template <typename Task = task::by_default>
class train_result {
public:
   train_result();

   const model<Task>& get_model() const;

   const table& get_means() const;

   const table& get_variances() const;

   const table& get_eigenvalues() const;

   const table& get_eigenvectors() const;
};
```

template<typename **Task** = task::*by_default*>
class train_result

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

train_result()
Creates a new instance of the class with the default property values.

Properties

const *model*<Task> &**model** = *model*<Task>{}
The trained PCA model.

Getter & Setter

```
const model< Task > & get_model() const
```

const table &**means** = table{}
A $1 \times r$ table that contains the mean values for the first r features.

Getter & Setter

```
const table & get_means() const
```

const table &**variances** = table{}
A $1 \times r$ table that contains the variances for the first r features.

Getter & Setter

```
const table & get_variances() const
```

const table &**eigenvalues** = table{}
A $1 \times r$ table that contains the eigenvalues for for the first r features.

Getter & Setter

```
const table & get_eigenvalues() const
```

const table &**eigenvectors** = table{}
An $r \times p$ table with the eigenvectors. Each row contains one eigenvector.

Getter & Setter

```
const table & get_eigenvectors() const
```

Invariants

```
eigenvectors == model.eigenvectors
```

Operation

template<typename **Float**, typename **Method**, typename **Task**>
train_result<*Task*> **train**(**const** *descriptor*<*Float*, *Method*, *Task*> &*desc*, **const** *train_input*<*Task*> &*input*)
Runs the training operation for PCA. For more details, see `oneapi::dal::train`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::cov` or `method::svd`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the training operation.

Preconditions

```
input.data.has_data == true
input.data.column_count >= desc.component_count
```

Postconditions

```
result.means.row_count == 1
result.means.column_count == desc.component_count
result.variances.row_count == 1
result.variances.column_count == desc.component_count
result.variances[i] >= 0.0
result.eigenvalues.row_count == 1
result.eigenvalues.column_count == desc.component_count
result.model.eigenvectors.row_count == 1
result.model.eigenvectors.column_count == desc.component_count
```

Inference `infer...`

Input

```
template <typename Task = task::by_default>
class infer_input {
public:
   infer_input(const model<Task>& m = model<Task>{},
               const table& data = table{});

   const model<Task>& get_model() const;
   infer_input& set_model(const model&);

   const table& get_data() const;
   infer_input& set_data(const table&);
};
```

template<typename **Task** = task::*by_default*>
class infer_input

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

infer_input(**const** *model*<Task> &*m* = *model*<Task>{}, **const** *table* &*data* = *table*{})
Creates a new instance of the class with the given `model` and `data` property values.

Properties

const *model*<Task> &**model** = *model*<Task>{}
The trained PCA model.

Getter & Setter

```
const model< Task > & get_model() const
```

```
infer_input & set_model(const model &)
```

const table &**data** = table{}
The dataset for inference X'.

Getter & Setter

```
const table & get_data() const
infer_input & set_data(const table &)
```

Result

```
template <typename Task = task::by_default>
class infer_result {
public:
   infer_result();

   const table& get_transformed_data() const;
};
```

template<typename **Task** = task::*by_default*>
class infer_result

Template Parameters **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Constructors

infer_result()
Creates a new instance of the class with the default property values.

Properties

const table &**transformed_data** = table{}
An $n \times r$ table that contains data projected to the r principal components.

Getter & Setter

```
const table & get_transformed_data() const
```

Operation

template<typename **Float**, typename **Method**, typename **Task**>
infer_result<*Task*> **infer**(**const** *descriptor*<*Float*, *Method*, *Task*> &*desc*, **const** *infer_input*<*Task*> &*input*)
Runs the inference operation for PCA. For more details see `oneapi::dal::infer`.

Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::cov` or `method::svd`.
- **Task** – Tag-type that specifies type of the problem to solve. Can be `task::dim_reduction`.

Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the inference operation.

Preconditions

```
input.data.has_data == true
input.model.eigenvectors.row_count == desc.component_count
input.model.eigenvectors.column_count == input.data.column_count
```

Postconditions

```
result.transformed_data.row_count == input.data.row_count
result.transformed_data.column_count == desc.component_count
```

8.8 Appendix

8.8.1 k-d Tree

k-d tree is a space-partitioning binary tree [Bentley80], where

- Each non-leaf node induces the hyperplane that splits the feature space into two parts. To define the splitting hyperplane explicitly, a non-leaf node stores the identifier of the feature (that defines axis in the feature space) and *a cut-point*
- Each leaf node of the tree has an associated subset (*a bucket*) of elements of the training data set. Feature vectors from a bucket belong to the region of the space defined by tree nodes on the path from the root node to the respective leaf.

Related terms

A cut-point A feature value that corresponds to a non-leaf node of a k-d tree and defines the splitting hyperplane orthogonal to the axis specified by the given feature.

8.9 Bibliography

For more information about algorithms implemented in oneAPI Data Analytics Library (oneDAL), refer to the following publications:

CHAPTER

NINE

ONETBB

9.1 General Information

9.1.1 Introduction

[intro]

This document specifies requirements for implementations of oneAPI Threading Building Blocks (oneTBB).

oneTBB is a programming model for scalable parallel programming using standard ISO C++ code. A program uses oneTBB to specify logical parallelism in algorithms, while a oneTBB implementation maps that parallelism onto execution threads.

oneTBB employs generic programming via C++ templates, with most of its interfaces defined by requirements on types and not specific types. Generic programming makes oneTBB flexible yet efficient through customizing APIs to specific needs of an application.

Here is the list of specific requirements for oneTBB implementations: * An implementation should use the C++11 version of the standard and should not require newer versions except where explicitly specified; it also should not require any non-standard language extensions. * An implementation can use platform-specific APIs if they are compatible with the C++ execution and memory models. For example, a platform-specific implementation of threads can be used if that implementation provides the same execution guarantees as C++ threads. * An implementation should support execution on single-core and multi-core CPUs, including those that provide simultaneous multithreading capabilities. * On CPU, an implementation should support nested parallelism to enable building larger parallel components from smaller ones.

9.1.2 Notational Conventions

[notational_conventions]

The following conventions are used in this document.

Convention	Explanation	Example
Italic	Used for introducing new terms, denotation of terms, placeholders, or titles of documents.	The filename consists of the *basename* and the *extension*. For more information, refer to the *TBB Developer Guide*.

`Monospace`	Indicates directory paths and filenames, commands and command line options, function names, methods, classes, data structures in body text, source code.	`tbb.h`
		`\alt\include`
		Use the `okCreateObjs()` function to...
		`printf("hello, world\n");`

`Monospace italic`	Indicates source code placeholders.	blocked_range<*Type*>
`Monospace bold`	Emphasizes parts of source code.	`x = ( h > 0 ? sizeof(m) : 0xF ) + min;`
[ ]	Square brackets indicate that the items enclosed in brackets are optional.	Fa[c] Indicates `Fa` or `Fac`.
{ | }	Braces and vertical bars indicate the choice of one item from a selection of two or more items.	X{K | W | P} Indicates `XK`, `XW`, or `XP`.
"[" "]" "{"" }" "|"	Writing a metacharacter in quotation marks negates the syntactical meaning stated above; the character is taken as a literal.	"[" X "]" [Y] Denotes the letter X enclosed in brackets, optionally followed by the letter Y.
...	The ellipsis indicates that the previous item can be repeated several times.	**filename** ... Indicates that one or more filenames can be specified.
,...	The ellipsis preceded by a comma indicates that the previous item can be repeated several times, separated by commas.	**word** ,... Indicates that one or more words can be specified. If more than one word is specified, the words are comma-separated.

Class members are summarized by informal class declarations that describe the class as it seems to clients, not how it is actually implemented. For example, here is an informal declaration of class `Foo`:

```
class Foo {
public:
     int x();
     int y;
     ~Foo();
};
```

The actual implementation might look like:

```
namespace internal {
     class FooBase  {
     protected:
              int x();
     };

     class Foo_v3: protected FooBase {
     private:
              int internal_stuff;
```

(continues on next page)

(continued from previous page)

```
    public:
            using FooBase::x;
            int y;
    };
}

typedef internal::Foo_v3 Foo;
```

The example shows two cases where the actual implementation departs from the informal declaration:

- `Foo` is actually a typedef to `Foo_v3`.
- Method `x()` is inherited from a protected base class.
- The destructor is an implicit method generated by the compiler.

The informal declarations are intended to show you what you need to know to use the class without the distraction of irrelevant clutter particular to the implementation.

9.1.3 Identifiers

[identifiers]

This section describes the identifier conventions used by oneTBB.

Case

The identifier convention in the library follows the style of the ISO C++ standard library. Identifiers are written in underscore_style, and concepts - in PascalCase.

Reserved Identifier Prefixes

The library reserves the `__TBB` prefix for internal identifiers and macros that should never be directly referenced by your code.

9.1.4 Named Requirements

[named_requirements]

This section describes named requirements used in the oneTBB Specification.

A *named requirement* is a set of requirements on a type. The requirements may be syntactic or semantic. The *named_requirement* term is similar to "Requirements on types and expressions" term which is defined by the ISO C++ Standard (chapter "Library Introduction") or "Named Requirements" section on the cppreference.com site.

For example, the named requirement of *sortable* could be defined as a set of requirements that enable an array to be sorted. A type `T` would be *sortable* if:

- `x < y` returns a boolean value, and represents a total order on items of type `T`.
- `swap(x,y)` swaps items `x` and `y`

You can write a sorting template function in C++ that sorts an array of any type that is *sortable*.

Two approaches for defining named requirements are *valid expressions* and *pseudo-signatures*. The ISO C++ standard follows the valid *expressions* approach, which shows what the usage pattern looks like for a requirement. It has the

drawback of relegating important details to notational conventions. This document uses pseudo-signatures because they are concise and can be cut-and-pasted for an initial implementation.

For example, the table below shows pseudo-signatures for a *sortable* type `T`:

Sortable Requirements : Pseudo-Signature, Semantics

bool **`operator<`**(**`const`** T &*x*, **`const`** T &*y*)
: Compare x and y.

void **`swap`**(T &*x*, T &*y*)
: Swap x and y.

A real signature may differ from the pseudo-signature that it implements in ways where implicit conversions would deal with the difference. For an example type `U`, the real signature that implements `operator<` in the table above can be expressed as `int operator<( U x, U y )`, because C++ permits implicit conversion from `int` to `bool`, and implicit conversion from `U` to (`const U&`). Similarly, the real signature `bool operator<( U& x, U& y )` is acceptable because C++ permits implicit addition of a const qualifier to a reference type.

Algorithms

Range

[req.range]

A *Range* can be recursively subdivided into two parts. Subdivision is done by calling *splitting constructor* of a *Range*. There are two types of splitting constructors:

- Basic splitting constructor. In this constructor, it is recommended that the division is done into nearly equal parts, but it is not required. Splitting as evenly as possible typically yields the best parallelism.
- Proportional splitting constructor. This constructor is optional and can be omitted. When using this type of constructor, for the best results, follow the given proportion with rounding to the nearest integer if necessary.

Ideally, a range is recursively splittable until the parts represent portions of work that are more efficient to execute serially rather than split further. The amount of work represented by *Range* typically depends on higher level context, therefore a typical type that models a *Range* should provide a way to control the degree of splitting. For example, the template class *blocked_range* has the *grainsize* parameter that specifies the biggest range considered indivisible.

If the set of values has a sense of direction, by convention the splitting constructor should construct the second part of the range and update its argument to be the first part of the range. This causes the *parallel_for*, *parallel_reduce*, and *parallel_scan* algorithms, when running sequentially, to work across a range in the increasing order, which is typical of an ordinary sequential loop.

Because a *Range* declares splitting and copy constructors, the default constructor for it is not generated automatically. You need to explicitly define the default constructor or add any other constructor to create an instance of a *Range* type in the program.

A type *R* meets *Range* if it satisfies the following requirements:

Range Requirements: Pseudo-Signature, Semantics

R`::`**`R`**(**`const`** *R*&)
: Copy constructor.

`R::~R()`
Destructor.

bool `R::empty() const`
True if range is empty.

bool `R::is_divisible() const`
True if range can be partitioned into two subranges.

R`::R(`*R &r*, split`)`
Basic splitting constructor. Splits `r` into two subranges.

R`::R(`*R &r*, *proportional_split proportion*`)`
Optional. Proportional splitting constructor. Splits `r` into two subranges in accordance with `proportion`.

See also:

- *blocked_range class*
- *blocked_range2d class*
- *blocked_range3d class*
- *parallel_reduce algorithm*
- *parallel_for algorithm*
- *split class*

Splittable

[req.splittable]

A type is splittable if it has a *splitting constructor* that allows an instance to be split into two pieces. The splitting constructor takes as arguments a reference to the original object, and a dummy argument of type `split`, which is defined by the library. The dummy argument distinguishes the splitting constructor from a copy constructor. After the constructor runs, `x` and the newly constructed object should represent the two pieces of the original `x`. The library uses splitting constructors in two contexts:

- *Partitioning* a range into two subranges that can be processed concurrently.
- *Forking* a body (function object) into two bodies that can run concurrently.

Types that meet the *Range requirements* may additionally define an optional *proportional splitting constructor*, distinguished by an argument of type *proportional_split Class*.

A type *X* satisfies *Splittable* if it meets the following requirements:

Splittable Requirements: Pseudo-Signature, Semantics

X`::X(`*X &x*, split`)`
Split `x` into `x` and newly constructed object.

See also:

- *Range requirements*

ParallelForBody

[req.parallel_for_body]

A type *Body* satisfies *ParallelForBody* if it meets the following requirements:

ParallelForBody Requirements: Pseudo-Signature, Semantics

Body`::Body(const` *Body*`&)`
Copy constructor.

`Body::~Body()`
Destructor.

void `Body::operator()`(Range &*range*) `const`
Applies body to a range. `Range` type must meet the *Range requirements*.

See also:

- *parallel_for algorithm*

ParallelForFunc

[req.parallel_for_func]

A type *F* satisfies *ParallelForFunc* if it meets the following requirements:

ParallelForFunc Requirements: Pseudo-Signature, Semantics

void `F::operator()`(Index *index*) `const`
Applies the function to the index. `Index` type must be the same as corresponding template parameter of the *parallel_for algorithm*.

See also:

- *parallel_for algorithm*
- *ParallelForIndex named requirement*

ParallelForIndex

[req.parallel_for_index]

A type *Index* satisfies *ParallelForIndex* if it meets the following requirements:

ParallelForIndex Requirements: Pseudo-Signature, Semantics

Index`::Index(int)`
Constructor from an `int` value.

Index`::Index(const` *Index*`&)`
Copy constructor.

`Index::~Index()`
Destructor.

void **operator=**(**const** Index&)
Assignment.

Note: The return type `void` in the pseudo-signature denotes that `operator=` is not required to return a value. The actual `operator=` can return a value, which will be ignored.

---

bool **operator<**(**const** Index *&i*, **const** Index *&j*)
Value of *i* precedes value of *j*.

D **operator-**(**const** Index *&i*, **const** Index *&j*)
Number of values in range `[i, j)`.

Index **operator+**(**const** Index *&i*, D *k*)
k-th value after *i*.

`D` is the type of the expression `j-i`. It can be any integral type that is convertible to `size_t`. Examples that model the Index requirements are integral types and pointers.

See also:

- *parallel_for algorithm*

ParallelReduceBody

[req.parallel_reduce_body]

A type *Body* satisfies *ParallelReduceBody* if it meets the following requirements:

ParallelReduceBody Requirements: Pseudo-Signature, Semantics

Body`::`**Body**(*Body*&, split)
Splitting constructor. Must be able to run concurrently with `operator()` and method `join`.

`Body::`**~Body**()
Destructor.

void `Body::`**operator()**(**const** Range *&range*)
Accumulates result for a subrange. `Range` type must meet the *Range requirements*.

void `Body::`**join**(Body *&rhs*)
Joins results. The result in rhs should be merged into the result of `this`.

See also:

- *parallel_reduce algorithm*
- *parallel_determinstic_reduce algorithm*

ParallelReduceFunc

[req.parallel_reduce_body]

A type *Func* satisfies *ParallelReduceFunc* if it meets the following requirements:

ParallelReduceFunc Requirements: Pseudo-Signature, Semantics

Value `Func::operator()` (**const** Range &*range*, **const** Value &*x*) **const**
: Accumulates result for a subrange, starting with initial value `x`. `Range` type must meet the *Range requirements*. `Value` type must be the same as a corresponding template parameter for the *parallel_reduce algorithm* algorithm.

See also:

- *parallel_reduce algorithm*
- *parallel_determinstic_reduce algorithm*

ParallelReduceReduction

[req.parallel_reduce_reduction]

A type *Reduction* satisfies *ParallelReduceReduction* if it meets the following requirements:

ParallelReduceReduction Requirements: Pseudo-Signature, Semantics

Value `Reduction::operator()` (**const** Value &*x*, **const** Value &*y*) **const**
: Combines results `x` and `y`. `Value` type must be the same as a corresponding template parameter for the *parallel_reduce algorithm* algorithm.

See also:

- *parallel_reduce algorithm*
- *parallel_determinstic_reduce algorithm*

ParallelForEachBody

[req.parallel_for_each_body]

A type *Body* satisfies *ParallelForBody* if it meets the *Function Objects* requirements from the [function.objects] ISO C++ Standard section. It should also meet one of the following requirements:

ParallelForEachBody Requirements: Pseudo-Signature, Semantics

`Body::operator()` (ItemType *item*) **const**
: Process the received item.

`Body::operator()` (ItemType *item*, tbb::feeder<ItemType> &*feeder*) **const**
: Process the received item. May invoke the `feeder.add(T)` function to spawn additional items.

Note: `ItemType` may be optionally passed to `Body::operator()` by reference. `const` and `volatile` type qualifiers are also applicable.

ItemType

The argument type `ItemType` should either satisfy the *CopyConstructibe*, *MoveConstructibe* or both requirements from the ISO C++ [utility.arg.requirements] section. If the type is not *CopyConstructibe*, there are additional usage restrictions:

- If `Body::operator()` accepts an argument by value, or if the `InputIterator` type from *parallel_for_each algorithm* does not satisfy the *Forward Iterator* requirements from the [forward.iterators] ISO C++ Standard section, dereferencing an `InputIterator` must produce an rvalue reference.
- Additional work items should be passed to the feeder as rvalues, for example, via the `std::move` function.

See also:

- *parallel_for_each algorithm*
- *feeder class*

ContainerBasedSequence

[req.container_based_sequence]

A type *C* satisfies *ContainerBasedSequence* if it meets the following requirements:

ContainerBasedSequence Requirements: Pseudo-Signature, Semantics

> **Note:** In this page `c` is an object of type (possibly `const`) `C`.
>
> Templates that use the named requirement can impose stricter requirements on the iterator concept.

`std::`**`begin`**`(c)`
: Returns an input iterator to the beginning of the sequence represented by `c`.

`std::`**`end`**`(c)`
: Returns an input iterator one past the end of the sequence represented by `c`.

See also:

- *parallel_for_each algorithm*
- *parallel_sort algorithm*

ParallelScanBody

[req.parallel_scan]

A type *Body* satisfies *ParallelScanBody* if it meets the following requirements:

ParallelScanBody Requirements: Pseudo-Signature, Semantics

void `Body::`**`operator()`**`(`**`const`** Range &*r*, pre_scan_tag)
: Accumulates summary for range `r`. For example, when computing a running sum of an array, the summary for a range `r` is the sum of the array elements corresponding to `r`.

void `Body::operator()` (**const** Range &*r*, final_scan_tag)
: Computes scan result and summary for range `r`.

`Body``::Body` (*Body* &*b*, split)
: Splits `b` so that `this` and `b` can accumulate summaries separately.

void `Body::reverse_join` (Body &*b*)
: Merges the summary accumulated by `b` into the summary accumulated by `this`, where `this` was created earlier from `b` by splitting constructor.

void `Body::assign` (Body &*b*)
: Assigns summary of `b` to `this`.

See also:

- *parallel_scan algorithm*

ParallelScanCombine

[req.parallel_scan_combine]

A type *Combine* satisfies *ParallelScanCombine* if it meets the following requirements:

ParallelScanCombine Requirements: Pseudo-Signature, Semantics

Value `Combine::operator()` (**const** Value &*left*, **const** Value &*right*) **const**
: Combines summaries `left` and `right` and returns the result `Value` type must be the same as a corresponding template parameter for the `parallel_scan` algorithm.

See also:

- *parallel_scan algorithm*

ParallelScanFunc

[req.parallel_scan_func]

A type *Scan* satisfies *ParallelScanFunc* if it meets the following requirements:

ParallelScanFunc Requirements: Pseudo-Signature, Semantics

Value `Scan::operator()` (**const** Range &*r*, **const** Value &*sum*, bool *is_final*) **const**
: Starting with `sum`, computes the summary and, for `is_final == true`, the scan result for range `r`. Returns the computed summary. `Value` type must be the same as a corresponding template parameter for the `parallel_scan` algorithm.

See also:

- *parallel_scan algorithm*

BlockedRangeValue

[req.blocked_range_value]

A type *Value* satisfies *BlockedRangeValue* if it meets the following requirements:

BlockedRangeValue Requirements: Pseudo-Signature, Semantics

Value`::`**`Value`**`(`**`const`** *Value*`&)`
Copy constructor.

`Value::~Value()`
Destructor.

void **`operator=`**`(`**`const`** Value`&)`
Assignment.

> **Note:** The return type `void` in the pseudo-signature denotes that `operator=` is not required to return a value. The actual `operator=` can return a value, which will be ignored by `blocked_range` .

bool **`operator<`**`(`**`const`** Value *&i*, **`const`** Value *&j*)
Value *i* precedes value *j*.

D **`operator-`**`(`**`const`** Value *&i*, **`const`** Value *&j*)
Number of values in range `[i, j)`.

Value **`operator+`**`(`**`const`** Value *&i*, D *k*)
k-th value after *i*.

`D` is the type of the expression `j-i`. It can be any integral type that is convertible to `size_t`. Examples that model the Value requirements are integral types, pointers, and STL random-access iterators whose difference can be implicitly converted to a `size_t`.

See also:

- *blocked_range class*
- *blocked_range2d class*
- *blocked_range3d class*
- *parallel_reduce algorithm*
- *parallel_for algorithm*

FilterBody

[req.filter_body]

A type *Body* should meet one of the following requirements depending on the filter type:

MiddleFilterBody Requirements: Pseudo-Signature, Semantics

OutputType `Body::`**`operator()`**`(`InputType *item*`)` **`const`**
Processes the received item and then returns it.

FirstFilterBody Requirements: Pseudo-Signature, Semantics

OutputType `Body::operator()` (tbb::flow_control *fc*) **const**
: Returns the next item from an input stream. Calls `fc.stop()` at the end of an input stream.

LastFilterBody Requirements: Pseudo-Signature, Semantics

void `Body::operator()` (InputType *item*) **const**
: Processes the received item.

SingleFilterBody Requirements: Pseudo-Signature, Semantics

void `Body::operator()` (tbb::flow_control *fc*) **const**
: Processes an element from an input stream. Calls `fc.stop()` at the end of an input stream.

See also:

- *filter class*

Mutexes

Mutex

[req.mutex]

The mutexes and locks have relatively spartan interfaces that are designed for high performance. The interfaces enforce the *scoped locking pattern*, which is widely used in C++ libraries because:

- Does not require to remember to release the lock
- Releases the lock if an exception is thrown out of the mutual exclusion region protected by the lock

There are two parts of the pattern: a *mutex* object, for which construction of a *lock* object acquires a lock on the mutex and destruction of the *lock* object releases the lock. Here is an example:

```
{
    // Construction of myLock acquires lock on myMutex
    M::scoped_lock myLock( myMutex );
    // ... actions to be performed while holding the lock ...
    // Destruction of myLock releases lock on myMutex
}
```

If the actions throw an exception, the lock is automatically released as the block is exited.

```
class M {
    // Implementation specifics
    // ...

    // Represents acquisition of a mutex
    class scoped_lock {
    public:
        constexpr scoped_lock() noexcept;
        scoped_lock(M& m);
        ~scoped_lock();
```

(continues on next page)

(continued from previous page)

```
        scoped_lock(const scoped_lock&) = delete;
        scoped_lock& operator=(const scoped_lock&) = delete;

        void acquire(M& m);
        bool try_acquire(M& m);
        void release();
    };
};
```

A type *M* satisfies the *Mutex* requirements if it meets the following conditions:

type `M::`**scoped_lock**
: Corresponding scoped lock type.

`M::`**scoped_lock**`()`
: Constructs `scoped_lock` without acquiring mutex.

`M::`**scoped_lock**`(M&)`
: Constructs `scoped_lock` and acquire the lock on a provided mutex.

`M::`**~scoped_lock**`()`
: Releases a lock (if acquired).

void `M::`*scoped_lock*`::`**acquire**`(M&)`
: Acquires a lock on a provided mutex.

bool `M::`*scoped_lock*`::`**try_acquire**`(M&)`
: Attempts to acquire a lock on a provided mutex. Returns true if the lock is acquired, false otherwise.

void `M::`*scoped_lock*`::`**release**`()`
: Releases an acquired lock.

Also, the `Mutex` type requires a set of traits to be defined:

static constexpr bool `M::`**is_rw_mutex**
: True if mutex is a reader-writer mutex; false, otherwise.

static constexpr bool `M::`**is_recursive_mutex**
: True if mutex is a recursive mutex; false, otherwise.

static constexpr bool `M::`**is_fair_mutex**
: True if mutex is fair; false, otherwise.

A mutex type and an `M::scoped_lock` type are neither copyable nor movable.

The following table summarizes the library classes that model the `Mutex` requirement and provided guarantees.

Table 1: Provided guarantees for Mutexes that model the Mutex requirement

.	Fair	Reentrant
`spin_mutex`	No	No
`speculative_spin_mutex`	No	No
`queuing_mutex`	Yes	No
`null_mutex`	Yes	Yes

Note: Implementation is allowed to have an opposite guarantees (positive) in case of negative statements from the

table above.

See the *oneAPI Threading Building Blocks Developer Guide* for description of the mutex properties and the rationale for null mutexes.

See also:

- *spin_mutex*
- *speculative_spin_mutex*
- *queuing_mutex*
- *null_mutex*

ReaderWriterMutex

[req.rw_mutex]

The *ReaderWriterMutex* requirement extends the *Mutex Requirement* to include the notion of reader-writer locks. It introduces a boolean parameter `write` that specifies whether a writer lock (`write = true`) or reader lock (`write = false`) is being requested. Multiple reader locks can be held simultaneously on a *ReaderWriterMutex* if it does not have a writer lock on it. A writer lock on a *ReaderWriterMutex* excludes all other threads from holding a lock on the mutex at the same time.

```
class RWM {
    // Implementation specifics
    // ...

    // Represents acquisition of a mutex.
    class scoped_lock {
    public:
        constexpr scoped_lock() noexcept;
        scoped_lock(RWM& m, bool write = true);
        ~scoped_lock();

        scoped_lock(const scoped_lock&) = delete;
        scoped_lock& operator=(const scoped_lock&) = delete;

        void acquire(RWM& m, bool write = true);
        bool try_acquire(RWM& m, bool write = true);
        void release();

        bool upgrade_to_writer();
        bool downgrade_to_reader();
    };
};
```

A type *RWM* satisfies *ReaderWriterMutex* if it meets the following requirements. They form a superset of the *Mutex requirements*.

type `RWM::scoped_lock`
: Corresponding scoped-lock type.

`RWM::scoped_lock()`
: Constructs `scoped_lock` without acquiring any mutex.

`RWM::`**`scoped_lock`**(RWM&, bool *write* = true)
Constructs `scoped_lock` and acquires a lock on a given mutex. The lock is a writer lock if `write` is true; a reader lock otherwise.

`RWM::`**`~scoped_lock`**()
Releases a lock (if acquired).

void `RWM::`*`scoped_lock`*`::`**`acquire`**(RWM&, bool *write* = true)
Acquires a lock on a given mutex. The lock is a writer lock if `write` is true; it is a reader lock, otherwise.

bool `RWM::`*`scoped_lock`*`::`**`try_acquire`**(RWM&, bool *write* = true)
Attempts to acquire a lock on a given mutex. The lock is a writer lock if `write` is true; it is a reader lock, otherwise. Returns `true` if the lock is acquired, `false` otherwise.

`RWM::`*`scoped_lock`*`::`**`release`**()
Releases a lock. The effect is undefined if no lock is held.

bool `RWM::`*`scoped_lock`*`::`**`upgrade_to_writer`**()
Changes a reader lock to a writer lock. Returns `false` if lock was released and reacquired. Otherwise, returns `true`, including the case when the lock was already a writer lock.

bool `RWM::`*`scoped_lock`*`::`**`downgrade_to_reader`**()
Changes a writer lock to a reader lock. Returns `false` if lock was released and reacquired. Otherwise, returns `true`, including the case when the lock was already a reader lock.

Like the *Mutex* requirement, *ReaderWriterMutex* also requires a set of traits to be defined.

`static constexpr` bool `M::`**`is_rw_mutex`**
True if mutex is a reader-writer mutex; false, otherwise.

`static constexpr` bool `M::`**`is_recursive_mutex`**
True if mutex is a recursive mutex; false, otherwise.

`static constexpr` bool `M::`**`is_fair_mutex`**
True if mutex is fair; false, otherwise.

The following table summarizes the library classes that model the *ReaderWriterMutex* requirement and provided guarantees.

Table 2: Provided guarantees for Mutexes that model the ReaderWriterMutex requirement

.	Fair	Reentrant
`spin_rw_mutex`	No	No
`speculative_spin_rw_mutex`	No	No
`queuing_rw_mutex`	Yes	No
`null_rw_mutex`	Yes	Yes

Note: Implementation is allowed to have an opposite guarantees (positive) in case of negative statements from the table above.

Note: For all currently provided reader-writer mutexes,

- `is_recursive_mutex` is `false`
- `scoped_lock::downgrade_to_reader` always returns `true`

However, other implementations of the ReaderWriterMutex requirement are not required to do the same.

See also:

- *spin_rw_mutex*
- *speculative_spin_rw_mutex*
- *queuing_rw_mutex*
- *null_rw_mutex*

Containers

HashCompare

[req.hash_compare]

HashCompare is an object which is used to calculate hash code for an object and compare two objects for equality.

The type `H` satisfies `HashCompare` if it meets the following requirements:

HashCompare Requirements: Pseudo-Signature, Semantics

H`::H(const` *H*`&)`
: Copy constructor.

`H::~H()`
: Destructor.

std::size_t `H::hash(const` KeyType *&k*`) const`
: Calculates the hash for a provided key.

ReturnType `H::equal(const` KeyType *&k1*`, const` KeyType *&k2*`) const`
: Requirements:

  - The type `ReturnType` should be implicitly convertible to `bool`.

  Compares `k1` and `k2` for equality.

  If this function returns `true`, `H::hash(k1)` should be equal to `H::hash(k2)`.

ContainerRange

[req.container_range]

`ContainerRange` is a range that represents a concurrent container or a part of the container.

The `ContainerRange` object can be used to traverse the container in parallel algorithms like `parallel_for`.

The type `CR` satisfies the `ContainerRange` requirements if:

- The type `CR` meets the requirements of *Range requirements*.
- The type `CR` provides the following member types and functions:

  type `CR::value_type`
  : The type of the item in the range.

  type `CR::reference`
  : Reference type to the item in the range.

type CR::**const_reference**
Constant reference type to the item in the range.

type CR::**iterator**
Iterator type for range traversal.

type CR::**size_type**
Unsigned integer type for obtaining grain size.

type CR::**difference_type**
The type of the difference between two iterators.

iterator CR::**begin**()
Returns an iterator to the beginning of the range.

iterator CR::**end**()
Returns an iterator to the position that follows the last element in the range.

size_type CR::**grainsize**() **const**
Returns the range grain size.

Task scheduler

SuspendFunc

[req.suspend_func]

A type *Func* satisfies *SuspendFunc* if it meets the following requirements:

SuspendFunc Requirements: Pseudo-Signature, Semantics

Func::**Func**(**const** *Func*&)
Copy constructor.

void Func::**operator()**(tbb::task::suspend_point)
Body that accepts the current task execution point to resume later.

See also:

- *resumable tasks*

Flow Graph

AsyncNodeBody

[req.async_node_body]

A type *Body* satisfies *AsyncNodeBody* if it meets the following requirements:

AsyncNodeBody Requirements: Pseudo-Signature, Semantics

Body::**Body**(**const** *Body*&)
Copy constructor.

Body::**~Body**()
Destructor.

void `Body::operator()` (**const** Input &*v*, GatewayType &*gateway*)

Requirements:

- The `Input` type must be the same as the `Input` template type argument of the `async_node` instance in which the `Body` object is passed during construction.
- The `GatewayType` type must be the same as the `gateway_type` member type of the `async_node` instance in which the `Body` object is passed during construction.

The input value `v` is submitted by the flow graph to an external activity. The *gateway interface* allows the external activity to communicate with the enclosing flow graph.

ContinueNodeBody

[req.continue_node_body]

A type *Body* satisfies *ContinueNodeBody* if it meets the following requirements:

ContinueNodeBody Requirements: Pseudo-Signature, Semantics

Body`::Body` (**const** *Body*&)
Copy constructor.

`Body::~Body` ()
Destructor.

Output `Body::operator()` (**const** continue_msg &*v*)

Requirements: The type `Output` must be the same as the template type argument `Output` of the `continue_node` instance in which the `Body` object is passed during construction.

Performs operation and returns a value of type Output.

See also:

- *continue_node class*
- *continue_msg class*

GatewayType

[req.gateway_type]

A type *T* satisfies *GatewayType* if it meets the following requirements:

GatewayType Requirements: Pseudo-Signature, Semantics

bool `T::try_put` (**const** Output &*v*)

Requirements: The type `Output` must be the same as the template type argument `Output` of the corresponding `async_node` instance.

Broadcasts `v` to all successors of the corresponding `async_node` instance.

void `T::reserve_wait` ()
Notifies a flow graph that work has been submitted to an external activity.

void `T::release_wait` ()
Notifies a flow graph that work submitted to an external activity has completed.

FunctionNodeBody

[req.function_node_body]

A type *Body* satisfies *FunctionNodeBody* if it meets the following requirements:

FunctionNodeBody Requirements: Pseudo-Signature, Semantics

Body`::`**`Body`**`(`**`const`** *Body*`&)`
Copy constructor.

`Body::`**`~Body`**`()`
Destructor.

Output `Body::`**`operator()`**`(`**`const`** Input *&v*)
Requirements: The `Input` and `Output` types must be the same as the `Input` and `Output` template type arguments of the `fucntion_node` instance in which the `Body` object is passed during construction.

Performs operation on `v` and returns a value of type `Output`.

JoinNodeFunctionObject

[req.join_node_function_object]

A type *Func* satisfies *JoinNodeFunctionObject* if it meets the following requirements:

JoinNodeFunctionObject Requirements: Pseudo-Signature, Semantics

Func`::`**`Func`**`(`**`const`** *Func*`&)`
Copy constructor.

`Func::`**`~Func`**`()`
Destructor.

Key `Func::`**`operator()`**`(`**`const`** Input *&v*)
Requirements: The `Key` and `Input` types must be the same as the `K` and the corresponding element of the `OutputTuple` template arguments of the `join_node` instance to which the `Func` object is passed during construction.

Returns key to be used for hashing input messages.

InputNodeBody

[req.input_node_body]

A type *Body* satisfies *InputNodeBody* if it meets the following requirements:

InputNodeBody Requirements: Pseudo-Signature, Semantics

Body`::`**`Body`**`(`**`const`** *Body*`&)`
Copy constructor.

`Body::`**`~Body`**`()`
Destructor.

Output `Body::operator()` (tbb::flow_control &*fc*)

Requirements: The type `Output` must be the same as the template type argument `Output` of the `input_node` instance in which the `Body` object is passed during construction.

Applies body to generate the next item. Call `fc.stop()` when new element cannot be generated. Because `Output` needs to be returned, `Body` may return any valid value of `Output`, to be immediately discarded.

MultifunctionNodeBody

[req.multifunction_node_body]

A type *Body* satisfies *MultifunctionNodeBody* if it meets the following requirements:

MultifunctionNodeBody Requirements: Pseudo-Signature, Semantics

Body`::Body(const` *Body*&`)`

Copy constructor.

`Body::~Body()`

Destructor.

void `Body::operator()(const` Input &*v*, OutputPortsType &*p*`)`

Requirements:

- The `Input` type must be the same as the `Input` template type argument of the `multifunction_node` instance in which the `Body` object is passed during construction.
- The `OutputPortsType` type must be the same as the `output_ports_type` member type of the `multifunction_node` instance in which the `Body` object is passed during construction.

Performs operation on `v`. May call `try_put()` on zero or more of the output ports. May call `try_put()` on any output port multiple times.

Sequencer

[req.sequencer]

A type *S* satisfies *Sequencer* if it meets the following requirements:

Sequencer Requirements: Pseudo-Signature, Semantics

S`::S(const` *S*&`)`

Copy constructor.

`S::~S()`

Destructor.

size_t `S::operator()(const` T &*v*`)`

Requirements: The type `T` must be the same as the template type argument `T` of the `sequencer_node` instance in which the `S` object is passed during construction.

Returns the sequence number for the provided message `v`.

See also:

- *sequencer_node class*

9.1.5 Thread Safety

[thread_safety]

Unless otherwise stated, the thread safety rules for the library are as follows:

- Two threads can invoke a method or function concurrently on different objects, but not the same object.
- It is unsafe for two threads to invoke concurrently methods or functions on the same object.

Departures from this convention are noted in the classes descriptions. For example, the concurrent containers are more liberal. By their nature, they do permit some concurrent operations on the same container object.

9.2 oneTBB Interfaces

9.2.1 Configuration

[configuration]

This section describes the most general features of oneAPI Threading Building Blocks (oneTBB) such as namespaces, versioning, and macros.

Namespaces

[configuration.namespaces]

This section describes the oneTBB namespace conventions.

tbb Namespace

The `tbb` namespace contains public identifiers defined by the library that you can reference in your program.

tbb::flow Namespace

The `tbb::flow` namespace contains public identifiers defined by the library that you can reference in your program related to the flow graph feature. See *Flow Graph* for more information.

oneapi::tbb Namespace

The `tbb` namespace is a part of the top level `oneapi` namespace. Therefore, all API from the `tbb` namespace (incl. the `tbb::flow` namespace) are available in the `oneapi::tbb` namespace. The `oneapi::tbb` namespace can be considered as an alias for the `tbb` namespace:

```
namespace oneapi { namespace tbb = ::tbb; }
```

Version Information

[configuration.version_information]

oneTBB has macros, an environment variable, and a function that reveal version and runtime information.

```
// Defined in header <tbb/version.h>

#define TBB_VERSION_MAJOR   /*implementation-defined*/
#define TBB_VERSION_MINOR   /*implementation-defined*/
#define TBB_VERSION_STRING  /*implementation-defined*/

#define TBB_INTERFACE_VERSION_MAJOR /*implementation-defined*/
#define TBB_INTERFACE_VERSION_MINOR /*implementation-defined*/
#define TBB_INTERFACE_VERSION       /*implementation-defined*/

const char* TBB_runtime_version();
int TBB_runtime_interface_version();
```

Version Macros

oneTBB defines macros related to versioning, as described below.

- `TBB_VERSION_MAJOR` macro defined to integral value that represents major library version.
- `TBB_VERSION_MINOR` macro defined to integral value that represents minor library version.
- `TBB_VERSION_STRING` macro defined to the string representation of the full library version.
- `TBB_INTERFACE_VERSION` macro defined to current interface version. The value is a decimal numeral of the form `xyyz` where `x` is the major interface version number and `y` is the minor interface version number. This macro is increased in each release.
- `TBB_INTERFACE_VERSION_MAJOR` macro defined to `TBB_INTERFACE_VERSION/1000`, which is the major interface version number.
- `TBB_INTERFACE_VERSION_MINOR` macro defined to `TBB_INTERFACE_VERSION%1000/10`, which is the minor interface version number.

TBB_runtime_interface_version Function

Function that returns the interface version of the oneTBB library that was loaded at runtime.

The value returned by `TBB_runtime_interface_version()` may differ from the value of `TBB_INTERFACE_VERSION` obtained at compile time. This can be used to identify whether an application was compiled against a compatible version of the oneTBB headers.

In general, the run-time value `TBB_runtime_interface_version()` must be greater than or equal to the compile-time value of `TBB_INTERFACE_VERSION`. Otherwise, the application may fail to resolve all symbols at run time.

TBB_runtime_version Function

Function that returns the version string of the oneTBB library that was loaded at runtime.

The value returned by `TBB_runtime_version()` may differ from the value of `TBB_VERSION_STRING` obtained at compile time.

TBB_VERSION Environment Variable

Set the environment variable `TBB_VERSION` to `1` to cause the library to print information on `stderr`. Each line is of the form "`TBB: tag value`", where *tag* and *value* provide additional library information below.

> **Caution:** This output is implementation specific and may change at any time.

Enabling Debugging Features

[configuration.debug_features]

The following macros control certain debugging features. In general, it is useful to compile with these features on for development code, and off for production code, because the features may decrease performance. The table below summarizes the macros and their default values. A value of 1 enables the corresponding feature; a value of 0 disables the feature.

Table 3: Debugging Macros

Macro	Default Value	Feature
`TBB_USE_DEBUG`	• Windows* OS: 1 if `_DEBUG` is defined, 0, otherwise. • All other systems: 0.	Default value for all other macros in this table.
`TBB_USE_ASSERT`	`TBB_USE_DEBUG`	Enable internal assertion checking. Can significantly slow down performance.
`TBB_USE_PROFILING_TOOLS`	`TBB_USE_DEBUG`	Enable full support for analysis tools.

TBB_USE_ASSERT Macro

The `TBB_USE_ASSERT` macro controls whether error checking is enabled in the header files. Define `TBB_USE_ASSERT` as `1` to enable error checking.

If an error is detected, the library prints an error message on `stderr` and calls the standard C routine `abort`. To stop a program when internal error checking detects a failure, place a breakpoint on `tbb::assertion_failure`.

TBB_USE_PROFILING_TOOLS Macro

The `TBB_USE_PROFILING_TOOLS` macro controls support for Intel® Inspector XE, Intel® VTune™ Amplifier XE and Intel® Advisor.

Define `TBB_USE_PROFILING_TOOLS` as `1` to enable full support for these tools. Leave `TBB_USE_PROFILING_TOOLS` undefined or equal to zero to enable top performance in release builds, at the expense of turning off some support for tools.

Feature Macros

[configuration.feature_macros]

Macros in this section control optional features of the library.

TBB_USE_EXCEPTIONS macro

The `TBB_USE_EXCEPTIONS` macro controls whether the library headers use exception-handling constructs such as `try`, `catch`, and `throw`. The headers do not use these constructs when `TBB_USE_EXCEPTIONS=0`.

For the Microsoft Windows*, Linux*, and macOS* operating systems, the default value is 1 if exception handling constructs are enabled in the compiler, and 0, otherwise.

> **Caution:** The runtime library may still throw an exception when `TBB_USE_EXCEPTIONS=0`.

TBB_USE_GLIBCXX_VERSION macro

The `TBB_USE_GLIBCXX_VERSION` macro can be used to specify the proper version of GNU libstdc++ if the detection fails. Define the value of the macro equal to `Major*10000 + Minor*100 + Patch`, where Major.Minor.Patch is the actual GCC/libstdc++ version (if unknown, it can be obtained with the `'gcc -dumpversion'` command). For example, if you use libstdc++ from GCC 4.9.2, define `TBB_USE_GLIBCXX_VERSION=40902`.

9.2.2 Algorithms

[algorithms]

oneAPI Threading Building Blocks provides a set of generic parallel algorithms.

Parallel Functions

parallel_for

[algorithms.parallel_for]

Function template that performs parallel iteration over a range of values.

```
// Defined in header <tbb/parallel_for.h>

namespace tbb {

    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, const Func& f, /* see-below */␣
↪partitioner, task_group_context& group);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, const Func& f, task_group_context&␣
↪group);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, const Func& f, /* see-below */␣
↪partitioner);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, const Func& f);

    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, Index step, const Func& f, /* see-
↪below */ partitioner, task_group_context& group);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, Index step, const Func& f, task_group_
↪context& group);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, Index step, const Func& f, /* see-
↪below */ partitioner);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, Index step, const Func& f);

    template<typename Range, typename Body>
    void parallel_for(const Range& range, const Body& body, /* see-below */␣
↪partitioner, task_group_context& group);
    template<typename Range, typename Body>
    void parallel_for(const Range& range, const Body& body, task_group_context&␣
↪group);
    template<typename Range, typename Body>
    void parallel_for(const Range& range, const Body& body, /* see-below */␣
↪partitioner);
    template<typename Range, typename Body>
    void parallel_for(const Range& range, const Body& body);

} // namespace tbb
```

A `partitioner` type may be one of the following entities:

- `const auto_partitioner&`
- `const simple_partitioner&`
- `const static_partitioner&`
- `affinity_partitioner&`

Requirements:

- The `Range` type must meet the *Range requirements*.
- The `Body` type must meet the *ParallelForBody requirements*.
- The `Index` type must meet the *ParallelForIndex requirements*.
- The `Func` type must meet the *ParallelForFunc requirements*.

The `tbb::parallel_for(first, last, step, f)` overload represents parallel execution of the loop:

```
for (auto i = first; i < last; i += step) f(i);
```

The loop must not wrap around. The step value must be positive. If omitted, it is implicitly 1. There is no guarantee that the iterations run in parallel. A deadlock may occur if a lesser iteration waits for a greater iteration. The partitioning strategy is `auto_partitioner` when the parameter is not specified.

The `parallel_for(range,body,partitioner)` overload provides a more general form of parallel iteration. It represents parallel execution of `body` over each value in `range`. The optional `partitioner` parameter specifies a partitioning strategy.

`parallel_for` recursively splits the range into subranges to the point such that `is_divisible()` is false for each subrange, and makes copies of the body for each of these subranges. For each such body/subrange pair, it invokes `Body::operator()`.

Some of the copies of the range and body may be destroyed after `parallel_for` returns. This late destruction is not an issue in typical usage, but is something to be aware of when looking at execution traces or writing range or body objects with complex side effects.

`parallel_for` may execute iterations in non-deterministic order. Do not rely on any particular execution order for correctness. However, for efficiency, do expect `parallel_for` to tend towards operating on consecutive runs of values.

In case of serial execution, `parallel_for` performs iterations from left to right in the following sense.

All overloads can accept a *task_group_context* object so that the algorithm's tasks are executed in this group. By default, the algorithm is executed in a bound group of its own.

Complexity

If the range and body take *O(1)* space, and the range splits into nearly equal pieces, the space complexity is *O(P log(N))*, where *N* is the size of the range and *P* is the number of threads.

See also:

- *Partitioners*

parallel_reduce

[algorithms.parallel_reduce]

Function template that computes reduction over a range.

```
// Defined in header <tbb/parallel_reduce.h>

namespace tbb {

    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_reduce(const Range& range, const Value& identity, const Func& func,
↪ const Reduction& reduction, /* see-below */ partitioner, task_group_context&␣
↪group);
    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_reduce(const Range& range, const Value& identity, const Func& func,
↪ const Reduction& reduction, /* see-below */ partitioner);
    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_reduce(const Range& range, const Value& identity, const Func& func,
↪ const Reduction& reduction, task_group_context& group);
    template<typename Range, typename Value, typename Func, typename Reduction>
```

(continues on next page)

(continued from previous page)

```
    Value parallel_reduce(const Range& range, const Value& identity, const Func& func,
↪ const Reduction& reduction);

    template<typename Range, typename Body>
    void parallel_reduce(const Range& range, Body& body, /* see-below */ partitioner,␣
↪task_group_context& group);
    template<typename Range, typename Body>
    void parallel_reduce(const Range& range, Body& body, /* see-below */ partitioner);
    template<typename Range, typename Body>
    void parallel_reduce(const Range& range, Body& body, task_group_context& group);
    template<typename Range, typename Body>
    void parallel_reduce(const Range& range, Body& body);

} // namespace tbb
```

A `partitioner` type may be one of the following entities:

- `const auto_partitioner&`
- `const simple_partitioner&`
- `const static_partitioner&`
- `affinity_partitioner&`

Requirements:

- The `Range` type must meet the *Range requirements*.
- The `Body` type must meet the *ParallelReduceBody requirements*.
- The `Func` type must meet the *ParallelReduceFunc requirements*.
- The `Reduction` types must meet :*ParallelReduceReduction requirements*.

The function template `parallel_reduce` has two forms: The functional form is designed to be easy to use in conjunction with lambda expressions. The imperative form is designed to minimize copying of data.

The functional form `parallel_reduce(range, identity, func, reduction)` performs a parallel reduction by applying *func* to subranges in *range* and reducing the results with the binary operator *reduction*. It returns the result of the reduction. The *identity* parameter specifies the left identity element for *func*'s `operator()`. Parameters *func* and *reduction* can be lambda expressions.

The imperative form `parallel_reduce(range,body)` performs parallel reduction of *body* over each value in *range*.

A `parallel_reduce` recursively splits the range into subranges to the point such that `is_divisible()` is false for each subrange. A `parallel_reduce` uses the splitting constructor to make one or more copies of the body for each thread. It may copy a body while the body's `operator()` or method `join` runs concurrently. You are responsible for ensuring the safety of such concurrency. In typical usage, the safety requires no extra effort.

`parallel_reduce` may invoke the splitting constructor for the body. For each such split of the body, it invokes the `join` method to merge the results from the bodies. Define `join` to update this to represent the accumulated result for this and rhs. The reduction operation should be associative, but does not have to be commutative. For a noncommutative operation *op*, `left.join(right)` should update *left* to be the result of *left op right*.

A body is split only if the range is split, but the converse is not necessarily to be so. The user must neither rely on a particular choice of body splitting nor on the subranges processed by a given body object being consecutive. `parallel_reduce` makes the choice of body splitting nondeterministically.

When executed serially `parallel_reduce` run sequentially from left to right in the same sense as for `parallel_for`. Sequential execution never invokes the splitting constructor or method join.

All overloads can accept a *task_group_context* object so that the algorithm's tasks are executed in this group. By default, the algorithm is executed in a bound group of its own.

Complexity

If the range and body take *O(1)* space, and the range splits into nearly equal pieces, the space complexity is *O(P×log(N))*, where *N* is the size of the range and *P* is the number of threads.

Example (Imperative Form)

The following code sums the values in an array.

```
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;

struct Sum {
    float value;
    Sum() : value(0) {}
    Sum( Sum& s, split ) {value = 0;}
    void operator()( const blocked_range<float*>& r ) {
        float temp = value;
        for( float* a=r.begin(); a!=r.end(); ++a ) {
            temp += *a;
        }
        value = temp;
    }
    void join( Sum& rhs ) {value += rhs.value;}
};

float ParallelSum( float array[], size_t n ) {
    Sum total;
    parallel_reduce( blocked_range<float*>( array, array+n ), total );
    return total.value;
}
```

The example generalizes to reduction for any associative operation *op* as follows:

- Replace occurrences of 0 with the identity element for *op*
- Replace occurrences of += with *op*= or its logical equivalent.
- Change the name `Sum` to something more appropriate for *op*.

The operation may be noncommutative. For example, *op* could be matrix multiplication.

Example with Lambda Expressions

The following is similar to the previous example, but written using lambda expressions and the functional form of `parallel_reduce`.

```
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;
```

(continues on next page)

(continued from previous page)

```
float ParallelSum( float array[], size_t n ) {
    return parallel_reduce(
        blocked_range<float*>( array, array+n ),
        0.f,
        [](const blocked_range<float*>& r, float init)->float {
            for( float* a=r.begin(); a!=r.end(); ++a )
                init += *a;
            return init;
        },
        []( float x, float y )->float {
            return x+y;
        }
    );
}
```

See also:

- *Partitioners*

parallel_deterministic_reduce

[algorithms.parallel_deterministic_reduce]

Function template that computes reduction over a range, with deterministic split/join behavior.

```
// Defined in header <tbb/parallel_reduce.h>

namespace tbb {

    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_deterministic_reduce( const Range& range, const Value& identity,␣
↪const Func& func, const Reduction& reduction, /* see-below */ partitioner, task_
↪group_context& group);
    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_deterministic_reduce( const Range& range, const Value& identity,␣
↪const Func& func, const Reduction& reduction, /* see-below */ partitioner);
    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_deterministic_reduce( const Range& range, const Value& identity,␣
↪const Func& func, const Reduction& reduction, task_group_context& group);
    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_deterministic_reduce( const Range& range, const Value& identity,␣
↪const Func& func, const Reduction& reduction);

    template<typename Range, typename Body>
    void parallel_deterministic_reduce( const Range& range, Body& body, /* see-below␣
↪*/ partitioner, task_group_context& group);
    template<typename Range, typename Body>
    void parallel_deterministic_reduce( const Range& range, Body& body, /* see-below␣
↪*/ partitioner);
    template<typename Range, typename Body>
    void parallel_deterministic_reduce( const Range& range, Body& body, task_group_
↪context& group);
    template<typename Range, typename Body>
    void parallel_deterministic_reduce( const Range& range, Body& body);

} // namespace tbb
```

A `partitioner` type may be one of the following entities:

- `const simple_partitioner&`
- `const static_partitioner&`

The function template `parallel_deterministic_reduce` is very similar to the `parallel_reduce` template. It also has the functional and imperative forms and has *similar requirements*.

Unlike `parallel_reduce`, `parallel_deterministic_reduce` has deterministic behavior with regard to splits of both `Body` and `Range` and joins of the bodies. For the functional form, `Func` is applied to a deterministic set of `Ranges`, and `Reduction` merges partial results in a deterministic order. To achieve that, `parallel_deterministic_reduce` uses a `simple_partitioner` or a `static_partitioner` only because other partitioners react to random work stealing behavior.

Caution: Since `simple_partitioner` does not automatically coarsen ranges, make sure to specify an appropriate grain size. See *Partitioners section* for more information.

`parallel_deterministic_reduce` always invokes the `Body` splitting constructor for each range split.

As a result, `parallel_deterministic_reduce` recursively splits a range until it is no longer divisible, and creates a new body (by calling the `Body` splitting constructor) for each new subrange. Like `parallel_reduce`, for each body split the method `join` is invoked in order to merge the results from the bodies.

Therefore, for given arguments, `parallel_deterministic_reduce` executes the same set of split and join operations no matter how many threads participate in execution and how tasks are mapped to the threads. If the user-provided functions are also deterministic (that is, different runs with the same input result in the same output), multiple calls to `parallel_deterministic_reduce` produce the same result. Note however that the result might differ from that obtained with an equivalent sequential (linear) algorithm.

Complexity

If the range and body take *O(1)* space, and the range splits into nearly equal pieces, the space complexity is *O(P log(N))*, where *N* is the size of the range and *P* is the number of threads.

See also:

- *parallel_reduce*
- *Partitioners*

parallel_scan

[algorithms.parallel_scan]

Function template that computes a parallel prefix.

```
// Defined in header <tbb/parallel_scan.h>

template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body );
template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body, /* see-below */ partitioner );

template<typename Range, typename Value, typename Scan, typename Combine>
Value parallel_scan( const Range& range, const Value& identity, const Scan& scan,␣
↪const Combine& combine );
```

(continues on next page)

(continued from previous page)

```
template<typename Range, typename Value, typename Scan, typename Combine>
Value parallel_scan( const Range& range, const Value& identity, const Scan& scan,␣
↪const Combine& combine, /* see-below */ partitioner );
```

A `partitioner` type may be one of the following entities:

- `const auto_partitioner&`
- `const simple_partitioner&`

Requirements:

- The `Range` type must meet the *Range requirement*.
- The `Body` type must meet the *ParallelScanBody requirements*.
- The `Scan` type must meet the *ParallelScanFunc requirements*.
- The `Combine` type must meet the *ParallelScanCombine requirements*.

The function template `parallel_scan` computes a parallel prefix, also known as a parallel scan. This computation is an advanced concept in parallel computing that is sometimes useful in scenarios that appear to have inherently serial dependences.

A mathematical definition of the parallel prefix is as follows. Let × be an associative operation with left-identity element id_x. The parallel prefix of × over a sequence $z_0, z_1, \ldots$ *z*$_{n-1}$ is a sequence $y_0, y_1, y_2, \ldots$ *y*$_{n-1}$ where:

- $y_0 = id_x \times z_0$
- $y_i = y_{i-1} \times z_i$

For example, if × is addition, the parallel prefix corresponds to a running sum. A serial implementation of a parallel prefix is:

```
T temp = id;
for( int i=1; i<=n; ++i ) {
   temp = temp + z[i];
   y[i] = temp;
}
```

Parallel prefix performs this in parallel by reassociating the application of × (+ in example) and using two passes. It may invoke × up to twice as many times as the serial prefix algorithm. Even though it does more work, given the right grain size the parallel algorithm can outperform the serial one because it distributes the work across multiple hardware threads.

The function template `parallel_scan` has two forms. The imperative form `parallel_scan(range, body)` implements parallel prefix generically.

A summary (refer to *ParallelScanBody requirements*) contains enough information such that for two consecutive subranges *r* and *s*:

- If *r* has no preceding subrange, the scan result for *s* can be computed from knowing *s* and the summary for *r*.
- A summary of *r* concatenated with *s* can be computed from the summaries of *r* and *s*.

The functional form `parallel_scan(range, identity, scan, combine)` is designed to use with functors and lambda expressions, hiding some complexities of the imperative form. It uses the same *scan* functor in both passes, differentiating them via a boolean parameter, combines summaries with *combine* functor, and returns the summary computed over the whole *range*. The *identity* argument is the left identity element for `Scan::operator()`.

pre_scan and final_scan Classes

pre_scan_tag and final_scan_tag

[algorithms.parallel_scan.scan_tags]

Types that distinguish the phases of `parallel_scan`.

Types `pre_scan_tag` and `final_scan_tag` are dummy types used in conjunction with `parallel_scan`. See the example in the *parallel_scan* section for demonstration of how they are used in the signature of `operator()`.

```
// Defined in header <tbb/parallel_scan.h>

namespace tbb {

   struct pre_scan_tag {
       static bool is_final_scan();
       operator bool();
   };

   struct final_scan_tag {
       static bool is_final_scan();
       operator bool();
   };

}
```

Member functions

bool **is_final_scan**()
: `true` for a `final_scan_tag`, `false`, otherwise.

operator bool()
: `true` for a `final_scan_tag`, `false`, otherwise.

The `parallel_scan` template makes an effort to avoid prescanning where possible. When executed serially, `parallel_scan` processes the subranges without any pre-scans by processing the subranges from left to right using final scans. That is why final scans must compute a summary as well as the final scan result. The summary might be needed to process the next subrange if no other thread has pre-scanned it yet.

Example (Imperative Form)

The following code demonstrates how `Body` could be implemented for `parallel_scan` to compute the same result as in the earlier sequential example.

```
class Body {
    T sum;
    T* const y;
    const T* const z;
public:
    Body( T y_[], const T z_[] ) : sum(id), z(z_), y(y_) {}
    T get_sum() const { return sum; }

    template<typename Tag>
```

(continues on next page)

(continued from previous page)

```
    void operator()( const tbb::blocked_range<int>& r, Tag ) {
        T temp = sum;
        for( int i=r.begin(); i<r.end(); ++i ) {
            temp = temp + z[i];
            if( Tag::is_final_scan() )
                y[i] = temp;
        }
        sum = temp;
    }
    Body( Body& b, tbb::split ) : z(b.z), y(b.y), sum(id) {}
    void reverse_join( Body& a ) { sum = a.sum + sum; }
    void assign( Body& b ) { sum = b.sum; }
};

T DoParallelScan( T y[], const T z[], int n ) {
    Body body(y,z);
    tbb::parallel_scan( tbb::blocked_range<int>(0,n), body );
    return body.get_sum();
}
```

The definition of `operator()` demonstrates typical patterns when using `parallel_scan`.

- A single template defines both versions. Doing so is not required, but usually saves coding effort, because two versions are usually similar. The library defines the static method `is_final_scan` to enable differentiation between the versions.
- The prescan variant computes the × reduction, but does not update `y`. The prescan is used by `parallel_scan` to generate look-ahead partial reductions.
- The final scan variant computes the × reduction and updates `y`.

The `reverse_join` operation is similar to the `join` operation used by `parallel_reduce`, except that the arguments are reversed. That is, `this` is the *right* argument of ×. The template function `parallel_scan` decides if and when to generate parallel work. Thus, it is crucial that × is associative and that the methods of `Body` faithfully represent it. Operations such as floating-point addition, which are somewhat associative, can be used with the understanding that the results may be rounded differently depending on the association used by `parallel_scan`. The reassociation may differ between runs even on the same machine. However, when executed serially, `parallel_scan` associates identically to the serial form shown at the beginning of this section.

If you change the example to use a `simple_partitioner`, be sure to provide a grain size. The code below shows how to do this for the grain size of 1000:

```
parallel_scan( blocked_range<int>(0,n,1000), total, simple_partitioner() );
```

Example with Lambda Expressions

The following is analogous to the previous example, but written using lambda expressions and the functional form of `parallel_scan`:

```
T DoParallelScan( T y[], const T z[], int n ) {
    return tbb::parallel_scan(
        tbb::blocked_range<int>(0,n),
        id,
        [](const tbb::blocked_range<int>& r, T sum, bool is_final_scan)->T {
            T temp = sum;
            for( int i=r.begin(); i<r.end(); ++i ) {
```

(continues on next page)

(continued from previous page)

```
                temp = temp + z[i];
                if( is_final_scan )
                    y[i] = temp;
            }
            return temp;
        },
        []( T left, T right ) {
            return left + right;
        }
    );
}
```

See also:

- *blocked_range class*
- *parallel_reduce algorithm*

parallel_for_each

[algorithms.parallel_for_each]

Function template that processes work items in parallel.

```
// Defined in header <tbb/parallel_for_each.h>

namespace tbb {

    template<typename InputIterator, typename Body>
    void parallel_for_each( InputIterator first, InputIterator last, Body body );
    template<typename InputIterator, typename Body>
    void parallel_for_each( InputIterator first, InputIterator last, Body body, task_
↪group_context& group );

    template<typename Container, typename Body>
    void parallel_for_each( Container& c, Body body );
    template<typename Container, typename Body>
    void parallel_for_each( Container& c, Body body, task_group_context& group );

    template<typename Container, typename Body>
    void parallel_for_each( const Container& c, Body body );
    template<typename Container, typename Body>
    void parallel_for_each( const Container& c, Body body, task_group_context& group␣
↪);

} // namespace tbb
```

Requirements:

- The `Body` type must meet the *ParallelForEachBody requirements*.
- The `InputIterator` type must meet the *Input Iterator* requirements from the [input.iterators] ISO C++ Standard section.
- The `Container` type must meet the *ContainerBasedSequence requirements*.

The `parallel_for_each` template has two forms.

The sequence form `parallel_for_each(first, last, body)` applies a function object `body` over a sequence `[first,last)`. Items may be processed in parallel.

The container form `parallel_for_each(c, body)` is equivalent to `parallel_for_each(std::begin(c), std::end(c), body)`.

All overloads can accept a *task_group_context* object so that the algorithm's tasks are executed in this group. By default, the algorithm is executed in a bound group of its own.

feeder Class

Additional work items can be added by `body` if it has a second argument of type `feeder`. The function terminates when `body(x)` returns for all items `x` that were in the input sequence or added by method `feeder::add`.

feeder

[algorithms.parallel_for_each.feeder]

Inlet into which additional work items for a `parallel_for_each` can be fed.

```
// Defined in header <tbb/parallel_for_each.h>

namespace tbb {

    template<typename Item>
    class feeder {
    public:
        void add( const Item& item );
        void add( Item&& item );
    };

} // namespace tbb
```

Member functions

void **add**(**const** Item &*item*)
: Adds item to a collection of work items to be processed.

void **add**(Item &&*item*)
: Same as the above but uses the move constructor of `Item`, if available.

Caution: Must be called from a `Body::operator()` created by the `parallel_for_each` function. Otherwise, the termination semantics of method `operator()` are undefined.

Example

The following code sketches a body with the two-argument form of `operator()`.

```
struct MyBody {
    void operator()(item_t item, parallel_do_feeder<item_t>& feeder ) {
        for each new piece of work implied by item do {
            item_t new_item = initializer;
            feeder.add(new_item);
        }
    }
};
```

parallel_invoke

[algortihms.parallel_invoke]

Function template that evaluates several functions in parallel.

```
// Defined in header <tbb/parallel_invoke.h>

namespace tbb {

    template<typename... Functions>
    void parallel_invoke(Functions&&... fs);

} // namespace tbb
```

Requirements:

- All members of `Functions` parameter pack must meet `Function Objects` requirements from the [function.objects] ISO C++ Standard section or be a pointer to a function.
- Last member of `Functions` parameter pack may be a `task_group_context&` type.

Evaluates each member passed to `parallel_invoke` possibly in parallel. Return values are ignored.

The algorithm can accept a *task_group_context* object so that the algorithm's tasks are executed in this group. By default, the algorithm is executed in a bound group of its own.

Example

The following example evaluates `f()`, `g()`, `h()`, and `bar(1)` in parallel.

```
#include "tbb/parallel_invoke.h"

extern void f();
extern void bar(int);

class MyFunctor {
    int arg;
public:
    MyFunctor(int a) : arg(a) {}
    void operator()() const { bar(arg); }
};
```

(continues on next page)

(continued from previous page)

```
void RunFunctionsInParallel() {
    MyFunctor g(2);
    MyFunctor h(3);

    tbb::parallel_invoke(f, g, h, []{bar(1);});
}
```

parallel_pipeline

[algorithms.parallel_pipeline]

Strongly-typed interface for pipelined execution.

```
// Defined in header <tbb/parallel_pipeline.h>

namespace tbb {

    void parallel_pipeline( size_t max_number_of_live_tokens, const filter<void,void>&
↪ filter_chain );
    void parallel_pipeline( size_t max_number_of_live_tokens, const filter<void,void>&
↪ filter_chain, task_group_context& group );

} // namespace tbb
```

A `parallel_pipeline` algorithm represents pipelined application of a series of filters to a stream of items. Each filter operates in a particular mode: parallel, serial in-order, or serial out-of-order.

To build and run a pipeline from functors g_0, g_1, g_2, ..., g_n, write:

```
parallel_pipeline( max_number_of_live_tokens,
                   make_filter<void,I1>(mode0,g0) &
                   make_filter<I1,I2>(mode1,g1) &
                   make_filter<I2,I3>(mode2,g2) &
                   ...
                   make_filter<In,void>(moden,gn) );
```

In general, the g_i functor should define its `operator()` to map objects of type I_i to objects of type I_{i+1}. Functor g_0 is a special case, because it notifies the pipeline when the end of an input stream is reached. Functor g_0 must be defined such that for a *flow_control* object *fc*, the expression g_0 (*fc*) either returns the next value in the input stream, or invokes *fc.stop()* if the end of the input stream is reached and returns a dummy value.

Each *filter* should be specified by two template arguments. These arguments define filters input and output types. The first and last filters are special cases. Input type of the first filter must be *void*, output type of the last filter must be *void* too.

Before passing to *parallel_pipeline*, concatenate all filters to one(`filter<void, void>`) with *filter::operator&()*. The operator requires that the second template argument of its left operand matches the first template argument of its second operand.

The number of items processed in parallel depends on the structure of the pipeline and number of available threads. *max_number_of_live_tokens* sets the threshold for concurrently processed items.

If the *group* argument is specified, pipeline's tasks are executed in this group. By default, the algorithm is executed in a bound group of its own.

Example

The following example uses `parallel_pipeline` to compute the root-mean-square of a sequence defined by [*first* , *last*).

```
float RootMeanSquare( float* first, float* last ) {
    float sum=0;
    parallel_pipeline( /*max_number_of_live_token=*/16,
        make_filter<void,float>(
            filter::serial,
            [&](flow_control& fc)-> float*{
                if( first<last ) {
                    return first++;
                } else {
                    fc.stop();
                    return nullptr;
                }
            }
        ) &
        make_filter<float*,float>(
            filter::parallel,
            [](float* p){return (*p)*(*p);}
        ) &
        make_filter<float,void>(
            filter::serial,
            [&](float x) {sum+=x;}
        )
    );
    return sqrt(sum);
}
```

filter Class Template

filter

[algorithms.parallel_pipeline.filter]

A `filter` class template represents a strongly-typed filter in a `parallel_pipeline` algorithm, with its template parameters specifying the filter input and output types. A `filter` can be constructed from a functor or by composing two `filter` objects with `operator&()`. The same `filter` object can be reused in multiple `&` expressions.

The `filter` class should only be used in conjunction with `parallel_pipeline` functions.

```
// Defined in header <tbb/parallel_pipeline.h>

namespace tbb {

    template<typename InputType, typename OutputType>
    class filter {
    public:
        filter() = default;
        filter( const filter& rhs ) = default;
        filter( filter&& rhs ) = default;
        void operator=(const filter& rhs) = default;
        void operator=( filter&& rhs ) = default;
```

(continues on next page)

(continued from previous page)

```
        template<typename Body>
        filter( filter_mode mode, const Body& body );

        filter& operator&=( const filter<OutputType,OutputType>& right );

        void clear();
    }

    template<typename T, typename U, typename Body>
    filter<T,U> make_filter( filter::mode mode, const Body& f );
    template<typename T, typename V, typename U>
    filter<T,U> operator&( const filter<T,V>& left, const filter<V,U>& right );

}
```

Requirements:

- If *InputType* is `void`, a `Body` type must meet the *StartFilterBody requirements*.
- If *OutputType* is `void`, a `Body` type must meet the *OutputFilterBody requirements*.
- If *InputType* and *OutputType* are not `void`, a `Body` type must meet the *MiddleFilterBody requirements*.
- If *InputType* and *OutputType* are `void`, a `Body` type must meet the *SingleFilterBody requirements*.

filter_mode Enumeration

filter_mode

[algorithms.parallel_pipeline.filter_mode]

A `filter_mode` enumeration represents an execution mode of a `filter` in a `parallel_pipeline` algorithm.

Its enumerated values and their meanings are as follows:

- A `parallel` filter can process multiple items in parallel and without a particular order.
- A `serial_out_of_order` filter processes items one at a time and without a particular order.
- A `serial_in_order` filter processes items one at a time. The order in which items are processed is implicitly set by the first *serial_in_order* filter and respected by all other such filters in the pipeline.

```
// Defined in header <tbb/parallel_pipeline.h>

namespace tbb {

    enum class filter_mode {
        parallel = /*implementation-defined*/,
        serial_in_order = /*implementation-defined*/,
        serial_out_of_order = /*implementation-defined*/
    };

}
```

Member functions

filter()
Constructs an undefined filter.

> **Caution:** The effect of using an undefined filter by operator&() or parallel_pipeline is undefined.

template<typename **Body**>
filter(filter_mode *mode*, **const** *Body* &*body*)
Constructs a filter that uses a copy of a provided body to map an input value of type *InputType* to an output value of type *OutputType*, and that operates in the specified mode.

void **clear**()
Sets *this to an undefined filter.

Non-member functions

template<typename **T**, typename **U**, typename **Func**>
filter<*T*, *U*> **make_filter**(*filter*::mode *mode*, **const** *Func* &*f*)
Returns filter<T, U>(mode, f).

template<typename **T**, typename **V**, typename **U**>
filter<*T*, *U*> **operator&**(**const** *filter*<*T*, *V*> &*left*, **const** *filter*<*V*, *U*> &*right*)
Returns a filter representing the composition of filters *left* and *right*. The composition behaves as if the output value of *left* becomes the input value of *right*.

Deduction Guides

```
template<typename Body>
filter(filter_mode, Body) -> filter<filter_input<Body>, filter_output<Body>>;
```

Where:

- filter_input<Body> is an alias to the Body::operator() input parameter type. If Body::operator() input parameter type is flow_control then filter_input<Body> is void.
- filter_output<Body> is an alias to the Body::operator() return type.

flow_control Class

flow_control

[algorithms.parallel_pipeline.flow_control]

Enables the first filter in a composite filter to indicate when the end of input stream is reached.

Template function parallel_pipeline passes a flow_control object to the functor of the first filter. When the functor reaches the end of its input stream, it should invoke fc.stop() and return a dummy value that will not be passed to the next filter.

```
// Defined in header <tbb/parallel_pipeline.h>

namespace tbb {

    class flow_control {
    public:
        void stop();
    };

}
```

Member functions

void **stop**()
: Indicates that first filter of the pipeline reaches the end of its output.

See also:

- *FilterBody requiremnts*
- *filter class*

See also:

- *task_group_context*

parallel_sort

[algorithms.parallel_sort]

Function template that sorts a sequence.

```
// Defined in header <tbb/parallel_sort.h>

namespace tbb {

    template<typename RandomAccessIterator>
    void parallel_sort( RandomAccessIterator begin, RandomAccessIterator end );
    template<typename RandomAccessIterator, typename Compare>
    void parallel_sort( RandomAccessIterator begin, RandomAccessIterator end, const␣
↪Compare& comp );

    template<typename Container>
    void parallel_sort( Container& c );
    template<typename Container>
    void parallel_sort( Container& c, const Compare& comp );

} //  namespace tbb
```

Requirements:

- The `RandomAccessIterator` type must meet the *Random Access Iterators* requirements from [random.access.iterators] and *Swappable* requirements from the [swappable.requirements] ISO C++ Standard section.
- The `Compare` type must meet the *Compare* type requirements from the [alg.sorting] ISO C++ Standard section.

- The `Container` type must meet the *ContainerBasedSequence requirements* which iterators must meet the *Random Access Iterators* requirements from [random.access.iterators] and *Swappable* requirements from the [swappable.requirements] ISO C++ Standard section.

Sorts a sequence or a container. The sort is neither stable nor deterministic: relative ordering of elements with equal keys is not preserved and not guaranteed to repeat if the same sequence is sorted again.

A call `parallel_sort( begin, end, comp )` sorts the sequence *[begin, end)* using the argument `comp` to determine relative orderings. If `comp( x, y )` returns `true`, *x* appears before *y* in the sorted sequence.

A call `parallel_sort( begin, end )` is equivalent to `parallel_sort( begin, end, comp )`, where *comp* uses *operator<* to determine relative orderings.

A call `parallel_sort( c, comp )` is equivalent to `parallel_sort( std::begin(c), std::end(c), comp )`.

A call `parallel_sort( c )` is equivalent to `parallel_sort( c, comp )`, where *comp* uses *operator<* to determine relative orderings.

Complexity

`parallel_sort` is a comparison sort with an average time complexity of *O(N×log(N))*, where *N* is the number of elements in the sequence. `parallel_sort` may be executed concurrently to improve execution time.

Blocked Ranges

Types that meet the *Range requirements*.

blocked_range

[algorithms.blocked_range]

Class template for a recursively divisible half-open interval.

A `blocked_range` represents a half-open range [*i*,*j*) that can be recursively split.

A `blocked_range` meets the *Range requirements*.

A `blocked_range` specifies a *grain size* of type `size_t`.

A `blocked_range` is splittable into two subranges if the size of the range exceeds its grain size. The ideal grain size depends on the context of the `blocked_range`, which is typically passed as the range argument to the loop templates `parallel_for`, `parallel_reduce`, or `parallel_scan`.

```
// Defined in header <tbb/blocked_range.h>

namespace tbb {

    template<typename Value>
    class blocked_range {
    public:
        // types
        using size_type = size_t;
        using const_iterator = Value;

        // constructors
        blocked_range( Value begin, Value end, size_type grainsize=1 );
        blocked_range( blocked_range& r, split );
        blocked_range( blocked_range& r, proportional_split& proportion );
```

(continues on next page)

(continued from previous page)

```
        // capacity
        size_type size() const;
        bool empty() const;

        // access
        size_type grainsize() const;
        bool is_divisible() const;

        // iterators
        const_iterator begin() const;
        const_iterator end() const;
    };
}
```

Requirements:

- The `Value` type must meet the *BlockedRangeValue requirements*.

Member functions

type size_type
The type for measuring the size of a `blocked_range`. The type is always a `size_t`.

type const_iterator
The type of a value in the range. Despite its name, the `const_iterator` type is not necessarily an STL iterator; it merely needs to meet the *BlockedRangeValue requirements*. However, it is convenient to call it `const_iterator` so that if it is a const_iterator, the `blocked_range` behaves like a read-only STL container.

blocked_range (Value *begin*, Value *end*, *size_type grainsize* = 1)
Requirements: The parameter `grainsize` must be positive. The debug version of the library raises an assertion failure if this requirement is not met.

Effects: Constructs a `blocked_range` representing the half-open interval `[begin, end)` with the given `grainsize`.

Example: The statement "`blocked_range<int> r(5, 14, 2);`" constructs a range of `int` that contains the values 5 through 13 inclusive, with the grain size of 2. Afterwards, `r.begin()==5` and `r.end()==14`.

blocked_range (*blocked_range* &*range*, split)
Basic splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges. The newly constructed `blocked_range` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same `grainsize` as the original range.

Example: Let `r` be a `blocked_range` that represents a half-open interval `[i, j)` with a grain size `g`. Running the statement `blocked_range<int> s(r, split);` subsequently causes r to represent `[i, i+(j-i)/2)` and `s` to represent `[i+(j-i)/2, j)`, both with grain size `g`.

blocked_range (*blocked_range* &*range*, *proportional_split proportion*)
Proportional splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges such that the ratio of their sizes is close to the ratio of `proportion.left()` to `proportion.right()`. The newly constructed `blocked_range` is the subrange at the right, and `range` is updated to be the subrange at the left.

Example: Let `r` be a `blocked_range` that represents a half-open interval `[i, j)` with a grain size `g`. Running the statement `blocked_range<int> s(r, proportional_split(2, 3));` subsequently causes `r` to represent `[i, i+2*(j-i)/(2+3))` and `s` to represent `[i+2*(j-i)/(2+3), j)`, both with grain size `g`.

size_type **size**() **const**
Requirements: `end()<begin()` is false.

Effects: Determines size of range.

Returns: `end()-begin()`.

bool **empty**() **const**
Effects: Determines if range is empty.

Returns: `!(begin()<end())`

size_type **grainsize**() **const**
Returns: Grain size of range.

bool **is_divisible**() **const**
Requirements: `end()<begin()` is false.

Effects: Determines if the range can be split into subranges.

Returns: True if `size()>grainsize()`; false, otherwise.

const_iterator **begin**() **const**
Returns: Inclusive lower bound of the range.

const_iterator **end**() **const**
Returns: Exclusive upper bound of the range.

See also:

- *parallel_reduce*
- *parallel_for*
- *parallel_scan*

blocked_range2d

[algorithms.blocked_range2d]

Class template that represents a recursively divisible two-dimensional half-open interval.

A `blocked_range2d` represents a half-open two-dimensional range `[i0,j0)×[i1,j1)`. Each axis of the range has its own splitting threshold. A `blocked_range2d` is divisible if either axis is divisible.

A `blocked_range2d` meets the *Range requirements*.

```
// Defined in header <tbb/blocked_range2d.h>

namespace tbb {

    template<typename RowValue, typename ColValue=RowValue>
    class blocked_range2d {
```

(continues on next page)

(continued from previous page)

```
    public:
        // Types
        using row_range_type = blocked_range<RowValue>;
        using col_range_type = blocked_range<ColValue>;

        // Constructors
        blocked_range2d(
            RowValue row_begin, RowValue row_end,
            typename row_range_type::size_type row_grainsize,
            ColValue col_begin, ColValue col_end,
            typename col_range_type::size_type col_grainsize);
        blocked_range2d( RowValue row_begin, RowValue row_end,
                         ColValue col_begin, ColValue col_end );
        // Splitting constructors
        blocked_range2d( blocked_range2d& r, split );
        blocked_range2d( blocked_range2d& r, proportional_split proportion );

        // Capacity
        bool empty() const;

        // Access
        bool is_divisible() const;
        const row_range_type& rows() const;
        const col_range_type& cols() const;
    };

} // namespace tbb
```

Requirements:

- The *RowValue* and *ColValue* must meet the *blocked_range requirements*

Member types

```
using row_range_type = blocked_range<RowValue>;
```

The type of the row values.

```
using col_range_type = blocked_range<ColValue>;
```

The type of the column values.

Member functions

```
blocked_range2d(
    RowValue row_begin, RowValue row_end,
    typename row_range_type::size_type row_grainsize,
    ColValue col_begin, ColValue col_end,
    typename col_range_type::size_type col_grainsize);
```

Effects: Constructs a `blocked_range2d` representing a two-dimensional space of values. The space is the half-open Cartesian product `[row_begin, row_end) x [col_begin, col_end)`, with the given grain sizes for the rows and columns.

Example: The statement `blocked_range2d<char,int> r('a', 'z'+1, 3, 0, 10, 2 );` constructs a two-dimensional space that contains all value pairs of the form `(i, j)`, where `i` ranges from `'a'` to `'z'` with a grain size of 3, and `j` ranges from 0 to 9 with a grain size of 2.

```
blocked_range2d(RowValue row_begin, RowValue row_end,
                ColValue col_begin, ColValue col_end);
```

Same as `blocked_range2d(row_begin,row_end,1,col_begin,col_end,1)`.

```
blocked_range2d(blocked_range2d& range, split);
```

Basic splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions range into two subranges. The newly constructed `blocked_range2d` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same grain size as the original `range`. Splitting is done either by rows or columns. The choice of which axis to split is intended to cause, after repeated splitting, the subranges to approach the aspect ratio of the respective row and column grain sizes.

```
blocked_range2d(blocked_range2d& range, proportional_split proportion);
```

Proportional splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges in the given `proportion` across one of its axes. The choice of which axis to split is made in the same way as for the basic splitting constructor; then, proportional splitting is done for the chosen axis. The second axis and the grain sizes for each subrange remain the same as in the original range.

```
bool empty() const;
```

Effects: Determines if range is empty.

Returns: `rows().empty()||cols().empty()`

```
bool is_divisible() const;
```

Effects: Determines if range can be split into subranges.

Returns: `rows().is_divisible()||cols().is_divisible()`

```
const row_range_type& rows() const;
```

Returns: Range containing the rows of the value space.

```
const col_range_type& cols() const;
```

Returns: Range containing the columns of the value space.

See also:

- *blocked_range*

blocked_range3d

[algorithms.blocked_range3d]

Class template that represents a recursively divisible three-dimensional half-open interval.

A `blocked_range3d` is the three-dimensional extension of `blocked_range2d`.

```
namespace tbb {
    template<typename PageValue, typename RowValue=PageValue, typename␣
↪ColValue=RowValue>
    class blocked_range3d {
    public:
        // Types
        using page_range_type = blocked_range<PageValue>;
        using row_range_type = blocked_range<RowValue>;
        using col_range_type = blocked_range<ColValue>;

        // Constructors
        blocked_range3d(
            PageValue page_begin, PageValue page_end,
            typename page_range_type::size_type page_grainsize,
            RowValue row_begin, RowValue row_end,
            typename row_range_type::size_type row_grainsize,
            ColValue col_begin, ColValue col_end,
            typename col_range_type::size_type col_grainsize );
        blocked_range3d( PageValue page_begin, PageValue page_end
                         RowValue row_begin, RowValue row_end,
                         ColValue col_begin, ColValue col_end );
        blocked_range3d( blocked_range3d& r, split );
        blocked_range3d( blocked_range3d& r, proportional_split& proportion );

        // Capacity
        bool empty() const;

        // Access
        bool is_divisible() const;
        const page_range_type& pages() const;
        const row_range_type& rows() const;
        const col_range_type& cols() const;
    };
}
```

Requirements:

- The *PageValue*, *RowValue* and *ColValue* must meet the *blocked_range requirements*

Member types

```
using page_range_type = blocked_range<PageValue>;
```

The type of the page values.

```
using row_range_type = blocked_range<RowValue>;
```

The type of the row values.

```
using col_range_type = blocked_range<ColValue>;
```

The type of the column values.

Member functions

```
blocked_range3d(PageValue page_begin, PageValue page_end,
    typename page_range_type::size_type page_grainsize,
    RowValue row_begin, RowValue row_end,
    typename row_range_type::size_type row_grainsize,
    ColValue col_begin, ColValue col_end,
    typename col_range_type::size_type col_grainsize);
```

Effects: Constructs a `blocked_range3d` representing a three-dimensional space of values. The space is the half-open Cartesian product `[page_begin, page_end) x [row_begin, row_end) x [col_begin, col_end)`, with the given grain sizes for the pages, rows and columns.

Example: The statement `blocked_range3d<int,char,int> r(0, 6, 2, 'a', 'z'+1, 3, 0, 10, 2 );` constructs a three-dimensional space that contains all value pairs of the form `(i, j, k)`, where `i` ranges from 0 to 6 with a grain size of 2, `j` ranges from `'a'` to `'z'` with a grain size of 3, and `k` ranges from 0 to 9 with a grain size of 2.

```
blocked_range3d(PageValue page_begin, PageValue page_end,
        RowValue row_begin, RowValue row_end,
        ColValue col_begin, ColValue col_end);
```

Same as `blocked_range3d(page_begin,page_end,1,row_begin,row_end,1,col_begin, col_end,1)`.

```
blocked_range3d( blocked_range3d& range, split );
```

Basic splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges. The newly constructed `blocked_range3d` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same grain size as the original `range`. Splitting is done either by pages, rows, or columns. The choice of which axis to split is intended to cause, after repeated splitting, the subranges to approach the aspect ratio of the respective page, row, and column grain sizes.

```
blocked_range3d( blocked_range3d& range, proportional_split proportion );
```

Proportional splitting constructor.

Requirements: `is_divisible()` is true.

Effects: Partitions `range` into two subranges in the given `proportion` across one of its axes. The choice of which axis to split is made in the same way as for the basic splitting constructor; then, proportional splitting is done for the chosen axis. The second axis and the grain sizes for each subrange remain the same as in the original range.

```
bool empty() const;
```

Effects: Determines if range is empty.

Returns: `pages.empty()||rows().empty()||cols().empty()`

```
bool is_divisible() const;
```

Effects: Determines if the range can be split into subranges.

Returns: `pages().is_divisible()||rows().is_divisible()||cols().is_divisible()`

```
const page_range_type& pages() const;
```

Returns: Range containing the pages of the value space.

```
const row_range_type& rows() const;
```

Returns: Range containing the rows of the value space.

```
const col_range_type& cols() const;
```

Returns: Range containing the columns of the value space.

See also:

- *blocked_range*
- *blocked_range2d*

Partitioners

A partitioner specifies how a loop template should partition its work among threads.

auto_partitioner

[algorithms.auto_partitioner]

Specifies that a parallel loop should optimize its range subdivision based on work-stealing events.

A loop template with an `auto_partitioner` attempts to minimize range splitting while providing ample opportunities for work stealing.

The range subdivision is initially limited to S subranges, where S is proportional to the number of threads specified by the *global_contol* or *task_arena*. Each of these subranges is not divided further unless it is stolen by an idle thread. If stolen, it is further subdivided to create additional subranges. Thus a loop template with an `auto_partitioner` creates additional subranges only when it is necessary to balance a load.

An `auto_partitioner` performs sufficient splitting to balance load, not necessarily splitting as finely as `Range::is_divisible` permits. When used with classes such as `blocked_range`, the selection of an appropriate grain size is less important, and often acceptable performance can be achieved with the default grain size of 1.

The `auto_partitioner` class satisfies the *CopyConstructibe* requirement from the ISO C++ [utility.arg.requirements] section.

Tip: When using `auto_partitioner` and a `blocked_range` for a parallel loop, the body may receive a subrange larger than the grain size of the `blocked_range`. Therefore, do not assume that the grain size is an upper bound of the subrange size. Use `simple_partitioner` if an upper bound is required.

```
// Defined in header <tbb/partitioner.h>

namespace tbb {

    class auto_partitioner {
    public:
        auto_partitioner() = default;
        ~auto_partitioner() = default;
    };

}
```

affinity_partitioner

[algorithms.affinity_partitioner]

Hints that loop iterations should be assigned to threads in a way that optimizes for cache affinity.

An `affinity_partitioner` hints that execution of a loop template should use the same task affinity pattern for splitting the work as used by previous execution of the loop (or another loop) with the same `affinity_partitioner` object.

`affinity_partitioner` uses proportional splitting when it is enabled for a *Range* type.

Unlike the other partitioners, it is important that the same `affinity_partitioner` object be passed to the loop templates to be optimized for affinity.

The `affinity_partitioner` class satisfies the *CopyConstructibe* requirement from the ISO C++ [utility.arg.requirements] section.

```
// Defined in header <tbb/partitioner.h>

namespace tbb {

   class affinity_partitioner {
   public:
       affinity_partitioner() = default;
       ~affinity_partitioner() = default;
   };

}
```

See also:

- *Range named requirement*

static_partitioner

[algorithms.static_partitioner]

Specifies that a parallel algorithm should distribute the work uniformly across threads and should not do additional load balancing.

An algorithm with a `static_partitioner` distributes the range across threads in subranges of approximately equal size. The number of subranges is equal to the number of threads that can possibly participate in task execution, as specified by *global_contol* or *task_arena* classes. These subranges are not further split.

Caution: The regularity of subrange sizes is not guaranteed if the range type does not support proportional splitting, or if the grain size is set larger than the size of the range divided by the number of threads participating in task execution.

In addition, `static_partitioner` uses a deterministic task affinity pattern to hint the task scheduler how the subranges should be assigned to threads.

The `static_partitioner` class satisfies the *CopyConstructibe* requirement from the ISO C++ [utility.arg.requirements] section.

Tip: Use `static_partitioner` to:

- Parallelize small well-balanced workloads where enabling additional load balancing opportunities brings more overhead than performance benefits.
- Port OpenMP* parallel loops with `schedule(static)` if deterministic work partitioning across threads is important.

```
// Defined in header <tbb/partitioner.h>

namespace tbb {

    class static_partitioner {
    public:
        static_partitioner() = default;
        ~static_partitioner() = default;
    };

}
```

See also:

- *Range named requirement*

simple_partitioner

[algorithms.simple_partitioner]

Specifies that a parallel loop should recursively split its range until it cannot be further subdivided.

A `simple_partitioner` specifies that a loop template should recursively divide its range until for each subrange *r*, the condition `!r.is_divisible()` holds. This is the default behavior of the loop templates that take a range argument.

The `simple_partitioner` class satisfies the *CopyConstructibe* requirement from the ISO C++ [utility.arg.requirements] section.

Tip: When using `simple_partitioner` and a `blocked_range` for a parallel loop, make sure to specify an appropriate grain size for the `blocked_range`. The default grain size is 1, which may make the subranges much too small for efficient execution.

```
// Defined in header <tbb/partitioner.h>
```

(continues on next page)

(continued from previous page)

```
namespace tbb {

   class simple_partitioner {
   public:
        simple_partitioner() = default;
        ~simple_partitioner() = default;
   };

}
```

See also:

- *Range named requirement*

Split Tags

proportional split

[algorithms.proportional_split]

Type of an argument for a proportional splitting constructor of *Range*.

An argument of type `proportional_split` may be used by classes that satisfy *Range requirements* to distinguish a proportional splitting constructor from a basic splitting constructor and from a copy constructor, and to suggest a ratio in which a particular instance of the class should be split.

```
// Defined in header <tbb/blocked_range.h>
// Defined in header <tbb/blocked_range2d.h>
// Defined in header <tbb/blocked_range3d.h>
// Defined in header <tbb/partitioner.h>
// Defined in header <tbb/parallel_for.h>
// Defined in header <tbb/parallel_reduce.h>
// Defined in header <tbb/parallel_scan.h>

namespace tbb {
   class proportional_split {
   public:
        proportional_split(std::size_t _left = 1, std::size_t _right = 1);

        std::size_t left() const;
        std::size_t right() const;

        explicit operator split() const;
   };
}
```

Member functions

proportional_split (std::size_t *_left* = 1, std::size_t *_right* = 1)
Constructs a proportion with the ratio specified by coefficients *_left* and *_right*.

std::size_t **left** () **const**
Returns the size of the left part of the proportion.

std::size_t **right** () **const**
Returns the size of the right part of the proportion.

explicit operator split () **const**
Makes `proportional_split` convertible to the `split` type to use with ranges that do not support proportional splitting.

See also:

- *split*
- *Range requirements*

split

[algorithms.split]

Type of an argument for a splitting constructor of *Range*. An argument of type `split` is used to distinguish a splitting constructor from a copy constructor.

```
// Defined in header <tbb/blocked_range.h>
// Defined in header <tbb/blocked_range2d.h>
// Defined in header <tbb/blocked_range3d.h>
// Defined in header <tbb/partitioner.h>
// Defined in header <tbb/parallel_for.h>
// Defined in header <tbb/parallel_reduce.h>
// Defined in header <tbb/parallel_scan.h>

class split;
```

See also:

- *Range requirements*

9.2.3 Flow Graph

[flow_graph]

In addition to loop parallelism, the oneAPI Threading Building Blocks (oneTBB) library also supports graph parallelism. With this feature, highly scalable and completely sequential graphs can be created.

There are three types of components used to implement a graph:

- A `graph` class instance
- Nodes
- Ports and edges

Graph Class

The `graph` class instance owns all the tasks created on behalf of the flow graph. Users can wait on the `graph` if they need to wait for the completion of all of the tasks related to the flow graph execution. Users can also register external interactions with the `graph` and run tasks under the ownership of the flow graph.

graph

[flow_graph.graph]

Class that serves as a handle to a flow graph of nodes and edges.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    class graph {
    public:
        graph();
        graph(task_group_context& context);
        ~graph();

        void wait_for_all();

        void reset(reset_flags f = rf_reset_protocol);
        void cancel();
        bool is_cancelled();
        bool exception_thrown();
    };

} // namespace flow
} // namespace tbb
```

reset_flags enumeration

reset_flags Enumeration

[flow_graph.reset_flags]

A `reset_flags` enumeration represents flags that can be passed to the `graph::reset()` function.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    enum reset_flags {
        rf_reset_protocol = /*implementation-defined*/,
        rf_reset_bodies = /*implementation-defined*/,
        rf_clear_edges = /*implementation-defined*/
    };

} // namespace flow
} // namespace tbb
```

Its enumerated values and their meanings are as follows:

- `rf_reset_protocol` - All buffers are emptied, internal state of nodes reinitialized. All calls to `reset()` perform these actions.
- `rf_reset_bodies` - When nodes with bodies are created, the body specified in the constructor is copied and preserved. When `rf_reset_bodies` is specified, the current body of the node is deleted and replaced with a copy of the body saved during construction.

  > **Caution:** If the body contains state which has an external component (such as a file descriptor), the node may not behave the same on re-execution of the graph after body replacement. In this case, the node should be re-created.

- `rf_clear_edges` - All edges are removed from the graph.

Member functions

`graph`(*task_group_context* &*group*)
Constructs a graph with no nodes. If `group` is specified, the graph tasks are executed in this group. By default, the graph is executed in a bound context of its own.

`~graph`()
Calls `wait_for_all()` on the graph, then destroys the graph.

void **`wait_for_all`**()
Blocks execution until all tasks associated with the graph have completed or cancelled.

void **`reset`**(reset_flags *f* = rf_reset_protocol)
Resets the graph according to the specified flags. Flags to `reset()` can be combined with bitwise-`or`.

void **`cancel`**()
Cancels all tasks in the graph.

bool **`is_cancelled`**()
Returns: `true` if the graph was cancelled during the last call to `wait_for_all()`; `false`, otherwise.

bool **`exception_thrown`**()
Returns: `true` if during the last call to `wait_for_all()` an exception was thrown; `false`, otherwise.

Nodes

Abstract Interfaces

To be used as a graph node type, a class needs to inherit certain abstract types and implement the corresponding interfaces. `graph_node` is the base class for any other node type; its interfaces always have to be implemented. If a node sends messages to other nodes, it has to implement the `sender` interface, while with the `receiver` interface the node may accept messages. For nodes that have multiple inputs and/or outputs, each input port is a `receiver` and each output port is a `sender`.

graph_node

[flow_graph.graph_node]

A base class for all graph nodes.

```
namespace tbb {
namespace flow {

    class graph_node {
    public:
        explicit graph_node( graph &g );
        virtual ~graph_node();
    };

} // namespace flow
} // namespace tbb
```

The `graph_node` class is a base class for all flow graph nodes. The virtual destructor allows flow graph nodes to be destroyed through pointers to `graph_node`. For example, a `vector< graph_node * >` can be used to hold the addresses of flow graph nodes that will need to be destroyed later.

sender

[flow_graph.sender]

A base class for all nodes that may send messages.

```
namespace tbb {
namespace flow {

    template< typename T >
    class sender { /*unspecified*/ };

} // namespace flow
} // namespace tbb
```

The `T` type is a message type.

receiver

[flow_graph.receiver]

A base class for all nodes that may receive messages.

```
namespace tbb {
namespace flow {

    template< typename T >
    class receiver { /*unspecified*/ };

} // namespace flow
} // namespace tbb
```

The `T` type is a message type.

Properties

Every node in a flow graph has its own properties.

Forwarding and Buffering

[flow_graph.forwarding_and_buffering]

Forwarding

In a `flow::graph`, nodes that forward messages to successors have one of two possible forwarding policies, which are a property of the node:

- **broadcast-push** - the message will be pushed to as many successors as will accept the message. If no successor accepts the message, the fate of the message depends on the output buffering policy of the node.
- **single-push** - if the message is accepted by a successor, no further push of that message will occur. If a successor rejects the message, the next successor in the set is tried. This continues until a successor accepts the message, or all successors have been attempted. If no successor accepts the message, it will be retained for a possible future resend. Message that is successfully transferred to a successor is removed from the node.

Buffering

There are two policies for handling a message that cannot be pushed to any successor:

- **buffering** - if no successor accepts a message, it is stored so subsequent node processing can use it. Nodes that buffer outputs have "yes" in the "try_get()?" column below.
- **discarding** - if no successor accepts a message, it is discarded and has no further effect on graph execution. Nodes that discard outputs have "no" in the "try_get()?" column below.

The following table lists the policies of each node:

Table 4: Buffering and Forwarding properties summary

Node	try_get()?	Forwarding
Functional Nodes		
input_node	yes	broadcast-push
function_node<rejecting>	no	broadcast-push
function_node<queueing>	no	broadcast-push
continue_node	no	broadcast-push
multifunction_node<rejecting>	no	broadcast-push
multifunction_node<queueing>	no	broadcast-push
Buffering Nodes		
buffer_node	yes	single-push
priority_queue_node	yes	single-push
queue_node	yes	single-push
sequencer_node	yes	single-push
overwrite_node	yes	broadcast-push
write_once_node	yes	broadcast-push
Split/Join Nodes		
join_node<queueing>	yes	broadcast-push
join_node<reserving>	yes	broadcast-push
join_node<tag_matching>	yes	broadcast-push
split_node	no	broadcast-push
indexer_node	no	broadcast-push
Other Nodes		
broadcast_node	no	broadcast-push
limiter_node	no	broadcast-push

Functional Nodes

Functional nodes do computations in response to input messages (if any), and send the result or a signal to their successors.

continue_node

[flow_graph.continue_node]

A node that executes a specified body object when triggered.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename Output, typename Policy = /*implementation-defined*/ >
    class continue_node : public graph_node, public receiver<continue_msg>, public␣
↪sender<Output> {
    public:
        template<typename Body>
        continue_node( graph &g, Body body, node_priority_t priority = no_priority );
        template<typename Body>
        continue_node( graph &g, Body body, Policy /*unspecified*/ = Policy(),
                       node_priority_t priority = no_priority );
```

(continues on next page)

(continued from previous page)

```
        template<typename Body>
        continue_node( graph &g, int number_of_predecessors, Body body,
                       node_priority_t priority = no_priority );
        template<typename Body>
        continue_node( graph &g, int number_of_predecessors, Body body,
                       Policy /*unspecified*/ = Policy(), node_priority_t priority =␣
↪no_priority );

        continue_node( const continue_node &src );
        ~continue_node();

        bool try_put( const input_type &v );
        bool try_get( output_type &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `Output` must meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.
- The type `Policy` can be specified as *lightweight policy* or defaulted.
- The type `Body` must meet the *ContinueNodeBody requirements*.

A `continue_node` is a `graph_node`, `receiver<continue_msg>`, and `sender<Output>`.

This node is used for nodes that wait for their predecessors to complete before executing, but no explicit data is passed across the incoming edges.

A `continue_node` maintains an internal threshold that defines the number of predecessors. This value can be provided at construction. Call of the *make_edge function* with `continue_node` as a receiver increases its threshold. Call of the *remove_edge function* with `continue_node` as a receiver decreases it.

Each time the number of `try_put()` calls reaches the defined threshold, node's `body` is called and the node starts counting the number of `try_put()` calls from the beginning.

`continue_node` has a *discarding* and *broadcast-push properties*.

The body object passed to a `continue_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body function* can be used to obtain an updated copy.

Member functions

```
template<typename Body>
continue_node( graph &g, Body body, node_priority_t priority = no_priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to 0.

This function specifies *node priority*.

```
template<typename Body>
continue_node( graph &g, Body body, Policy /*unspecified*/ = Policy(),
               node_priority_t priority = no_priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to 0.

This function specifies *lightweight policy* and *node priority*.

```
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body,
               node_priority_t priority = no_priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to `number_of_predecessors`.

This function specifies *node priority*.

```
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body,
               Policy /*unspecified*/ = Policy(), node_priority_t priority = no_
↪priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to `number_of_predecessors`.

This function specifies *lightweight policy* and *node priority*.

```
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to `number_of_predecessors`.

```
continue_node( const continue_node &src )
```

Constructs a `continue_node` that has the same initial state that `src` had after its construction. It does not copy the current count of `try_puts` received, or the current known number of predecessors. The `continue_node` that is constructed has a reference to the same `graph` object as `src`, has a copy of the initial `body` used by `src`, and only has a non-zero threshold if `src` is constructed with a non-zero threshold.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction.

```
bool try_put( const Input &v )
```

Increments the count of `try_put()` calls received. If the incremented count is equal to the number of known predecessors, performs the `body` function object execution. It does not wait for the execution of the body to complete.

Returns: `true`

```
bool try_get( Output &v )
```

Returns: `false`

Deduction Guides

```
template <typename Body, typename Policy>
continue_node(graph&, Body, Policy, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>,␣
↪Policy>;

template <typename Body, typename Policy>
continue_node(graph&, int, Body, Policy, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>,␣
↪Policy>;

template <typename Body>
continue_node(graph&, Body, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, /
↪*default-policy*/>;

template <typename Body>
continue_node(graph&, int, Body, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, /
↪*default-policy*/>;
```

Where:

- `continue_output_t<Output>` is an alias to *Output* template argument type. If *Output* specified as `void`, `continue_output_t<Output>` is an alias to `continue_msg` type.

Example

A set of `continue_nodes` forms a *Dependency Flow Graph*.

function_node

[flow_graph.function_node]

A node that executes a user-provided body on incoming messages.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template < typename Input, typename Output = continue_msg, typename Policy = /
↪*implementation-defined*/ >
    class function_node : public graph_node, public receiver<Input>, public sender
↪<Output> {
    public:
        template<typename Body>
        function_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/
↪ = Policy(),
                       node_priority_t priority = no_priority );
        template<typename Body>
        function_node( graph &g, size_t concurrency, Body body,
                       node_priority_t priority = no_priority );
        ~fuction_node();
```

(continues on next page)

(continued from previous page)

```
        function_node( const function_node &src );

        bool try_put( const Input &v );
        bool try_get( Output &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The `Input` type must meet the *DefaultConstructible* requirements from [defaultconstructible] and the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.
- The `Output` type must meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.
- The type `Policy` may be specified as *lightweight, queueing and rejecting policies* or defaulted.
- The type `Body` must meet the *FunctionNodeBody requirements*.

`function_node` has a user-settable concurrency limit. It can be set to one of *predefined values*. The user can also provide a value of type `std::size_t` to limit concurrency to a value between 1 and *tbb::flow::unlimited*.

Messages that cannot be immediately processed due to concurrency limits are handled according to the *Policy* template argument.

`function_node` is a `graph_node`, `receiver<Input>`, and `sender<Output>`.

`function_node` has a *discarding* and *broadcast-push* *properties*.

The body object passed to a `function_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body* function can be used to obtain an updated copy.

Member functions

```
template<typename Body>
function_node( graph &g, size_t concurrency, Body body,
               node_priority_t priority = no_priority );
```

Constructs a `function_node` that invokes a copy of `body`. Most of `concurrency` calls to `body` can be made concurrently.

Use this function to specify *node priority*.

```
template<typename Body>
function_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ =␣
↪Policy(),
               node_priority_t priority = no_priority );
```

Constructs a `function_node` that invokes a copy of `body`. Most of `concurrency` calls to `body` can be made concurrently.

Use this function to specify *policy* and *node priority*.

```
function_node( const function_node &src )
```

Constructs a `function_node` that has the same initial state that `src` had when it was constructed. The `function_node` that is constructed has a reference to the same `graph` object as `src`, has a copy of the initial body used by `src`, and has the same concurrency threshold as `src`. The predecessors and successors of `src` are not copied.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction. Changes made to member variables in `src`'s body after the construction of `src` do not affect the body of the new `function_node`.

```
bool try_put( const Input &v )
```

If the concurrency limit allows, executes the user-provided body on the incoming message `v`. Otherwise, depending on the policy of the node, either queues the incoming message `v` or rejects it.

Returns: `true` if the input was accepted; and `false`, otherwise.

```
bool try_get( Output &v )
```

Returns: `false`

Deduction Guides

```
template <typename Body, typename Policy>
function_node(graph&, size_t, Body, Policy, node_priority_t = no_priority)
    ->function_node<std::decay_t<input_t<Body>>, output_t<Body>, Policy>;

template <typename Body>
function_node(graph&, size_t, Body, node_priority_t = no_priority)
    ->function_node<std::decay_t<input_t<Body>>, output_t<Body>, /*default-policy*/>;
```

Where:

- `input_t` is an alias to `Body` input argument type.
- `output_t` is an alias to `Body` return type.

Example

Data Flow Graph example illustrates how `function_node` performs computation on input data and passes the result to successors.

input_node

[flow_graph.input_node]

A node that generates messages by invoking the user-provided functor and broadcasts the result to all of its successors.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template < typename Output >
    class input_node : public graph_node, public sender<Output> {
    public:
        template< typename Body >
        input_node( graph &g, Body body );
        input_node( const input_node &src );
        ~input_node();

        void activate();
        bool try_get( Output &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The `Output` type must meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Body` must meet the *InputNodeBody requirements*.

This node can have no predecessors. It executes a user-provided `body` function object to generate messages that are broadcast to all successors. It is a serial node and never calls its `body` concurrently. This node can buffer a single item. If no successor accepts an item that it has generated, the message is buffered and provided to successors before a new item is generated.

`input_node` is a `graph_node` and `sender<Output>`.

`input_node` has a *buffering* and *broadcast-push* *properties*.

An `input_node` continues to invoke `body` and broadcast messages until the `body` toggles `fc.stop()` or it has no valid successors. A message may be generated and then rejected by all successors. In this case, the message is buffered and will be the next message sent once a successor is added to the node or `try_get` is called. Calls to `try_get` return a message from the buffer, or invoke `body` to attempt to generate a new message. A call to `body` is made only when the buffer is empty.

The body object passed to an `input_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body function* can be used to obtain an updated copy.

Member functions

template<typename **Body**>
input_node(*graph* &*g*, *Body body*)
Constructs an `input_node` that invokes `body`. By default, the node is created in an inactive state, which means that messages are not generated until a call to `activate` is made.

input_node(**const** *input_node* &*src*)
Constructs an `input_node` that has the same initial state that `src` had when it was constructed. The `input_node` that is constructed has a reference to the same `graph` object as `src`, has a copy of the initial body used by `src`, and has the same initial active state as `src`. The successors of `src` are not copied.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction. Changes made to member variables in `src` body after the construction of `src` do not affect the body of the new `input_node`.

void **activate**()
Sets the `input_node` to the active state, which enables messages generation.

bool **try_get**(Output &*v*)
Copies the message from the buffer to `v` if available, or, if the node is in active state, invokes `body` to attempt to generate a new message that will be copied into `v`.

Returns: `true` if a message is copied to `v`; `false`, otherwise.

Deduction Guides

```
template <typename Body>
input_node(graph&, Body) -> input_node<std::decay_t<input_t<Body>>>;
```

Where:

- `input_t` is an alias to `Body` input argument type.

multifunction_node

[flow_graph.multifunction_node]

A node that used for nodes that receive messages at a single input port and may generate one or more messages that are broadcast to successors.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template < typename Input, typename Output, typename Policy = /*implementation-
↪defined*/ >
    class multifunction_node : public graph_node, public receiver<Input> {
    public:
        template<typename Body>
        multifunction_node( graph &g, size_t concurrency, Body body, Policy /
↪*unspecified*/ = Policy(),
                            node_priority_t priority = no_priority );
        template<typename Body>
        multifunction_node( graph &g, size_t concurrency, Body body,
```

(continues on next page)

(continued from previous page)

```
                            node_priority_t priority = no_priority );

        multifunction_node( const multifunction_node& other );
        ~multifunction_node();

        bool try_put( const Input &v );

        using output_ports_type = /*implementation-defined*/;
        output_ports_type& output_ports();
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The `Input` type must meet the *DefaultConstructible* requirements from [defaultconstructible] and the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.
- The type `Policy` can be specified as *lightweight, queueing and rejecting policies* or defaulted.
- The type `Body` must meet the *MultifunctionNodeBody requirements*.

`multifunction_node` has a user-settable concurrency limit. It can be set to one of *predefined values*. The user can also provide a value of type `std::size_t` to limit concurrency to a value between 1 and *tbb::flow::unlimited*.

When the concurrency limit allows, it executes the user-provided body on incoming messages. The body can create one or more output messages and broadcast them to successors.

`multifunction_node` is a `graph_node`, `receiver<InputType>` and has a tuple of `sender<Output>` outputs.

`multifunction_node` has a *discarding* and *broadcast-push properties*.

The body object passed to a `multifunction_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body function* can be used to obtain an updated copy.

Member types

`output_ports_type` is an alias to a tuple of output ports.

Member functions

```
template<typename Body>
multifunction_node( graph &g, size_t concurrency, Body body,
                    node_priority_t priority = no_priority );
```

Constructs a `multifunction_node` that invokes a copy of `body`. Most `concurrency` calls to `body` can be made concurrently.

Use this function to specify *node priority*.

```
template<typename Body>
multifunction_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ =␣
↪Policy(),
                    node_priority_t priority = no_priority );
```

Constructs a `multifunction_node` that invokes a copy of `body`. Most `concurrency` calls to `body` can be made concurrently.

Use this function to specify a *policy* and *node priority*.

```
multifunction_node( const multifunction_node &src )
```

Constructs a `multifunction_node` that has the same initial state that `other` had when it was constructed. The `multifunction_node` that is constructed has a reference to the same `graph` object as `other`, has a copy of the initial `body` used by `other`, and has the same concurrency threshold as `other`. The predecessors and successors of `other` are not copied.

The new body object is copy-constructed from a copy of the original body provided to `other` at its construction. Changes made to member variables in `other` body after the construction of `other` do not affect the body of the new `multifunction_node`.

```
bool try_put( const input_type &v )
```

If the concurrency limit allows, executes the user-provided body on the incoming message `v`. Otherwise, depending on the policy of the node, either queues the incoming message `v` or rejects it.

Returns: `true` if the input was accepted; `false`, otherwise.

```
output_ports_type& output_ports();
```

Returns: a tuple of output ports.

async_node

[flow_graph.async_node]

A node that enables communication between a flow graph and an external activity managed by the user or another runtime.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template < typename Input, typename Output, typename Policy = /*implemetation-
↪defined*/ >
    class async_node : public graph_node, public receiver<Input>, public sender
↪<Output> {
    public:
        template<typename Body>
        async_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ =␣
↪Policy(),
```

(continues on next page)

(continued from previous page)

```
                    node_priority_t priority = no_priority );
        template<typename Body>
        async_node( graph &g, size_t concurrency, Body body, node_priority_t priority␣
↪= no_priority );

        async_node( const async_node& src );
        ~async_node();

        using gateway_type = /*implementation-defined*/;
        gateway_type& gateway();

        bool try_put( const input_type& v );
        bool try_get( output_type& v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The `Input` type must meet the *DefaultConstructible* requirements from [defaultconstructible] and the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.
- The type `Policy` can be specified as *lightweight, queueing and rejecting policies* or defaulted.
- The type `Body` must meet the *AsyncNodeBody requirements*.

`async_node` executes a user-provided body on incoming messages. The body typically submits the messages to an external activity for processing outside of the graph. It is responsibility of `body` to be able to pass the message to an external activity. This node also provides the `gateway_type` interface that allows an external activity to communicate with the flow graph.

`async_node` is a `graph_node`, `receiver<Input>`, and a `sender<Output>`.

`async_node` has a *discarding* and *broadcast-push properties*.

`async_node` has a user-settable concurrency limit, which can be set to one of *predefined values*. The user can also provide a value of type `std::size_t` to limit concurrency to a value between 1 and *tbb::flow::unlimited*.

The body object passed to a `async_node` is copied. Updates to member variables do not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy_body* function can be used to obtain an updated copy.

Member types

`gateway_type` meets the *GatewayType requirements*.

Member functions

```
template<typename Body>
async_node( graph &g, size_t concurrency, Body body,
               node_priority_t priority = no_priority );
```

Constructs an `async_node` that invokes a copy of `body`. The `concurrency` value limits the number of simultaneous `body` invocations for the node.

This function specifies *node priority*.

```
template<typename Body>
async_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ =␣
↪Policy(),
               node_priority_t priority = no_priority );
```

Constructs a `async_node` that invokes a copy of `body`. Most `concurrency` calls to `body` can be made concurrently.

This function specifies a *policy* and *node priority*.

```
async_node( const async_node &src )
```

Constructs an `async_node` that has the same initial state that `src` had when it was constructed. The `async_node` that is constructed has a reference to the same `graph` object as `src`, has a copy of the initial body used by `src`, and has the same concurrency threshold as `src`. The predecessors and successors of `src` are not copied.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction. Changes made to member variables in `src`'s body after the construction of `src` do not affect the body of the new `async_node`.

```
gateway_type& gateway()
```

Returns reference to the `gateway_type` interface.

```
bool try_put( const input_type& v )
```

If the concurrency limit allows, executes the user-provided body on the incoming message `v`. Otherwise, depending on the policy of the node, either queues the incoming message `v` or rejects it.

Returns: `true` if the input was accepted; and `false`, otherwise.

```
bool try_get( output_type& v )
```

Returns: `false`

Auxiliary

Function Nodes Policies

[flow_graph.function_node_policies]

`function_node`, `multifunction_node`, `async_node` and `continue_node` can be specified by the `Policy` parameter, which is represented as a set of tag classes. This parameter affects behavior of node execution.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    class queueing { /*unspecified*/ };
    class rejecting { /*unspecified*/ };
    class lightweight { /*unspecified*/ };
    class queueing_lightweight { /*unspecified*/ };
    class rejecting_lightweight { /*unspecified*/ };

} // namespace flow
} // namespace tbb
```

Each policy class satisfies the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.

Queueing

This policy defines behavior for input messages acceptance. The `queueing` policy means that input messages that cannot be processed right away are kept to be processed when possible.

Rejecting

This policy defines behavior for input messages acceptance. The `rejecting` policy means that input messages that cannot be processed right away are not accepted by the node and it is responsibility of a predecessor to handle this.

Lightweight

This policy helps to reduce the overhead associated with the execution scheduling of the node.

For functional nodes that have a default value for the `Policy` template parameter, specifying the `lightweight` policy results in extending the behavior of the default value of `Policy` with the behavior defined by the `lightweight` policy. For example, if the default value of `Policy` is `queueing`, specifying `lightweight` as the `Policy` value is equivalent to specifying `queueing_lightweight`.

Example

The example below shows the application of the `lightweight` policy to a graph with a pipeline topology. It is reasonable to apply the `lightweight` policy to the second and third nodes because the bodies of these nodes are small. This allows the second and third nodes to execute without task scheduling overhead. The `lightweight` policy is not specified for the first node in order to permit concurrent invocations of the graph.

```
#include "tbb/flow_graph.h"

int main() {
    using namespace tbb::flow;

    graph g;

    function_node< int, int > add( g, unlimited, [](const int &v) {
        return v+1;
    } );
    function_node< int, int, lightweight > multiply( g, unlimited, [](const int &v) {
        return v*2;
    } );
    function_node< int, int, lightweight > cube( g, unlimited, [](const int &v) {
        return v*v*v;
    } );

    make_edge(add, multiply);
    make_edge(multiply, cube);

    for(int i = 1; i <= 10; ++i)
        add.try_put(i);
    g.wait_for_all();

    return 0;
}
```

Nodes Priorities

[flow_graph.node_priorities]

Flow graph provides interface for setting relative priorities at construction of flow graph functional nodes, guiding threads that execute the graph to prefer nodes with higher priority.

```
namespace tbb {
namespace flow {

    typedef unsigned int node_priority_t;

    const node_priority_t no_priority = node_priority_t(0);

} // namespace flow
} // namespace tbb
```

`function_node`, `multifunction_node`, `async_node` and `continue_node` has a constructor with parameter of `node_priority_t` type, which sets the node priority in the graph: the larger the specified value for the parameter, the higher the priority. The special constant value `no_priority`, which is also the default value of the parameter, switches priority off for a particular node.

For a particular graph, tasks to execute the nodes whose priority is specified have precedence over tasks for the nodes with lower or no priority value set. When looking for a task to execute, a thread chooses the one with the highest priority from those in the graph that are available for execution.

Example

The following basic example demonstrates prioritization of one path in the graph over the other, which may help to improve overall performance of the graph.

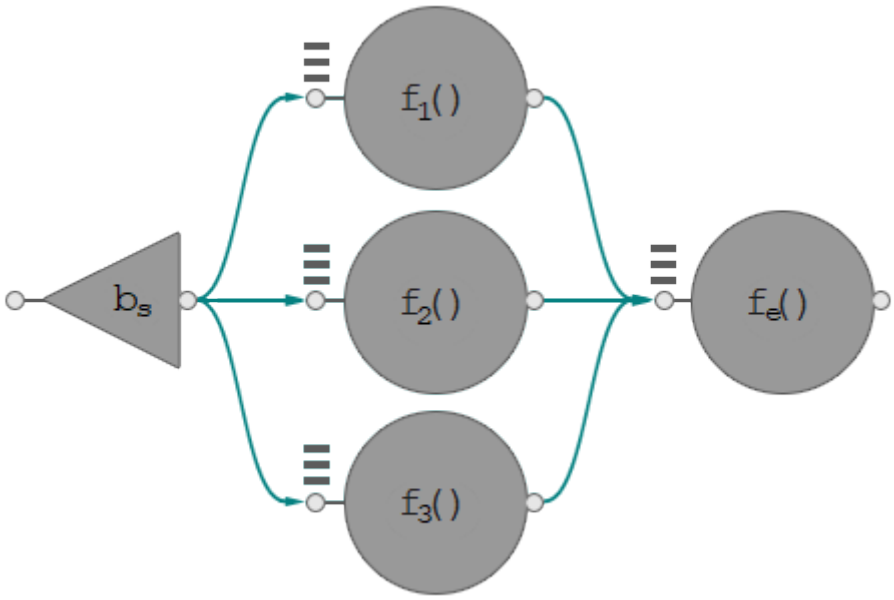


Fig. 1: Dependency flow graph with a critical path.

Consider executing the graph from the picture above using two threads. Assume that the nodes `f1` and `f3` take equal time to execute, while the node `f2` takes longer. That makes the nodes `bs`, `f2`, and `fe` constitute the critical path in this graph. Due to the non-deterministic behavior in selection of the tasks, oneTBB might execute nodes `f1` and `f3` in parallel first, which would make the whole graph execution time last longer than the case when one of the threads chooses the node `f2` just after the broadcast node. By setting a higher priority on node `f2`, threads are guided to take the critical path task earlier, thus reducing overall execution time.

```
#include <iostream>
#include <cmath>

#include "tbb/tick_count.h"
#include "tbb/global_control.h"

#include "tbb/flow_graph.h"

void spin_for( double delta_seconds ) {
    tbb::tick_count start = tbb::tick_count::now();
    while( (tbb::tick_count::now() - start).seconds() < delta_seconds ) ;
}

static const double unit_of_time = 0.1;

struct Body {
    unsigned factor;
    Body( unsigned times ) : factor( times ) {}
    void operator()( const tbb::flow::continue_msg& ) {
```

(continues on next page)

(continued from previous page)

```
        // body execution takes 'factor' units of time
        spin_for( factor * unit_of_time );
    }
};

int main() {
    using namespace tbb::flow;

    const int max_threads = 2;
    tbb::global_control control(tbb::global_control::max_allowed_parallelism, max_
↪threads);

    graph g;

    broadcast_node<continue_msg> bs(g);

    continue_node<continue_msg> f1(g, Body(5));

    // f2 is a heavy one and takes the most execution time as compared to the other␣
↪nodes in the
    // graph. Therefore, let the graph start this node as soon as possible by␣
↪prioritizing it over
    // the other nodes.
    continue_node<continue_msg> f2(g, Body(10), node_priority_t(1));

    continue_node<continue_msg> f3(g, Body(5));

    continue_node<continue_msg> fe(g, Body(7));

    make_edge( bs, f1 );
    make_edge( bs, f2 );
    make_edge( bs, f3 );

    make_edge( f1, fe );
    make_edge( f2, fe );
    make_edge( f3, fe );

    tbb::tick_count start = tbb::tick_count::now();

    bs.try_put( continue_msg() );
    g.wait_for_all();

    double elapsed = std::floor((tbb::tick_count::now() - start).seconds() / unit_of_
↪time);

    std::cout << "Elapsed approximately " << elapsed << " units of time" << std::endl;

    return 0;
}
```

Predefined Concurrency Limits

[flow_graph.concurrency_limits]

Predefined constants that can be used as `function_node`, `multifunction_node`, and `async_node` constructors arguments to define concurrency limit.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    std::size_t unlimited = /*implementation-defined*/;
    std::size_t serial = /*implementation-defined*/;

} // namespace flow
} // namespace tbb
```

`unlimited` concurrency allows an unlimited number of invocations of the body to execute concurrently.

`serial` concurrency allows only a single call of body to execute concurrently.

copy_body

[flow_graph.copy_body]

`copy_body` is a function template that returns a copy of the body function object from the following nodes:

- *continue_node*
- *function_node*
- *multifunction_node*
- *input_node*
- *async_node*

```
namespace tbb {
namespace flow {

    // Defined in header <tbb/flow_graph.h>

    template< typename Body, typename Node >
    Body copy_body( Node &n );

} // namespace flow
} // namespace tbb
```

Buffering Nodes

Buffering nodes are designed to accumulate input messages and pass them to successors in a predefined order, depending on the node type.

overwrite_node

[flow_graph.overwrite_node]

A node that is a buffer of a single item that can be overwritten.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename T>
    class overwrite_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit overwrite_node( graph &g );
        overwrite_node( const overwrite_node &src );
        ~overwrite_node();

        bool try_put( const T &v );
        bool try_get( T &v );

        bool is_valid( );
        void clear( );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `T` must meet the *DefaultConstructible* requirements from [defaultconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

This type of node buffers a single item of type `T`. The value is initially invalid. Gets from the node are non-destructive.

`overwrite_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

`overwrite_node` has a *buffering* and *broadcast-push* *properties*.

`overwrite_node` allows overwriting its single item buffer.

Member functions

explicit overwrite_node(*graph &g*)
: Constructs an object of type `overwrite_node` that belongs to the graph `g` with an invalid internal buffer item.

overwrite_node(**const** *overwrite_node &src*)
: Constructs an object of type `overwrite_node` that belongs to the graph `g` with an invalid internal buffer item. The buffered value and list of successors are not copied from `src`.

~overwrite_node()
: Destroys the overwrite_node.

bool **try_put**(**const** T &*v*)
Stores `v` in the internal single item buffer and calls `try_put(v)` on all successors.

Returns: `true`

bool **try_get**(T &*v*)
If the internal buffer is valid, assigns the value to `v`.

Returns:`true` if `v` is assigned to; `false`, otherwise.

bool **is_valid**()
Returns: `true` if the buffer holds a valid value; `false`, otherwise.

void **clear**()
Invalidates the value held in the buffer.

Examples

The example demonstrates `overwrite_node` as a single-value storage that might be updated. Data can be accessed with direct `try_get()` call.

```
#include "tbb/flow_graph.h"

int main() {
    const int data_limit = 20;
    int count = 0;

    tbb::flow::graph g;

    tbb::flow::function_node< int, int > data_set_preparation(g,
        tbb::flow::unlimited, []( int data ) {
            printf("Prepare large data set and keep it inside node storage\n");
            return data;
        });

    tbb::flow::overwrite_node< int > overwrite_storage(g);

    tbb::flow::source_node<int> data_generator(g,
        [&]( int& v ) -> bool {
            if ( count < data_limit ) {
                ++count;
                v = count;
                return true;
            } else {
                return false;
            }
        });

    tbb::flow::function_node< int > process(g, tbb::flow::unlimited,
        [&]( const int& data) {
            int data_from_storage = 0;
            overwrite_storage.try_get(data_from_storage);
            printf("Data from a storage: %d\n", data_from_storage);
            printf("Data to process: %d\n", data);
        });

    tbb::flow::make_edge(data_set_preparation, overwrite_storage);
    tbb::flow::make_edge(data_generator, process);
```

(continues on next page)

(continued from previous page)

```
    data_set_preparation.try_put(1);
    data_generator.activate();

    g.wait_for_all();

    return 0;
}
```

`overwrite_node` supports reserving `join_node` as its successor. See the example in *the example section of write_once_node*.

write_once_node

[flow_graph.write_once_node]

A node that is a buffer of a single item that cannot be overwritten.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename T >
    class write_once_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit write_once_node( graph &g );
        write_once_node( const write_once_node &src );
        ~write_once_node();

        bool try_put( const T &v );
        bool try_get( T &v );

        bool is_valid( );
        void clear( );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The `T` type must meet the *DefaultConstructible* requirements from [defaultconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

This type of node buffers a single item of type `T`. The value is initially invalid. Gets from the node are non-destructive.

`write_once_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

`write_once_node` has a *buffering* and *broadcast-push* *properties*.

`write_once_node` does not allow overwriting its single item buffer.

Member functions

explicit write_once_node(*graph &g*)
Constructs an object of type `write_once_node` that belongs to the graph `g`, with an invalid internal buffer item.

write_once_node(**const** *write_once_node &src*)
Constructs an object of type `write_once_node` with an invalid internal buffer item. The buffered value and list of successors is not copied from `src`.

~write_once_node()
Destroys the `write_once_node`.

bool **try_put**(**const** T *&v*)
Stores `v` in the internal single item buffer if it does not contain a valid value already. If a new value is set, the node broadcast it to all successors.

Returns: `true` for the first time after construction or a call to `clear()`; `false`, otherwise.

bool **try_get**(T *&v*)
If the internal buffer is valid, assigns the value to `v`.

Returns: `true` if `v` is assigned to; `false`, otherwise.

bool **is_valid**()
Returns: `true` if the buffer holds a valid value; `false`, otherwise.

void **clear**()
Invalidates the value held in the buffer.

Example

Usage scenario is similar to *overwrite_node* but an internal buffer can be updated only after `clear()` call. The following example shows the possibility to connect the node to a reserving `join_node`, avoiding direct calls to the `try_get()` method from the body of the successor node.

```
#include "tbb/flow_graph.h"

typedef int data_type;

int main() {
    using namespace tbb::flow;

    graph g;

    function_node<data_type, data_type> static_result_computer_n(
        g, serial,
        [&](const data_type& msg) {
            // compute the result using incoming message and pass it further, e.g.:
            data_type result = data_type((msg << 2 + 3) / 4);
            return result;
        });
    write_once_node<data_type> write_once_n(g); // for buffering once computed value

    buffer_node<data_type> buffer_n(g);
    join_node<tuple<data_type, data_type>, reserving> join_n(g);

    function_node<tuple<data_type, data_type>> consumer_n(
```

(continues on next page)

(continued from previous page)

```
        g, unlimited,
        [&](const tuple<data_type, data_type>& arg) {
            // use the precomputed static result along with dynamic data
            data_type precomputed_result = get<0>(arg);
            data_type dynamic_data = get<1>(arg);
        });

    make_edge(static_result_computer_n, write_once_n);
    make_edge(write_once_n, input_port<0>(join_n));
    make_edge(buffer_n, input_port<1>(join_n));
    make_edge(join_n, consumer_n);

    // do one-time calculation that will be reused many times further in the graph
    static_result_computer_n.try_put(1);

    for (int i = 0; i < 100; i++) {
        buffer_n.try_put(1);
    }

    g.wait_for_all();

    return 0;
}
```

buffer_node

[flow_graph.buffer_node]

A node that is an unbounded buffer of messages. Messages are forwarded in an arbitrary order.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename T>
    class buffer_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit buffer_node( graph &g );
        buffer_node( const buffer_node &src );
        ~buffer_node();

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `T` must meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

`buffer_node` is a `graph_node`, `receiver<T>`, and `sender<T>`.

`buffer_node` has a *buffering* and *single-push* *properties*.

`buffer_node` forwards messages in an arbitrary order to a single successor in its successor set.

Member functions

explicit buffer_node(*graph &g*)
Constructs an empty `buffer_node` that belongs to the graph `g`.

explicit buffer_node(**const** *buffer_node &src*)
Constructs an empty `buffer_node` that belongs to the same graph `g` as `src`. Any intermediate state of `src`, including its links to predecessors and successors, is not copied.

bool **try_put**(**const** T *&v*)
Adds `v` to the set of items managed by the node, and tries forwarding it to a successor.

Returns: `true`

bool **try_get**(T *&v*)
Returns: `true` if an item can be removed from the node and assigned to `v`. Returns `false` if there is no non-reserved item currently in the node.

queue_node

[flow_graph.queue_node]

A node that forwards messages in a first-in first-out (FIFO) order.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template <typename T >
    class queue_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit queue_node( graph &g );
        queue_node( const queue_node &src );
        ~queue_node();

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `T` must meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

`queue_node` forwards messages in a FIFO order to a single successor in its successor set.

`queue_node` is a `graph_node`, `receiver` and `sender`.

`queue_node` has a *buffering* and *single-push properties*.

Member functions

explicit queue_node (*graph &g*)
Constructs an empty `queue_node` that belongs to the graph `g`.

queue_node (**const** *queue_node &src*)
Constructs an empty `queue_node` that belongs to the same graph `g` as `src`. Any intermediate state of `src`, including its links to predecessors and successors, is not copied.

bool **try_put** (**const** T *&v*)
Adds `v` to the set of items managed by the node, and tries forwarding the least recently added item to a successor.

Returns: `true`.

bool **try_get** (T *&v*)
Returns: `true` if an item can be taken from the node and assigned to `v`. Returns `false` if there is no item currently in the `queue_node` or if the node is reserved.

Example

Usage scenario is similar to *buffer_node* except that messages are passed in first-in first-out (FIFO) order.

priority_queue_node

[flow_graph.priority_queue_node]

A class template that forwards messages in a priority order.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename T, typename Compare = std::less<T>>
    class priority_queue_node : public graph_node, public receiver<T>, public sender
↪<T> {
    public:
        typedef size_t size_type;
        explicit priority_queue_node( graph &g );
        priority_queue_node( const priority_queue_node &src );
        ~priority_queue_node();

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `T` must meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Compare` must meet the *Compare* type requirements from [alg.sorting] ISO C++ Standard section. If `Compare` instance throws an exception, then behavior is undefined.

The next message to be forwarded has the largest priority as determined by the `Compare` template argument.

`priority_queue_node` is a `graph_node`, `receiver<T>`, and `sender<T>`.

`priority_queue_node` has a *buffering* and *single-push properties*.

Member functions

explicit priority_queue_node(*graph &g*)
Constructs an empty `priority_queue_node` that belongs to the graph `g`.

priority_queue_node(**const** *priority_queue_node &src*)
Constructs an empty `priority_queue_node` that belongs to the same graph `g` as `src`. Any intermediate state of `src`, including its links to predecessors and successors, is not copied.

bool **try_put**(**const** T *&v*)
Adds `v` to the `priority_queue_node` and tries forwarding to a successor the item with the largest priority among all of the items that were added to the node and have not been yet forwarded to successors.

Returns: `true`

bool **try_get**(T *&v*)
Returns: `true` if a message is available in the node and the node is not currently reserved. Otherwise, returns `false`. If the node returns `true`, the message with the largest priority is copied to `v`.

Example

Usage scenario is similar to *sequencer_node* except that the `priority_queue_node` provides local order, passing the message with highest priority of all stored at the moment, while `sequencer_node` enforces global order and does not allow a "smaller priority" message to pass through before all preceding messages.

sequencer_node

[flow_graph.sequencer_node]

A node that forwards messages in a sequence order.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename T >
    class sequencer_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        template< typename Sequencer >
        sequencer_node( graph &g, const Sequencer &s );
        sequencer_node( const sequencer_node &src );

        bool try_put( const T &v );
        bool try_get( output_type &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `T` must meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Sequencer` must meet the *Sequencer requirements* If `Sequencer` instance throws an exception, behavior is undefined.

`sequencer_node` forwards messages in a sequence order to a single successor in its successor set.

`sequencer_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

Each item that passes through a `sequencer_node` is ordered by its sequencer order number. These sequence order numbers range from 0 to the largest integer representable by the `std::size_t` type. Sequencer order number of an item is determined by passing the item to a user-provided `Sequencer` function object.

Note: The `sequencer_node` rejects duplicate sequencer numbers.

Member functions

template<typename **Sequencer**>
sequencer_node(*graph* &*g*, **const** *Sequencer* &*s*)
Constructs an empty `sequencer_node` that belongs to the graph `g` and uses `s` to compute sequence numbers for items.

sequencer_node(**const** *sequencer_node* &*src*)
Constructs an empty `sequencer_node` that belongs to the same graph `g` as `src` and uses a copy of the `Sequencer s` used to construct `src`. The list of predecessors, the list of successors, and the messages inside are not copied.

> **Caution:** The new sequencer object is copy-constructed from a copy of the original sequencer object provided to `src` at its construction. Changes made to member variables in the `src` object do not affect the sequencer of the new `sequencer_node`.

bool **try_put**(**const** T &*v*)
Adds `v` to the `sequencer_node` and tries forwarding the next item in the sequence to a successor.

Returns: `true`

bool **try_get**(T &*v*)
Returns: `true` if the next item in the sequence is available in the `sequencer_node`. If so, it is removed from the node and assigned to `v`. Returns `false` if the next item in sequencer order is not available or if the node is reserved.

Deduction Guides

```
template <typename Body>
sequencer_node(graph&, Body) -> input_node<std::decay_t<input_t<Body>>>;
```

Where:

- `input_t` is an alias to `Body` input argument type.

Example

The example demonstrates ordering capabilities of the `sequencer_node`. While being processed in parallel, the data is passed to the successor node in the exact same order it was read.

```
#include "tbb/flow_graph.h"

struct Message {
    int id;
    int data;
};

int main() {
    tbb::flow::graph g;

    // Due to parallelism the node can push messages to its successors in any order
    tbb::flow::function_node< Message, Message > process(g, tbb::flow::unlimited, []
↪(Message msg) -> Message {
        msg.data++;
        return msg;
    });

    tbb::flow::sequencer_node< Message > ordering(g, [](const Message& msg) -> int {
        return msg.id;
    });

    tbb::flow::function_node< Message > writer(g, tbb::flow::serial, [] (const
↪Message& msg) {
        printf("Message recieved with id: %d\n", msg.id);
    });

    tbb::flow::make_edge(process, ordering);
    tbb::flow::make_edge(ordering, writer);

    for (int i = 0; i < 100; ++i) {
        Message msg = { i, 0 };
        process.try_put(msg);
    }

    g.wait_for_all();
}
```

Service Nodes

These nodes are designed for advanced control of the message flow, such as combining messages from different paths in a graph or limiting the number of simultaneously processed messages, as well as for creating reusable custom nodes.

limiter_node

[flow_graph.limiter_node]

A node that counts and limits the number of messages that pass through it.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename T, typename DecrementType=continue_msg >
    class limiter_node : public graph_node, public receiver<T>, public sender<T> {
    public:
        limiter_node( graph &g, size_t threshold );
        limiter_node( const limiter_node &src );

        receiver<DecrementType>& decrementer();

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- `T` type must meet the *DefaultConstructible* requirements from [defaultconstructible] ISO C++ Standard section.
- The `DecrementType` type must be an integral type or `continue_msg`.

`limiter_node` is a `graph_node`, `receiver<T>`, and `sender<T>`

`limiter_node` has a *discarding* and *broadcast-push* *properties*.

This node does not accept new messages once the user-specified `threshold` is reached. The internal counter of broadcasts is adjusted through use of the *decrementer*, a `receiver` object embedded into the node that can be obtained by calling the `decrementer` method. The counter values are truncated to be inside the [0, `threshold`] interval.

The template parameter `DecrementType` specifies the type of the message that can be sent to the decrementer. This template parameter is defined to `continue_msg` by default. If an integral type is specified, positive values sent to the decrementer determine the value by which the internal counter of broadcasts will be decreased, while negative values determine the value by which the internal counter of broadcasts will be increased.

The `continue_msg` sent to the decrementer decreases the internal counter of broadcasts by one.

When `try_put` call on the decrementer results in the new value of the counter of broadcasts to be less than the `threshold`, the `limiter_node` tries to get a message from one of its known predecessors and forward that message to all its successors. If it cannot obtain a message from a predecessor, it decrements the counter of broadcasts.

Member functions

limiter_node(*graph* &*g*, size_t *threshold*)
Constructs a limiter_node that allows up to threshold items to pass through before rejecting try_put's.

limiter_node(**const** *limiter_node* &*src*)
Constructs a limiter_node that has the same initial state that src had at its construction. The new limiter_node belongs to the same graph g as src, has the same threshold. The list of predecessors, the list of successors, and the current count of broadcasts are not copied from src.

receiver<DecrementType> &**decrementer**()
Obtains a reference to the embedded receiver object that is used for the internal counter adjustments.

bool **try_put**(**const** T &*v*)
If the broadcast count is below the threshold, v is broadcast to all successors.

Returns: true if v is broadcast; false if v is not broadcast because the threshold has been reached.

bool **try_get**(T &*v*)
Returns: false.

broadcast_node

[flow_graph.broadcast_node]

A node that broadcasts incoming messages to all of its successors.

```
// Defined in header <tbb/flow_graph.h>
namespace tbb {
namespace flow {

    template< typename T >
    class broadcast_node :
    public graph_node, public receiver<T>, public sender<T> {
    public:
        explicit broadcast_node( graph &g );
        broadcast_node( const broadcast_node &src );

        bool try_put( const T &v );
        bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
```

broadcast_node is a graph_node, receiver<T>, and sender<T>.

broadcast_node has a *discarding* and *broadcast-push* *properties*.

All messages are forwarded immediately to all successors.

Member functions

explicit broadcast_node (*graph &g*)
Constructs an object of type `broadcast_node` that belongs to the graph `g`.

broadcast_node (**const** *broadcast_node &src*)
Constructs an object of type `broadcast_node` that belongs to the same graph `g` as `src`. The list of predecessors and the list of successors are not copied.

bool **try_put** (**const** input_type *&v*)
Broadcasts `v` to all successors.

Returns: always returns `true`, even if it was unable to successfully forward the message to any of its successors.

bool **try_get** (output_type *&v*)
Returns: `false`.

join_node

[flow_graph.join_node]

A node that creates a tuple from a set of messages received at its input ports and broadcasts the tuple to all of its successors.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {
    using tag_value = /*implementation-specific*/;

    template<typename OutputTuple, class JoinPolicy = /*implementation-defined*/>
    class join_node : public graph_node, public sender< OutputTuple > {
    public:
        using input_ports_type = /*implementation-defined*/;

        explicit join_node( graph &g );
        join_node( const join_node &src );

        input_ports_type &input_ports( );

        bool try_get( OutputTuple &v );
    };

    template<typename OutputTuple, typename K, class KHash=tbb_hash_compare<K> >
    class join_node< OutputTuple, key_matching<K,KHash> > : public graph_node, public␣
↪sender< OutputTuple > {
    public:
        using input_ports_type = /*implementation-defined*/;

        explicit join_node( graph &g );
        join_node( const join_node &src );

        template<typename B0, typename B1>
        join_node( graph &g, B0 b0, B1 b1 );
        template<typename B0, typename B1, typename B2>
        join_node( graph &g, B0 b0, B1 b1, B2 b2 );
```

(continues on next page)

(continued from previous page)

```
        template<typename B0, typename B1, typename B2, typename B3>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3 );
        template<typename B0, typename B1, typename B2, typename B3, typename B4>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6, typename B6>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6, typename B7>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6, typename B7, typename B8>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7,␣
↪B8 b8 );
        template<typename B0, typename B1, typename B2, typename B3, typename B5,␣
↪typename B6, typename B7, typename B8, typename B9>
        join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7,␣
↪B8 b8, B9 b9 );

        input_ports_type &input_ports( );

        bool try_get( OutputTuple &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `OutputTuple` must be an instantiation of `std::tuple`. Each type that the tuple stores must meet the *DefaultConstructible* requirements from [defaultconstructible], *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The `JoinPolicy` type must be specified as one of *buffering policies* for `join_node`.
- The `KHash` type must meet the *HashCompare requirements*.
- The `Bi` types must meet the *JoinNodeFunctionObject requirements*.

A `join_node` is a `graph_node` and a `sender<OutputTuple>`. It contains a tuple of input ports, each of which is a `receiver<Type>` for each *Type* in `OutputTuple`. It supports multiple input receivers with distinct types and broadcasts a tuple of received messages to all of its successors. All input ports of a `join_node` must use the same buffering policy.

The behavior of a `join_node` is based on its buffering policy.

join_node Policies

[flow_graph.join_node_policies]

`join_node` supports three buffering policies at its input ports: `reserving`, `queueing`, and `key_matching`.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    struct reserving;
    struct queueing;
    template<typename K, class KHash=tbb_hash_compare<K> > struct key_matching;
    using tag_matching = key_matching<tag_value>;

} // namespace flow
} // namespace tbb
```

- `queueing` - As each input port is put to, the incoming message is added to an unbounded first-in first-out queue in the port. When there is at least one message at each input port, the `join_node` broadcasts a tuple containing the head of each queue to all successors. If at least one successor accepts the tuple, the head of each input port's queue is removed; otherwise, the messages remain in their respective input port queues.
- `reserving` - As each input port is put to, the `join_node` marks that an input may be available at that port and returns `false`. When all ports have been marked as possibly available, the `join_node` tries to reserve a message at each port from their known predecessors. If it is unable to reserve a message at a port, it unmarks that port, and releases all previously acquired reservations. If it is able to reserve a message at all ports, it broadcasts a tuple containing these messages to all successors. If at least one successor accepts the tuple, the reservations are consumed; otherwise, they are released.
- `key_matching<typename K, class KHash=tbb_hash_compare<K>>` - As each input port is put to, a user-provided function object is applied to the message to obtain its key. The message is then added to a hash table of the input port. When there is a message at each input port for a given key, the `join_node` removes all matching messages from the input ports, constructs a tuple containing the matching messages and attempts to broadcast it to all successors. If no successor accepts the tuple, it is saved and will be forwarded on a subsequent `try_get`.
- `tag_matching` - A specialization of `key_matching` that accepts keys of type `tag_value`.

The function template *input_port* simplifies the syntax for getting a reference to a specific input port.

`join_node` has a *buffering* and *broadcast-push* *properties*.

Member types

`input_ports_type` is an alias to a tuple of input ports.

Member functions

```
explicit join_node( graph &g );
```

Constructs an empty `join_node` that belongs to the graph `g`.

```
template<typename B0, typename B1>
join_node( graph &g, B0 b0, B1 b1 );
template<typename B0, typename B1, typename B2>
join_node( graph &g, B0 b0, B1 b1, B2 b2 );
template<typename B0, typename B1, , typename B2, typename B3>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3 );
template<typename B0, typename B1, , typename B2, typename B3, typename B4>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename␣
↪B6>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename␣
↪B6, typename B6>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename␣
↪B6, typename B7>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename␣
↪B6, typename B7, typename B8>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7, B8 b8 );
template<typename B0, typename B1, , typename B2, typename B3, typename B5, typename␣
↪B6, typename B7, typename B8, typename B9>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7, B8 b8,␣
↪B9 b9 );
```

A constructor only available in the `key_matching` specialization of `join_node`.

Creates a `join_node` that uses the function objects `b0`, `b1`, ... , `bN` to determine the tags for the input ports `0` through `N`.

Caution: Function objects passed to the join_node constructor must not throw. They are called in parallel, and should be pure, take minimal time, and be non-blocking.

```
join_node( const join_node &src )
```

Creates a `join_node` that has the same initial state that `src` had at its construction. The list of predecessors, messages in the input ports, and successors are not copied.

```
input_ports_type &input_ports( )
```

Returns: a `std::tuple` of receivers. Each element inherits values from `receiver<T>`, where `T` is the type of message expected at that input. Each tuple element can be used like any other `receiver<T>`. The behavior of the ports is based on the selected `join_node` policy.

```
bool try_get( output_type &v )
```

Attempts to generate a tuple based on the buffering policy of the `join_node`.

If it can successfully generate a tuple, it copies it to `v` and returns `true`. Otherwise, it returns `false`.

Non-Member Types

```
using tag_value = /*implementation-specific*/;
```

`tag_value` is an unsigned integral type for defining the `tag_matching` policy.

Deduction Guides

```
template <typename Body, typename... Bodies>
join_node(graph&, Body, Bodies...)
    ->join_node<std::tuple<std::decay_t<input_t<Body>>, std::decay_t<input_t<Bodies>>.
↪..>, key_matching<output_t<Body>>>;
```

Where:

- `input_t` is an alias to the input argument type of the passed function object.
- `output_t` is an alias to the return type of the passed function object.

split_node

[flow_graph.split_node]

A `split_node` sends each element of the incoming tuple to the output port that matches the element index in the incoming tuple.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template < typename TupleType >
    class split_node : public graph_node, public receiver<TupleType> {
    public:
        explicit split_node( graph &g );
        split_node( const split_node &other );
        ~split_node();

        bool try_put( const TupleType &v );
```

(continues on next page)

(continued from previous page)

```
        using output_ports_type = /*implementation-defined*/ ;
        output_ports_type& output_ports();
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `TupleType` must be an instantiation of `std::tuple`.

`split_node` is a `receiver<TupleType>` and has a tuple of `sender` output ports. Each of output ports is specified by corresponding tuple element type. This node receives a tuple at its single input port and generates a message from each element of the tuple, passing each to the corresponding output port.

`split_node` has a *discarding* and *broadcast-push* *properties*.

`split_node` has unlimited concurrency, and behaves as a `broadcast_node` with multiple output ports.

Member functions

explicit split_node(*graph &g*)
: Constructs a `split_node` registered with graph `g`.

split_node(**const** *split_node &other*)
: Constructs a `split_node` that has the same initial state that `other` had when it was constructed. The `split_node` that is constructed has a reference to the same `graph` object as `other`. The predecessors and successors of `other` are not copied.

~split_node()
: Destructor

bool **try_put**(**const** TupleType *&v*)
: Broadcasts each element of the incoming tuple to the nodes connected to the `split_node` output ports. The element at index `i` of `v` will be broadcast through the i^{th} output port.

  Returns: `true`

output_ports_type &**output_ports**()
: **Returns**: a tuple of output ports.

indexer_node

[flow_graph.indexer_node]

`indexer_node` broadcasts messages received at input ports to all of its successors. The messages are broadcast individually as they are received at each port. The output is a *tagged message* that contains a tag and a value; the tag identifies the input port on which the message was received.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename T0, typename... TN>
```

(continues on next page)

(continued from previous page)

```
    class indexer_node : public graph_node, public sender</*implementation_defined*/>
↪{
    public:
        indexer_node(graph &g);
        indexer_node(const indexer_node &src);

        using input_ports_type = /*implementation_defined*/;
        input_ports_type &input_ports();

        using output_type = tagged_msg<size_t, T0, TN...>;
        bool try_get( output_type &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The `T0` type and all types in `TN` template parameter pack must meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.

An `indexer_node` is a `graph_node` and `sender<tagged_msg<size_t, T0, TN...>>`. It contains a tuple of input ports, each of which is a `receiver` specified by corresponding input template parameter pack element. It supports multiple input receivers with distinct types and broadcasts each received message to all of its successors. Unlike a `join_node`, each message is broadcast individually to all successors of the `indexer_node` as it arrives at an input port. Before broadcasting, a message is tagged with the index of the port on which the message arrived.

`indexer_node` has a *discarding* and *broadcast-push* *properties*.

The function template *input_port* simplifies the syntax for getting a reference to a specific input port.

Member types

- `input_ports_type` is an alias to a tuple of input ports.
- `output_type` is an alias to the message of type `tagged_msg`, which is sent to successors.

Member functions

`indexer_node`(*graph* &*g*)
: Constructs an `indexer_node` that belongs to the graph `g`.

`indexer_node`(**`const`** *indexer_node* &*src*)
: Constructs an `indexer_node`. The list of predecessors, messages in the input ports, and successors are not copied.

input_ports_type &**`input_ports`**()
: **Returns**: A `std::tuple` of receivers. Each element inherits from `receiver<T>` where `T` is the type of message expected at that input. Each tuple element can be used like any other `receiver<T>`.

bool **`try_get`**(output_type &*v*)
: An `indexer_node` contains no buffering and therefore does not support gets.

  Returns: `false`.

See also:

- *input_port function template*
- *tagged_msg template class*

composite_node

[flow_graph.composite_node]

A node that encapsulates a collection of other nodes as a first class graph node.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename InputTuple, typename OutputTuple > class composite_node;

    // composite_node with both input ports and output ports
    template< typename... InputTypes, typename... OutputTypes>
    class composite_node <std::tuple<InputTypes...>, std::tuple<OutputTypes...> > :␣
↪public graph_node {
    public:
        typedef std::tuple< receiver<InputTypes>&... > input_ports_type;
        typedef std::tuple< sender<OutputTypes>&... > output_ports_type;

        composite_node( graph &g );
        virtual ~composite_node();

        void set_external_ports(input_ports_type&& input_ports_tuple, output_ports_
↪type&& output_ports_tuple);
        input_ports_type& input_ports();
        output_ports_type& output_ports();
    };

    // composite_node with only input ports
    template< typename... InputTypes>
    class composite_node <std::tuple<InputTypes...>, std::tuple<> > : public graph_
↪node{
    public:
        typedef std::tuple< receiver<InputTypes>&... > input_ports_type;

        composite_node( graph &g );
        virtual ~composite_node();

        void set_external_ports(input_ports_type&& input_ports_tuple);
        input_ports_type& input_ports();
    };

    // composite_nodes with only output_ports
    template<typename... OutputTypes>
    class composite_node <std::tuple<>, std::tuple<OutputTypes...> > : public graph_
↪node{
    public:
        typedef std::tuple< sender<OutputTypes>&... > output_ports_type;

        composite_node( graph &g );
        virtual ~composite_node();
```

(continues on next page)

(continued from previous page)

```
        void set_external_ports(output_ports_type&& output_ports_tuple);
        output_ports_type& output_ports();
    };

} // namespace flow
} // namespace tbb
```

- The `InputTuple` and `OutputTuple` must be instantiations of `std::tuple`.

`composite_node` is a `graph_node`, `receiver<T>`, and `sender<T>`.

The `composite_node` can package any number of other nodes. It maintains input and output port references to nodes in the package that border the `composite_node`. This allows the references to be used to make edges to other nodes outside of the `composite_node`. The `InputTuple` is a tuple of input types. The `composite_node` has an input port for each type in `InputTuple`. Likewise, the `OutputTuple` is a tuple of output types. The `composite_node` has an output port for each type in `OutputTuple`.

The composite_node is a multi-port node with three specializations.

- **A multi-port node with multi-input ports and multi-output ports:** This specialization has a tuple of input ports, each of which is a `receiver` of a type in `InputTuple`. Each input port is a reference to a port of a node that the `composite_node` encapsulates. Similarly, this specialization also has a tuple of output ports, each of which is a `sender` of a type in `OutputTuple`. Each output port is a reference to a port of a node that the `composite_node` encapsulates.
- **A multi-port node with only input ports and no output ports:** This specialization only has a tuple of input ports.
- **A multi-port node with only output ports and no input_ports:** This specialization only has a tuple of output ports.

The function template *input_port* can be used to get a reference to a specific input port and the function template *output_port* can be used to get a reference to a specific output port.

Construction of a `composite_node` is done in two stages:

- Defining the `composite_node` with specification of `InputTuple` and `OutputTuple`.
- Making aliases from the encapsulated nodes that border the `composite_node` to the input and output ports of the `composite_node`. This step is mandatory as without it the `composite_node` input and output ports are not bound to any actual nodes. Making the aliases is achieved by calling the method `set_external_ports`.

The composite_node does not meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.

Member functions

composite_node(*graph &g*)
: Constructs a `composite_node` that belongs to the graph `g`.

void **set_external_ports**(input_ports_type &&*input_ports_tuple*, output_ports_type &&*output_ports_tuple*)
: Creates input and output ports of the `composite_node` as aliases to the ports referenced by `input_ports_tuple` and `output_ports_tuple`, respectively. That is, a port referenced at position `N` in `input_ports_tuple` is mapped as the `Nth` input port of the `composite_node`, similarly for output ports.

input_ports_type &**input_ports**()

Returns: A `std::tuple` of `receivers`. Each element is a reference to the actual node or input port that was aliased to that position in `set_external_ports()`.

> **Caution:** Calling `input_ports()` without a prior call to `set_external_ports()` results in undefined behavior.

output_ports_type &**output_ports**()

Returns: A `std::tuple` of `senders`. Each element is a reference to the actual node or output port that was aliased to that position in `set_external_ports()`.

> **Caution:** Calling `output_ports()` without a prior call to `set_external_ports()` results in undefined behavior.

See also:

- *input_port function template*
- *output_port function template*

Ports and Edges

Flow Graph provides an API to manage connections between the nodes. For nodes that have more than one input or output ports (for example, `join_node`), making a connection requires to specify a certain port by using special helper functions.

input_port

[flow_graph.input_port]

A template function that returns a reference to a specific input port of a given `join_node`, `indexer_node` or `composite_node`.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<size_t N, typename NodeType>
    /*implementation-defined*/& input_port(NodeType &n);

} // namespace flow
} // namespace tbb
```

See also:

- *join_node template class*
- *indexer_node template class*
- *composite_node template class*

output_port

[flow_graph.output_port]

A template function that returns a reference to a specific output port of a given `split_node`, `indexer_node`, or `composite_node`.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<size_t N, typename NodeType>
    /*implementation-defined*/& output_port(NodeType &n);

} // namespace flow
} // namespace tbb
```

See also:

- *split_node Template Class*
- *multifunction_node Template Class*
- *composite_node Template Class*

make_edge

[flow_graph.make_edge]

A function template for building edges between nodes.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename Message>
    inline void make_edge( sender<Message> &p, receiver<Message> &s );

    template< typename MultiOutputNode, typename MultiInputNode >
    inline void make_edge( MultiOutputNode& output, MultiInputNode& input );

    template<typename MultiOutputNode, typename Message>
    inline void make_edge( MultiOutputNode& output, receiver<Message> input );

    template<typename Message, typename MultiInputNode>
    inline void make_edge( sender<Message> output, MultiInputNode& input );

} // namespace flow
} // namespace tbb
```

Requirements:

- The *MultiOutputNode* type must have a valid `MultiOutputNode::output_ports_type` qualified-id that denotes a type.
- The *MultiInputNode* type must have a valid `MultiInputNode::input_ports_type` qualified-id that denotes a type.

The common form of `make_edge(sender, receiver)` creates an edge between provided `sender` and `receiver` instances.

Overloads that accept a *MultiOutputNode* type instance make an edge from port `0` of a multi-output predecessor.

Overloads that accept a *MultiInputNode* type instance make an edge to port `0` of a multi-input successor.

remove_edge

[flow_graph.remove_edge]

A function template for building edges between nodes.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename Message>
    inline void remove_edge( sender<Message> &p, receiver<Message> &s );

    template< typename MultiOutputNode, typename MultiInputNode >
    inline void remove_edge( MultiOutputNode& output, MultiInputNode& input );

    template<typename MultiOutputNode, typename Message>
    inline void remove_edge( MultiOutputNode& output, receiver<Message> input );

    template<typename Message, typename MultiInputNode>
    inline void remove_edge( sender<Message> output, MultiInputNode& input );

} // namespace flow
} // namespace tbb
```

Requirements:

- The *MultiOutputNode* type must have a valid `MultiOutputNode::output_ports_type` qualified-id that denotes a type.
- The *MultiInputNode* type must have a valid `MultiInputNode::input_ports_type` qualified-id that denotes a type.

The common form of `remove_edge(sender, receiver)` creates an edge between provided `sender` and `receiver` instances.

Overloads that accept a *MultiOutputNode* type instance remove an edge from port `0` of a multi-output predecessor.

Overloads that accept a *MultiInputNode* type instance remove an edge to port `0` of a multi-input successor.

Special Messages Types

Flow Graph supports a set of specific message types.

continue_msg

[flow_graph.continue_msg]

An empty class that represents a continue message. An object of this class is used to indicate that the sender has completed.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    class continue_msg {};

} // namespace flow
} // namespace tbb
```

tagged_msg

[flow_graph.tagged_msg]

A class template composed of a tag and a message. The message is a value that can be one of several defined types.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename TagType, typename... TN>
    class tagged_msg {
    public:
        template<typename T, typename R>
        tagged_msg(T const &index, R const &val);

        TagType tag() const;

        template<typename V>
        const V& cast_to() const;

        template<typename V>
        bool is_a() const;

    };

} // namespace flow
} // namespace tbb
```

Requirements:

- All types in `TN` template parameter pack must meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.
- The type *TagType* must be an integral unsigned type.

The `tagged_msg` class template is intended for messages whose type is determined at runtime. A message of one of the types `TN` is tagged with a tag of type `TagType`. The tag then can serve to identify the message. In the flow graph, `tagged_msg` is used as the output of *indexer_node*.

Member functions

template<typename **T**, typename **R**>
tagged_msg(*T* **const** &*index*, *R* **const** &*value*)
Requirements:

- The type *R* must be the same as one of the `TN` types.
- The type *T* must be acceptable as a `TagType` constructor parameter.

Constructs a `tagged_msg` with tag `index` and value `val`.

TagType **tag**() **const**
Returns the current tag.

template<typename **V**>
const *V* &**cast_to**() **const**
Requirements:

- The type `V` must be the same as one of the `TN` types.

Returns the value stored in `tagged_msg`. If the value is not of type `V`, the `std::runtime_error` exception is thrown.

template<typename **V**>
bool **is_a**() **const**
Requirements:

- The type `V` must be the same as one of the `TN` types.

Returns true if `V` is the type of the value held by the `tagged_msg`. Returns false, otherwise.

Non-member functions

```
template<typename V, typename T>
const V& cast_to(T const &t) {
    return t.cast_to<V>();
}

template<typename V, typename T>
bool is_a(T const &t);
```

Requirements:

- The type `T` must be an instantiated `tagged_msg` class template.
- The type `V` must be the same as one of the corresponding template arguments for `tagged_msg`.

The free-standing template functions `cast_to` and `is_a` applied to a `tagged_msg` object are equivalent to the calls of the corresponding methods of that object.

See also:

- *indexer_node class template*

Examples

Dependency Flow Graph Example

In the following example, five computations A-E are set up with the partial ordering shown below in "A simple dependency graph.". For each edge in the flow graph, the node at the tail of the edge must complete its execution before the node at the head may begin.

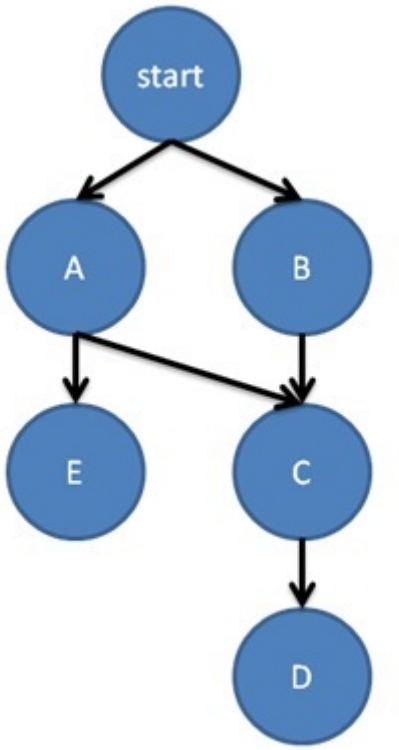


Fig. 2: A simple dependency graph.

```
#include <cstdio>
#include "tbb/flow_graph.h"

using namespace tbb::flow;

struct body {
    std::string my_name;
    body(const char *name) : my_name(name) {}
    void operator()(continue_msg) const {
        printf("%s\n", my_name.c_str());
    }
};

int main() {
    graph g;

    broadcast_node< continue_msg > start(g);
```

(continues on next page)

(continued from previous page)

```
    continue_node<continue_msg> a(g, body("A"));
    continue_node<continue_msg> b(g, body("B"));
    continue_node<continue_msg> c(g, body("C"));
    continue_node<continue_msg> d(g, body("D"));
    continue_node<continue_msg> e(g, body("E"));

    make_edge(start, a);
    make_edge(start, b);
    make_edge(a, c);
    make_edge(b, c);
    make_edge(c, d);
    make_edge(a, e);

    for (int i = 0; i < 3; ++i) {
        start.try_put(continue_msg());
        g.wait_for_all();
    }

    return 0;
}
```

In this example, nodes A-E print out their names. All of these nodes are therefore able to use `struct body` to construct their body objects.

In function `main`, the flow graph is set up once and then run three times. All of the nodes in this example pass around `continue_msg` objects. This type is used to communicate that a node has completed execution.

The first line in function `main` instantiates a `graph` object `g`. On the next line, a `broadcast_node` named `start` is created. Anything passed to this node will be broadcast to all of its successors. The node `start` is used in the `for` loop at the bottom of `main` to launch the execution of the rest of the flow graph.

In the example, five `continue_node` objects are created, named a - e. Each node is constructed with a reference to `graph g` and the function object to invoke when it runs. The successor / predecessor relationships are set up by the `make_edge` calls that follow the declaration of the nodes.

After the nodes and edges are set up, the `try_put` in each iteration of the `for` loop results in a broadcast of a `continue_msg` to both `a` and `b`. Both `a` and `b` are waiting for a single `continue_msg`, since they both have only a single predecessor, `start`.

When they receive the message from `start`, they execute their body objects. When complete, each of them forwards a message to a successor, and so on. The graph uses tasks to execute the node bodies as well as to forward messages between the nodes, allowing computation to execute concurrently when possible.

See also:

- *continue_msg class*
- *continue_node class*

Message Flow Graph Example

This example calculates the sum `x*x + x*x*x` for all `x = 1 to 10`. The layout of this example is shown in the figure below.

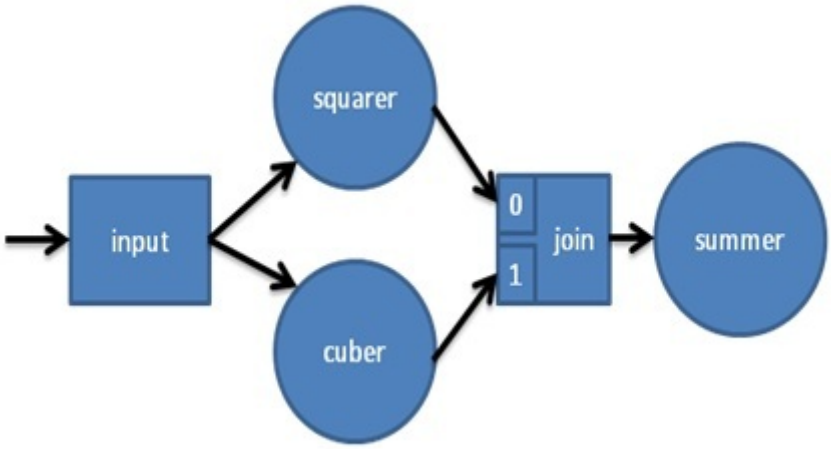


Fig. 3: A simple message flow graph.

Each value enters through the `broadcast_node<int> input`. This node broadcasts the value to both `squarer` and `cuber`, which calculate `x*x` and `x*x*x`, respectively. The output of each of these nodes is put to one of `join`'s ports. A tuple containing both values is created by `join_node< tuple<int,int> > join` and forwarded to `summer`, which adds both values to the running total. Both `squarer` and `cuber` allow unlimited concurrency, that is they each may process multiple values simultaneously. The final `summer`, which updates a shared total, is only allowed to process a single incoming tuple at a time, eliminating the need for a lock around the shared value.

```
#include <cstdio>
#include "tbb/flow_graph.h"

using namespace tbb::flow;

struct square {
  int operator()(int v) { return v*v; }
};

struct cube {
  int operator()(int v) { return v*v*v; }
};

class sum {
  int &my_sum;
public:
  sum( int &s ) : my_sum(s) {}
  int operator()( tuple< int, int > v ) {
    my_sum += get<0>(v) + get<1>(v);
    return my_sum;
  }
};

int main() {
```

(continues on next page)

(continued from previous page)

```
  int result = 0;

  graph g;
  broadcast_node<int> input(g);
  function_node<int,int> squarer( g, unlimited, square() );
  function_node<int,int> cuber( g, unlimited, cube() );
  join_node< tuple<int,int>, queueing > join( g );
  function_node<tuple<int,int>,int>
      summer( g, serial, sum(result) );

  make_edge( input, squarer );
  make_edge( input, cuber );
  make_edge( squarer, get<0>( join.input_ports() ) );
  make_edge( cuber, get<1>( join.input_ports() ) );
  make_edge( join, summer );

  for (int i = 1; i <= 10; ++i)
      input.try_put(i);
  g.wait_for_all();

  printf("Final result is %d\n", result);
  return 0;
}
```

In the example code above, the classes `square`, `cube`, and `sum` define the three user-defined operations. Each class is used to create a `function_node`.

In function `main`, the flow graph is set up and then the values 1-10 are put into the node `input`. All the nodes in this example pass around values of type `int`. The nodes used in this example are all class templates and therefore can be used with any type that supports copy construction, including pointers and objects.

9.2.4 Task Scheduler

[scheduler]

oneAPI Threading Building Blocks (oneTBB) provides a task scheduler, which is the engine that drives the algorithm templates and task groups. The exact tasking API depends on the implementation.

The tasks are quanta of computation. The scheduler implements worker thread pool and maps tasks onto these threads. The mapping is non-preemptive. Once a thread starts running a task, the task is bound to that thread until completion. During that time, the thread services other tasks only when it waits for completion of nested parallel constructs, as described below. While waiting, either user or worker thread may run any available task, including unrelated tasks created by this or other threads.

The task scheduler is intended for parallelizing computationally intensive work. Because task objects are not scheduled preemptively, they should generally avoid making calls that might block a thread for long periods during which the thread cannot service other tasks.

> **Caution:** There is no guarantee that *potentially* parallel tasks *actually* execute in parallel, because the scheduler adjusts actual parallelism to fit available worker threads. For example, given a single worker thread, the scheduler creates no actual parallelism. For example, it is generally unsafe to use tasks in a producer consumer relationship, because there is no guarantee that the consumer runs at all while the producer is running.

Scheduling controls

task_group_context

[scheduler.task_group_context]

`task_group_context` represents a set of properties used by task scheduler for execution of the associated tasks. Each task is associated with only one `task_group_context` object.

The `task_group_context` objects form a forest of trees. Each tree's root is a `task_group_context` constructed as `isolated`.

`task_group_context` is cancelled explicitly by the user request, or implicitly when an exception is thrown out of an associated task. Canceling `task_group_context` causes the entire subtree rooted at it to be cancelled.

The `task_group_context` carries floating point settings inherited from the parent `task_group_context` object or captured with a dedicated interface.

```
// Defined in header <tbb/task_group.h>

namespace tbb {

    class task_group_context {
    public:
        enum kind_t {
            isolated = /* implementation-defined */,
            bound = /* implementation-defined */
        };
        enum traits_type {
            fp_settings = /* implementation-defined */,
            default_traits = 0
        };

        task_group_context( kind_t relation_with_parent = bound,
                            uintptr_t traits = default_traits );
        ~task_group_context();

        void reset();
        bool cancel_group_execution();
        bool is_group_execution_cancelled() const;
        void capture_fp_settings();
        uintptr_t traits() const;
    };

} // namespace tbb;
```

Member types and constants

enum `kind_t`**::isolated**
: When passed to the specific constructor, the created `task_group_context` object has no parent.

enum `kind_t`**::bound**
: When passed to the specific constructor, the created `task_group_context` object becomes a child of the innermost running task's group when the first task associated to the `task_group_context` is passed to the task scheduler. If there is no innermost running task on the current thread, the `task_group_context` becomes `isolated`.

enum `traits_type::fp_settings`
When passed to the specific constructor, the flag forces the context to capture floating-point settings from the current thread.

Member functions

`task_group_context`(kind_t *relation_to_parent* = bound, uintptr_t *traits* = default_traits)
Constructs an empty `task_group_context`.

`~task_group_context`()
Destroys an empty task_group_context. The behavior is undefined if there are still extant tasks associated with this `task_group_context`.

bool **`cancel_group_execution`**()
Requests that tasks associated with this `task_group_context` are not executed.

Returns `false` if this `task_group_context` is already cancelled; `true`, otherwise. If concurrently called by multiple threads, exactly one call returns `true` and the rest return `false`.

bool **`is_group_execution_cancelled`**() **`const`**
Returns `true` if this `task_group_context` has received the cancellation request.

void **`reset`**()
Reinitializes this `task_group_context` to the uncancelled state.

> **Caution:** This method is only safe to call once all tasks associated with the group's subordinate groups have completed. This method must not be invoked concurrently by multiple threads.

void **`capture_fp_settings`**()
Captures floating-point settings from the current thread.

> **Caution:** This method is only safe to call once all tasks associated with the group's subordinate groups have completed. This method must not be invoked concurrently by multiple threads.

uintptr_t **`traits`**() **`const`**
Returns traits of this `task_group_context`.

global_control

[scheduler.global_control]

Use this class to control certain settings or behavior of the oneTBB dynamic library.

An object of class `global_control`, or a "control variable", affects one of several behavioral aspects, or parameters, of TBB. The `global_control` class is primarily intended for use at the application level, to control the whole application behavior.

The current set of parameters that you can modify is defined by the `global_control::parameter` enumeration. The parameter and the value it should take are specified as arguments to the constructor of a control variable. The impact of the control variable ends when its lifetime is complete.

Control variables can be created in different threads, and may have nested or overlapping scopes. However, at any point in time each controlled parameter has a single active value that applies to the whole process. This value is selected from all currently existing control variables by applying a parameter-specific selection rule.

```
// Defined in header <tbb/global_control.h>

namespace tbb {
    class global_control {
    public:
        enum parameter {
            max_allowed_parallelism,
            thread_stack_size,
            terminate_on_exception
        };

        global_control(parameter p, size_t value);
        ~global_control();

        static size_t active_value(parameter param);
    };
} // namespace tbb
```

Member types and constants

enum parameter::**max_allowed_parallelism**
Selection rule: minimum

Limits total number of worker threads that can be active in the task scheduler to `parameter_value - 1`.

Note: With `max_allowed_parallelism` set to `1`, `global_control` enforces serial execution of all tasks by the application thread(s), that is, the task scheduler does not allow worker threads to run. There is one exception: if some work is submitted for execution via `task_arena::enqueue`, a single worker thread will still run ignoring the `max_allowed_parallelism` restriction.

---

enum parameter::**thread_stack_size**
Selection rule: maximum

Set stack size for threads created by the library, including working threads in the task scheduler and threads controlled by thread wrapper classes.

enum parameter::**terminate_on_exception**
Selection rule: logical disjunction

Setting the parameter to 1 causes termination in any condition that would throw or rethrow an exception. If set to 0 (default), the parameter does not affect the implementation behavior.

Member functions

global_control(parameter *param*, size_t *value*)
Constructs a `global_control` object with a specified control parameter and it's value.

~global_control()
Destructs a control variable object and ends it's impact.

static size_t **active_value**(parameter *param*)
Returns the currently active value of the setting defined by `param`.

See also:

- *task_arena*

Resumable tasks

[scheduler.resumable_tasks]

Functions to suspend task execution at a specific point and signal to resume it later.

```
// Defined in header <tbb/task.h>

using tbb::task::suspend_point = /* implementation-defined */;
template < typename Func > void tbb::task::suspend( Func );
void tbb::task::resume( tbb::task::suspend_point );
```

Requirements:

- The `Func` type must meet the *SuspendFunc requirements*.

The `tbb::task::suspend` function called within a running task suspends execution of the task and switches the thread to participate in other oneTBB parallel work. This function accepts a user callable object with the current execution context `tbb::task::suspend_point` as an argument.

The `tbb::task::suspend_point` context tag must be passed to the `tbb::task::resume` function to trigger a program execution at the suspended point. The `tbb::task::resume` function can be called at any point of an application, even on a separate thread. In this regard, this function acts as a signal for the task scheduler.

Note: There are no guarantees that the same thread that called `tbb::task::suspend` continues execution after the suspended point. However, these guarantees are provided for the outermost blocking oneTBB calls (such as `tbb::parallel_for` and `tbb::flow::graph::wait_for_all`) and `tbb::task_arena::execute` calls.

Example

```
// Parallel computation region
tbb::parallel_for(0, N, [&](int) {
    // Suspend the current task execution and capture the context
    tbb::task::suspend([&] (tbb::task::suspend_point tag) {
        // Dedicated user-managed activity that processes async requests.
        async_activity.submit(tag); // could be OpenCL/IO/Database/Network etc.
    }); // execution will be resumed after this function
});
```

```
// Dedicated user-managed activity:

// Signal to resume execution of the task referenced by the tbb::task::suspend_point
// from a dedicated user-managed activity
tbb::task::resume(tag);
```

Task Group

task_group

[scheduler.task_group]

A `task_group` represents concurrent execution of a group of tasks. Tasks can be dynamically added to the group while it is executing.

```
// Defined in header <tbb/task_group.h>

namespace tbb {

    class task_group {
    public:
        task_group();
        ~task_group();

        template<typename Func>
        void run( Func&& f );

        template<typename Func>
        task_group_status run_and_wait( const Func& f );

        task_group_status wait();
        void cancel();
    };

    bool is_current_task_group_canceling();

} // namespace tbb
```

Member functions

task_group()
: Constructs an empty `task_group`.

~task_group()
: Destroys the `task_group`.

  Requires: Method `wait` must be called before destroying a `task_group`, otherwise, the destructor throws an exception.

template<typename **Func**>
void **run**(*Func* &&*f*)
: Adds a task to compute `f()` and returns immediately. The `Func` type must meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section.

template<typename **Func**>
task_group_status **run_and_wait**(**const** *Func* &*f*)
: Equivalent to `{run(f); return wait();}`, but guarantees that `f()` runs on the current thread. The `Func` type must meet the *Function Objects* requirements from the [function.objects] ISO C++ Standard section.

  Returns: The status of `task_group`. See *task_group_status*.

task_group_status **wait**()
: Waits for all tasks in the group to complete or be cancelled.

Returns: The status of `task_group`. See *task_group_status*.

void **cancel**()
Cancels all tasks in this `task_group`.

Non-member functions

bool **is_current_task_group_canceling**()
Returns true if an innermost `task_group` executing on this thread is cancelling its tasks.

task_group_status

[scheduler.task_group_status]

A `task_group_status` type represents the status of a `task_group`.

```
namespace tbb {
    enum task_group_status {
        not_complete,
        complete,
        canceled
    };
}
```

Member constants

not_complete
Not cancelled and not all tasks in a group have completed.

complete
Not cancelled and all tasks in a group have completed.

canceled
Task group received cancellation request.

Task Arena

task_arena

[scheduler.task_arena]

A class that represents an explicit, user-managed task scheduler arena.

```
// Defined in header <tbb/task_arena.h>

namespace tbb {

    class task_arena {
    public:
        static const int automatic = /* unspecified */;
        static const int not_initialized = /* unspecified */;
        enum class priority : /* unspecified type */ {
            low = /* unspecified */,
```

(continues on next page)

(continued from previous page)

```
            normal = /* unspecified */,
            high = /* unspecified */
        };
        struct attach {};
        struct constraints {
            numa_node_id numa_node;
            int max_concurrency;

            constraints(numa_node_id numa_node_       = task_arena::automatic,
                        int          max_concurrency_ = task_arena::automatic);
        };

        task_arena(int max_concurrency = automatic, unsigned reserved_for_masters = 1,
                   priority a_priority = priority::normal);
        task_arena(constraints a_constraints, unsigned reserved_for_masters = 1,
                   priority a_priority = priority::normal);
        task_arena(const task_arena &s);
        explicit task_arena(task_arena::attach);
        ~task_arena();

        void initialize();
        void initialize(int max_concurrency, unsigned reserved_for_masters = 1,
                        priority a_priority = priority::normal);
        void initialize(constraints a_constraints, unsigned reserved_for_masters = 1,
                        priority a_priority = priority::normal);
        void initialize(task_arena::attach);
        void terminate();

        bool is_active() const;
        int max_concurrency() const;

        template<typename F> auto execute(F&& f) -> decltype(f());
        template<typename F> void enqueue(F&& f);
    };

} // namespace tbb
```

A `task_arena` class represents a place where threads may share and execute tasks.

The number of threads that may simultaneously execute tasks in a `task_arena` is limited by its concurrency level.

Each user thread that invokes any parallel construction outside an explicit `task_arena` uses an implicit task arena representation object associated with the calling thread.

The tasks spawned or enqueued into one arena cannot be executed in another arena.

Each `task_arena` has a `priority`. The tasks from `task_arena` with higher priority are given a precedence in execution over the tasks from `task_arena` with lower priority.

Note: The `task_arena` constructors do not create an internal task arena representation object. It may already exist in case of the "attaching" constructor; otherwise, it is created by an explicit call to `task_arena::initialize` or lazily on first use.

Member types and constants

static const int **automatic**
When passed as `max_concurrency` to the specific constructor, arena concurrency is automatically set based on the hardware configuration.

static const int **not_initialized**
When returned by a method or function, indicates that there is no active `task_arena` or that the `task_arena` object has not yet been initialized.

enum `priority::` **low**
When passed to a constructor or the `initialize` method, the initialized `task_arena` has a lowered priority.

enum `priority::` **normal**
When passed to a constructor or the `initialize` method, the initialized `task_arena` has regular priority.

enum `priority::` **high**
When passed to a constructor or the `initialize` method, the initialized `task_arena` has a raised priority.

struct attach
A tag for constructing a `task_arena` with attach.

struct constraints
Represents limitations applied to threads within `task_arena`.

`numa_node` - An integral logical index uniquely identifying a NUMA node. All threads joining the `task_arena` are bound to this NUMA node.

Note: NUMA node ID is considered valid if it was obtained through tbb::info::numa_nodes().

`max_concurrency` - The maximum number of threads that can participate in work processing within the `task_arena` at the same time.

Member functions

task_arena (int *max_concurrency* = *automatic*, unsigned *reserved_for_masters* = 1, priority *a_priority* = priority::*normal*)
Creates a `task_arena` with a certain concurrency limit (`max_concurrency`) and priority (`a_priority`). Some portion of the limit can be reserved for application threads with `reserved_for_masters`. The amount for reservation cannot exceed the limit.

> **Caution:** If `max_concurrency` and `reserved_for_masters` are explicitly set to be equal and greater than 1, oneTBB worker threads will never join the arena. As a result, the execution guarantee for enqueued tasks is not valid in such arena. Do not use `task_arena::enqueue()` with an arena set to have no worker threads.

task_arena (*constraints a_constraints*, unsigned *reserved_for_masters* = 1, priority *a_priority* = priority::*normal*)
Creates a `task_arena` with a certain constraints(`a_constraints`) and priority (`a_priority`). Some portion of the limit can be reserved for application threads with `reserved_for_masters`. The amount for reservation cannot exceed the concurrency limit specified in `constraints`.

> **Caution:** If `constraints::max_concurrency` and `reserved_for_masters` are explicitly set to be equal and greater than 1, oneTBB worker threads will never join the arena. As a result, the execution guarantee for enqueued tasks is not valid in such arena. Do not use `task_arena::enqueue()` with an arena set to have no worker threads.

If `constraints::numa_node` is specified, then all threads that enter the arena are automatically pinned to corresponding NUMA node.

`task_arena`(**`const`** *task_arena*&)
Copies settings from another `task_arena` instance.

`explicit task_arena`(*task_arena*::attach)
Creates an instance of `task_arena` that is connected to the internal task arena representation currently used by the calling thread. If no such arena exists yet, creates a `task_arena` with default parameters.

Note: Unlike other constructors, this one automatically initializes the new `task_arena` when connecting to an already existing arena.

`~task_arena`()
Destroys the `task_arena` instance, but the destruction may not be synchronized with any task execution inside this `task_arena`. It means that an internal task arena representation associated with this `task_arena` instance can be destroyed later. Not thread-safe for concurrent invocations of other methods.

void **`initialize`**()
Performs actual initialization of internal task arena representation.

Note: After the call to `initialize`, the arena parameters are fixed and cannot be changed.

---

void **`initialize`**(int *max_concurrency*, unsigned *reserved_for_masters* = 1, priority *a_priority* = priority::*normal*)
Same as above, but overrides previous arena parameters.

void **`initialize`**(*constraints a_constraints*, unsigned *reserved_for_masters* = 1, priority *a_priority* = priority::*normal*)
Same as above.

void **`initialize`**(*task_arena*::attach)
If an instance of class `task_arena::attach` is specified as the argument, and there is an internal task arena representation currently used by the calling thread, the method ignores arena parameters and connects `task_arena` to that internal task arena representation. The method has no effect when called for an already initialized `task_arena`.

void **`terminate`**()
Removes the reference to the internal task arena representation without destroying the task_arena object, which can then be re-used. Not thread safe for concurrent invocations of other methods.

bool **`is_active`**() **`const`**
Returns `true` if the `task_arena` has been initialized; `false`, otherwise.

int **`max_concurrency`**() **`const`**
Returns the concurrency level of the `task_arena`. Does not require the `task_arena` to be initialized and does not perform initialization.

template<F>
void **`enqueue`**(F &&*f*)
Enqueues a task into the `task_arena` to process the specified functor and immediately returns. The `F` type

must meet the *Function Objects* requirements from the [function.objects] ISO C++ Standard section. The task is scheduled for eventual execution by a worker thread even if no thread ever explicitly waits for the task to complete. If the total number of worker threads is zero, a special additional worker thread is created to execute enqueued tasks.

Note: The method does not require the calling thread to join the arena; that is, any number of threads outside of the arena can submit work to it without blocking.

Caution: There is no guarantee that tasks enqueued into an arena execute concurrently with respect to any other tasks there.

Caution: An exception thrown and not caught in the functor results in undefined behavior.

template<F>
auto **execute** (F &&*f*) -> decltype(*f*())

Executes the specified functor in the `task_arena` and returns the value returned by the functor. The `F` type must meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section.

The calling thread joins the `task_arena` if possible, and executes the functor. Upon return it restores the previous task scheduler state and floating-point settings.

If joining the `task_arena` is not possible, the call wraps the functor into a task, enqueues it into the arena, waits using an OS kernel synchronization object for another opportunity to join, and finishes after the task completion.

An exception thrown in the functor will be captured and re-thrown from `execute`.

Note: Any number of threads outside of the arena can submit work to the arena and be blocked. However, only the maximal number of threads specified for the arena can participate in executing the work.

Example

The example demonstrates `task_arena` NUMA support API. Each constructed `task_arena` is pinned to the corresponding NUMA node.

```
#include "tbb/task_group.h"
#include "tbb/task_arena.h"

#include <vector>

int main() {
    std::vector<tbb::numa_node_id> numa_nodes = tbb::info::numa_nodes();
    std::vector<tbb::task_arena> arenas(numa_nodes.size());
    std::vector<tbb::task_group> task_groups(numa_nodes.size());

    for (int i = 0; i < numa_nodes.size(); i++) {
        arenas[i].initialize(tbb::task_arena::constraints(numa_nodes[i]));
    }
```

(continues on next page)

(continued from previous page)

```
    for (int i = 0; i < numa_nodes.size(); i++) {
        arenas[i].execute([&task_groups, i] {
            task_groups[i].run([] {
                /* executed by the thread pinned to specified NUMA node */
            });
        });
    }

    for (int i = 0; i < numa_nodes.size(); i++) {
        arenas[i].execute([&task_groups, i] {
            task_groups[i].wait();
        });
    }

    return 0;
}
```

See also:

- *task_group*
- *task_scheduler_observer*

this_task_arena

[scheduler.this_task_arena]

The namespace for functions applicable to the current `task_arena`.

The namespace `this_task_arena` contains global functions for interaction with the `task_arena` currently used by the calling thread.

```
// Defined in header <tbb/task_arena.h>

namespace tbb {
    namespace this_task_arena {
        int current_thread_index();
        int max_concurrency();
        template<typename F> auto isolate(F&& f) -> decltype(f());
    }
}
```

int **`current_thread_index`**`()`

Returns the thread index in a `task_arena` currently used by the calling thread, or `task_arena::not_initialized` if the thread has not yet initialized the task scheduler.

A thread index is an integer number between 0 and the `task_arena` concurrency level. Thread indexes are assigned to both application threads and worker threads on joining an arena and are kept until exiting the arena. Indexes of threads that share an arena are unique, that is, no two threads within the arena can have the same index at the same time - but not necessarily consecutive.

Note: Since a thread may exit the arena at any time if it does not execute a task, the index of a thread may change between any two tasks, even those belonging to the same task group or algorithm.

Note: Threads that use different arenas may have the same current index value.

Note: Joining a nested arena in `execute()` may change current index value while preserving the index in the outer arena which will be restored on return.

int **max_concurrency**()
Returns the concurrency level of the `task_arena` currently used by the calling thread. If the thread has not yet initialized the task scheduler, returns the concurrency level determined automatically for the hardware configuration.

template<F>
auto **isolate**(F &&*f*) -> decltype(*f*())
Runs the specified functor in isolation by restricting the calling thread to process only tasks scheduled in the scope of the functor (also called the isolation region). The function returns the value returned by the functor. The `F` type must meet the *Function Objects* requirements from the [function.objects] ISO C++ Standard section.

Caution: The object returned by the functor cannot be a reference. `std::reference_wrapper` can be used instead.

task_scheduler_observer

[scheduler.task_scheduler_observer]

Class that represents thread interest in task scheduling services.

```
// Defined in header <tbb/task_scheduler_observer.h>

namespace tbb {

   class task_scheduler_observer {
   public:
       task_scheduler_observer();
       explicit task_scheduler_observer( task_arena& a );
       virtual ~task_scheduler_observer();

       void observe( bool state=true );
       bool is_observing() const;

       virtual void on_scheduler_entry( bool is_worker ) {}
       virtual void on_scheduler_exit( bool is_worker ) {}
   };

}
```

A `task_scheduler_observer` permits clients to observe when a thread starts and stops processing tasks, either globally or in a certain task scheduler arena. You typically derive your own observer class from `task_scheduler_observer`, and override virtual methods `on_scheduler_entry` or `on_scheduler_exit`. Observation can be enabled and disabled for an observer instance; it is disabled on creation. Remember to call `observe()` to enable observation.

Exceptions thrown and not caught in the overridden methods of `task_scheduler_observer` result in undefined behavior.

Member functions

`task_scheduler_observer``()`
Constructs a `task_scheduler_observer` object in the inactive state (observation is disabled). For a created observer, entry/exit notifications are invoked whenever a worker thread joins/leaves the arena of the observer's owner thread. If a thread is already in the arena when the observer is activated, the entry notification is called before it executes the first stolen task.

`explicit task_scheduler_observer`(*task_arena*`&`)
Constructs a `task_scheduler_observer` object for a given arena in inactive state (observation is disabled). For created observer, entry/exit notifications are invoked whenever a thread joins/leaves arena. If a thread is already in the arena when the observer is activated, the entry notification is called before it executes the first stolen task.

Constructs a `task_scheduler_observer` object in the inactive state (observation is disabled), which receives notifications from threads entering and exiting the specified `task_arena`.

`~task_scheduler_observer``()`
Disables observing and destroys the observer instance. Waits for extant invocations of `on_scheduler_entry` and `on_scheduler_exit` to complete.

void **`observe`**(bool *state* = true)
Enables observing if `state` is true; disables observing if `state` is false.

bool **`is_observing`**`() const`
Returns: True if observing is enabled; false, otherwise.

`virtual` void **`on_scheduler_entry`**(bool *is_worker*)
The task scheduler invokes this method for each thread that starts participating in oneTBB work or enters an arena after the observation is enabled. For threads that already execute tasks, the method is invoked before executing the first task stolen after enabling the observation.

If a thread enables the observation and then spawns a task, it is guaranteed that the task, as well as all the tasks it creates, will be executed by threads which have invoked `on_scheduler_entry`.

The flag `is_worker` is true if the thread was created by oneTBB; false, otherwise.

Effects: The default behavior does nothing.

`virtual` void **`on_scheduler_exit`**(bool *is_worker*)
The task scheduler invokes this method when a thread stops participating in task processing or leaves an arena.

> **Caution:** A process does not wait for the worker threads to clean up, and can terminate before `on_scheduler_exit` is invoked.

Effects: The default behavior does nothing.

Example

The following example sketches the code of an observer that pins oneTBB worker threads to hardware threads.

```
class pinning_observer : public tbb::task_scheduler_observer {
public:
    affinity_mask_t m_mask; // HW affinity mask to be used for threads in an arena
    pinning_observer( tbb::task_arena &a, affinity_mask_t mask )
        : tbb::task_scheduler_observer(a), m_mask(mask) {
        observe(true); // activate the observer
    }
    void on_scheduler_entry( bool worker ) override {
        set_thread_affinity(tbb::this_task_arena::current_thread_index(), m_mask);
    }
    void on_scheduler_exit( bool worker ) override {
        restore_thread_affinity();
    }
};
```

9.2.5 Containers

[containers]

The container classes provided by oneAPI Threading Building Blocks (oneTBB) permit multiple threads to simultaneously invoke certain methods on the same container.

Sequences

concurrent_vector

[containers.concurrent_vector]

`concurrent_vector` is a class template for a vector that can be concurrently grown and accessed.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_vector.h>

namespace tbb {

    template <typename T,
              typename Allocator = cache_aligned_allocator<T>>
    class concurrent_vector {
        using value_type = T;
        using allocator_type = Allocator;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using reference = value_type&;
        using const_reference = const value_type&;

        using pointer = typename std::allocator_traits<allocator_type>::pointer;
```

(continues on next page)

(continued from previous page)

```
        using const_pointer = typename std::allocator_traits<allocator_type>::const_
↪pointer;

        using iterator = <implementation-defined RandomAccessIterator>;
        using const_iterator = <implementation-defined constant RandomAccessIterator>;

        using reverse_iterator = std::reverse_iterator<iterator>;
        using const_reverse_iterator = std::reverse_iterator<const_iterator>;

        using range_type = <implementation-defined ContainerRange>;
        using const_range_type = <implementation-defined constant ContainerRange>;

        // Construction, destruction, copying
        concurrent_vector();
        explicit concurrent_vector( const allocator_type& alloc ) noexcept;

        explicit concurrent_vector( size_type count, const value_type& value,
                                    const allocator_type& alloc = allocator_type() );

        explicit concurrent_vector( size_type count,
                                    const allocator_type& alloc = allocator_type() );

        template <typename InputIterator>
        concurrent_vector( InputIterator first, InputIterator last,
                           const allocator_type& alloc = allocator_type() );

        concurrent_vector( std::initializer_list<value_type> init,
                           const allocator_type& alloc = allocator_type() );

        concurrent_vector( const concurrent_vector& other );
        concurrent_vector( const concurrent_vector& other, const allocator_type&␣
↪alloc );

        concurrent_vector( concurrent_vector&& other ) noexcept;
        concurrent_vector( concurrent_vector&& other, const allocator_type& alloc );

        ~concurrent_vector();

        concurrent_vector& operator=( const concurrent_vector& other );

        concurrent_vector& operator=( concurrent_vector&& other ) noexcept(/*See␣
↪details*/);

        concurrent_vector& operator=( std::initializer_list<value_type> init );

        void assign( size_type count, const value_type& value );

        template <typename InputIterator>
        void assign( InputIterator first, InputIterator last );

        void assign( std::initializer_list<value_type> init );

        // Concurrent growth
        iterator grow_by( size_type delta );
        iterator grow_by( size_type delta, const value_type& value );

        template <typename InputIterator>
```

(continues on next page)

(continued from previous page)

```
        iterator grow_by( InputIterator first, InputIterator last );

        iterator grow_by( std::initializer_list<value_type> init );

        iterator grow_to_at_least( size_type n );
        iterator grow_to_at_least( size_type n, const value_type& value );

        iterator push_back( const value_type& value );
        iterator push_back( value_type&& value );

        template <typename... Args>
        iterator emplace_back( Args&&... args );

        // Element access
        value_type& operator[]( size_type index );
        const value_type& operator[]( size_type index ) const;

        value_type& at( size_type index );
        const value_type& at( size_type index ) const;

        value_type& front();
        const value_type& front() const;

        value_type& back();
        const value_type& back() const;

        // Iterators
        iterator begin();
        const_iterator begin() const;
        const_iterator cbegin() const;

        iterator end();
        const_iterator end() const;
        const_iterator cend() const;

        reverse_iterator rbegin();
        const_reverse_iterator rbegin() const;
        const_reverse_iterator crbegin() const;

        reverse_iterator rend();
        const_reverse_iterator rend() const;
        const_reverse_iterator crend() const;

        // Size and capacity
        size_type size() const noexcept;

        bool empty() const noexcept;

        size_type max_size() const noexcept;

        size_type capacity() const noexcept;

        // Concurrently unsafe operations
        void reserve( size_type n );

        void resize( size_type n );
        void resize( size_type n, const value_type& value );
```

(continues on next page)

(continued from previous page)

```
        void shrink_to_fit();

        void swap( concurrent_vector& other ) noexcept(/*See details*/);

        void clear();

        allocator_type get_allocator() const;

        // Parallel iteration
        range_type range( size_type grainsize = 1 );
        const_range_type range( size_type grainsize = 1 ) const;
    }; // class concurrent_vector

} // namespace tbb
```

Requirements

- The type `T` must meet the following requirements:
  - Requirements of `Erasable` from the [container.requirements] ISO C++ Standard section.
  - Its destructor must not throw an exception.
  - If its default constructor can throw an exception, the destructor must be non-virtual and work correctly on zero-filled memory.
  - Member functions can impose stricter requirements depending on the type of the operation.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ section.

Description

`tbb::concurrent_vector` is a class template that represents a sequence container with the following features:

- Multiple threads can concurrently grow the container and append new elements.
- Random access by index. The index of the first element is zero.
- Growing the container does not invalidate any existing iterators or indices.

Exception Safety

Concurrent growing is fundamentally incompatible with ideal exception safety. Nonetheless, `tbb::concurrent_vector` offers a practical level of exception safety.

Growth and vector assignment append a sequence of elements to a vector. If an exception occurs, the impact on the vector depends on the cause of the exception:

- If the exception is thrown by the constructor of an element, all subsequent elements in the appended sequence will be zero-filled.
- Otherwise, the exception is thrown by the vector allocator. The vector becomes broken. Each element in the appended sequence will be in one of three states:

  - constructed
  - zero-filled
  - unallocated in memory

Once a vector becomes broken, note the following when accessing it:

- Accessing an unallocated element with the method `at` causes an exception `std::range_error`. Accessing an unallocated element using any other method has undefined behavior.
- The values of `capacity()` and `size()` may be less than expected.
- Access to a broken vector via `back()` has undefined behavior.

However, the following guarantees hold for broken or unbroken vectors:

- Let `k` be an index of an unallocated element. Then `size() <= capacity() <= k`.
- Growth operations never cause `size()` or `capacity()` to decrease.

If a concurrent growth operation successfully completes, the appended sequence remains valid and accessible even if a subsequent growth operations fails.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_vector();

explicit concurrent_vector( const allocator_type& alloc );
```

Constructs an empty `concurrent_vector`.

If provided, uses the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
explicit concurrent_vector( size_type count, const value_type& value,
                            const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_vector` containing `count` copies of the `value` using the allocator `alloc`.

```
explicit concurrent_vector( size_type count,
                            const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_vector` containing `n` default constructed in-place elements using the allocator `alloc`.

```
template <typename InputIterator>
concurrent_vector( InputIterator first, InputIterator last,
                   const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_vector` contains all elements from the half-open interval `[first, last)` using the allocator `alloc`.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_vector( std::initializer_list<value_type> init,
                   const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_vector(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_vector( const concurrent_vector& other );

concurrent_vector( const concurrent_vector& other,
                   const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_vector( concurrent_vector&& other );

concurrent_vector( concurrent_vector&& other,
                   const allocator_type& alloc );
```

Constructs a `concurrent_vector` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_vector();
```

Destroys the `concurrent_vector`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_vector& operator=( const concurrent_vector& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_vector& operator=( concurrent_vector&& other ) noexcept(/*See␣
↪below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::propagate_on_
↪container_move_assignment::value ||
         std::allocator_traits<allocator_type>::is_always_
↪equal::value)
```

```
concurrent_vector& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

assign

```
void assign( size_type count, const value_type& value );
```

Replaces all elements in `*this` by `count` copies of `value`.

```
template <typename InputIterator>
void assign( InputIterator first, InputIterator last );
```

Replaces all elements in `*this` by the elements from the half-open interval `[first, last)`.

This overload only participates in overload resolution if the type `InputIterator` meets the requirements of *InputIterator* from the [input.iterators] ISO C++ Standard section.

```
void assign( std::initializer_list<value_type> init );
```

Equivalent to `assign(init.begin(), init.end())`.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

Concurrent growth

All member functions in this section can be performed concurrently with each other, element access methods and while traversing the container.

grow_by

```
iterator grow_by( size_type delta );
```

Appends a sequence comprising `delta` new default-constructed in-place elements to the end of the vector.

Returns: iterator to the beginning of the appended sequence.

Requirements: the type `value_type` must meet the `DefaultConstructible` and `EmplaceConstructible` requirements from [defaultconstructible] and [container.requirements] ISO C++ sections.

```
iterator grow_by( size_type delta, const value_type& value );
```

Appends a sequence comprising `delta` copies of `value` to the end of the vector.

Returns: iterator to the beginning of the appended sequence.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename InputIterator>
iterator grow_by( InputIterator first, InputIterator last );
```

Appends a sequence comprising all elements from the half-open interval `[first, last)` to the end of the vector.

Returns: iterator to the beginning of the appended sequence.

This overload participates in overload resolution only if the type `InputIterator` meets the requirements of *InputIterator* from the [input.iterators] ISO C++ Standard section.

```
iterator grow_by( std::initializer_list<value_type> init );
```

Equivalent to `grow_by(init.begin(), init.end())`.

grow_to_at_least

```
iterator grow_to_at_least( size_type n );
```

Appends minimal sequence of default constructed in-place elements such that `size() >= n`.

Returns: iterator to the beginning of the appended sequence.

Requirements: the type `value_type` must meet the `DefaultConstructible` and `EmplaceConstructible` requirements from [defaultconstructible] and [container.requirements] ISO C++ sections.

```
iterator grow_to_at_least( size_type n, const value_type& value );
```

Appends minimal sequence of comprising copies of `value` such that `size() >= n`.

Returns: iterator to the beginning of the appended sequence.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

push_back

```
iterator push_back( const value_type& value );
```

Appends a copy of `value` to the end of the vector.

Returns: iterator to the appended element.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator push_back( value_type&& value );
```

Appends `value` to the end of the vector using move semantics.

`value` is left in a valid, but unspecified state.

Returns: iterator to the appended element.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

emplace_back

```
template <typename... Args>
iterator emplace_back( Args&&... args );
```

Appends an element constructed in-place from `args` to the end of the vector.

Returns: iterator to the appended element.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Element access

All member functions in this section can be performed concurrently with each other, concurrent growth methods and while traversing the container.

In case of concurrent growth, the element returned by the access method can refer to the element that is under construction of the other thread.

Access by index

```
value_type& operator[]( size_type index );

const value_type& operator[]( size_type index ) const;
```

Returns: a reference to the element on the position `index`.

The behavior is undefined if `index() >= size()`.

```
value_type& at( size_type index );

const value_type& at( size_type index ) const;
```

Returns: a reference to the element on the position `index`.

Throws:

- `std::out_of_range` if `index >= size()`.
- `std::range_error` if the vector is broken and the element on the position `index` unallocated.

Access the first and the last element

```
value_type& front();

const value_type& front() const;
```

Returns: a reference to the first element in the vector.

```
value_type& back();

const value_type& back() const;
```

Returns: a reference to the last element in the vector.

Iterators

The types `concurrent_vector::iterator` and `concurrent_vector::const_iterator` meet the requirements of `RandomAccessIterator` from the [random.access.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the vector.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the vector.

rbegin and crbegin

```
reverse_iterator rbegin();

const_reverse_iterator rbegin() const;

const_reverse_iterator crbegin() const;
```

Returns: a reverse iterator to the first element of the reversed vector.

rend and crend

```
reverse_iterator rend();

const_reverse_iterator rend() const;

const_reverse_iterator crend() const;
```

Returns: a reverse iterator that follows the last element of the reversed vector.

Size and capacity

size

```
size_type size() const noexcept;
```

Returns: the number of elements in the vector.

empty

```
bool empty() const noexcept;
```

Returns: `true` if the vector is empty; `false`, otherwise.

max_size

```
size_type max_size() const noexcept;
```

Returns: the maximum number of elements that the vector can hold.

capacity

```
size_type capacity() const noexcept;
```

Returns: the maximum number of elements that the vector can hold without allocating more memory.

Concurrently unsafe operations

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Reserving

```
void reserve( size_type n );
```

Reserves memory for at least `n` elements.

Throws: `std::length_error` if `n > max_size()`.

Resizing

```
void resize( size_type n );
```

If `n < size()`, the vector is reduced to its first `n` elements.

Otherwise, appends `n - size()` new elements default-constructed in-place to the end of the vector.

```
void resize( size_type n, const value_type& value );
```

If `n < size()`, the vector is reduced to its first `n` elements.

Otherwise, appends `n - size()` copies of `value` to the end of the vector.

shrink_to_fit

```
void shrink_to_fit();
```

Removes the unused capacity of the vector.

Call for this method can also reorganize the internal vector representation in the memory.

clear

```
void clear();
```

Removes all elements from the container.

swap

```
void swap( concurrent_vector& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::propagate_on_
↪container_swap::value ||
        std::allocator_traits<allocator_type>::is_always_
↪equal::value
```

Parallel iteration

Member types `concurrent_vector::range_type` and `concurrent_vector::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_vector::const_range_type` are of type `concurrent_vector::const_iterator`, whereas the bounds for a `concurrent_vector::range_type` are of type `concurrent_vector::iterator`.

range member function

```
range_type range( size_type grainsize = 1 );

const_range_type range( size_type grainsize = 1 ) const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary and lexicographical comparison and swap operations on `tbb::concurrent_vector` objects.

The exact namespace where these functions are defined is unspecified, as long as they can be used in respective comparison operations. For example, an implementation can define the classes and functions in the same internal namespace and define `tbb::concurrent_vector` as a type alias, for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Allocator>
bool operator==( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator!=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator<( const concurrent_vector<T, Allocator>& lhs,
                const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator<=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator>( const concurrent_vector<T, Allocator>& lhs,
                const concurrent_vector<T, Allocator>& rhs );
```

(continues on next page)

(continued from previous page)

```
template <typename T, typename Allocator>
bool operator>=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
void swap( concurrent_vector<T, Allocator>& lhs,
           concurrent_vector<T, Allocator>& rhs );
```

Non-member binary comparisons

Two objects of `concurrent_vector` are equal if:

- they contains an equal number of elements.
- the elements on the same positions are equal.

```
template <typename T, typename Allocator>
bool operator==( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is equal to `rhs`, `false` otherwise.

```
template <typename T, typename Allocator>
bool operator!=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is not equal to `rhs`, `false` otherwise.

Non-member lexicographical comparisons

```
template <typename T, typename Allocator>
bool operator<( const concurrent_vector<T, Allocator>& lhs,
                const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`; `false`, otherwise.

```
template <typename T, typename Allocator>
bool operator<=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`; `false`, otherwise.

```
template <typename T, typename Allocator>
bool operator>( const concurrent_vector<T, Allocator>& lhs,
                const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`; `false`, otherwise.

```
template <typename T, typename Allocator>
bool operator>=( const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`; `false`, otherwise.

Non-member swap

```
template <typename T, typename Allocator>
void swap( concurrent_vector<T, Allocator>& lhs,
           concurrent_vector<T, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Other

Deduction guides

Where possible, constructors of `concurrent_vector` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Allocator = cache_aligned_allocator<iterator_value_t<InputIterator>
↪>>
concurrent_vector( InputIterator, InputIterator,
                   const Allocator& = Allocator() )
-> concurrent_vector<iterator_value_t<InputIterator>,
                     Allocator>;
```

Where type alias `iterator_value_t` defines as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

Example

```
#include <tbb/concurrent_vector.h>
#include <array>
#include <memory>

int main() {
    std::array<int, 100> arr;

    // Deduces cv1 as tbb::concurrent_vector<int>
    tbb::concurrent_vector cv1(arr.begin(), arr.end());

    std::allocator<int> alloc;

    // Deduces cv2 as tbb::concurrent_vector<int, std::allocator<int>>
    tbb::concurrent_vector cv2(arr.begin(), arr.end(), alloc);
}
```

Queues

concurrent_queue

[containers.concurrent_queue]

`tbb::concurrent_queue` is a class template for an unbounded first-in-first-out data structure that permits multiple threads to concurrently push and pop items.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_queue.h>

namespace tbb {

    template <typename T, typename Allocator = cache_aligned_allocator<T>>
    class concurrent_queue {
    public:
        using value_type = T;
        using reference = T&;
        using const_reference = const T&;
        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;
        using allocator_type = Allocator;

        using size_type = <implementation-defined unsigned integer type>;
        using difference-type = <implementation-defined signed integer type>;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        // Construction, destruction, copying
        concurrent_queue();

        explicit concurrent_queue( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_queue( InputIterator first, InputIterator last,
                          const allocator_type& alloc = allocator_type() );

        concurrent_queue( const concurrent_queue& other );
        concurrent_queue( const concurrent_queue& other, const allocator_type& alloc␣
↪);

        concurrent_queue( concurrent_queue&& other );
        concurrent_queue( concurrent_queue&& other, const allocator_type& alloc );

        ~concurrent_queue();

        void push( const value_type& value );
        void push( value_type&& value );

        template <typename... Args>
        void emplace( Args&&... args );
```

(continues on next page)

(continued from previous page)

```
        bool try_pop( value_type& result );

        allocator_type get_allocator() const;

        size_type unsafe_size() const;
        bool empty() const;

        void clear();

        iterator unsafe_begin();
        const_iterator unsafe_begin() const;
        const_iterator unsafe_cbegin() const;

        iterator unsafe_end();
        const_iterator unsafe_end() const;
        const_iterator unsafe_cend() const;
    }; // class concurrent_queue

} // namespace tbb
```

Requirements:

- The type `T` must meet the `Erasable` requirements from the [container.requirements] ISO C++ Standard section. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_queue();

explicit concurrent_queue( const allocator_type& alloc );
```

Constructs an empty `concurrent_queue`. If provided, uses the allocator `alloc` to allocate the memory.

Constructor from the sequence of elements

```
template <typename InputIterator>
concurrent_queue( InputIterator first, InputIterator last,
                  const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_queue` containing all elements from the half-open interval `[first, last)` using the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

Copying constructors

```
concurrent_queue( const concurrent_queue& other );

concurrent_queue( const concurrent_queue& other,
                  const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_queue( concurrent_queue&& other );

concurrent_queue( concurrent_queue&& other,
                  const allocator_type& alloc );
```

Constructs a `concurrent_queue` with the content of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_queue();
```

Destroys the `concurrent_queue`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Concurrently safe member functions

All member functions in this section can be performed concurrently with each other.

Pushing elements

```
void push( const value_type& value );
```

Pushes a copy of `value` into the container.

Requirements: the type `T` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
void push( value_type&& value );
```

Pushes `value` into the container using move semantics.

Requirements: the type `T` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

`value` is left in a valid, but unspecified state.

```
template <typename... Args>
void emplace( Args&&... args );
```

Pushes a new element constructed from `args` into the container.

Requirements: the type `T` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Popping elements

```
bool try_pop( value_type& value );
```

If the container is empty, does nothing.

Otherwise, copies the last element from the container and assigns it to `value`. The popped element is destroyed.

Requirements: the type `T` must meet the `MoveAssignable` requirements from the [moveassignable] ISO C++ Standard section.

Returns: `true` if the element was popped; `false`, otherwise.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator, associated with `*this`.

Concurrently unsafe member functions

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

The number of elements

```
size_type unsafe_size() const;
```

Returns: the number of elements in the container.

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

clear

```
void clear();
```

Removes all elements from the container.

Iterators

The types `concurrent_queue::iterator` and `concurrent_queue::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

unsafe_begin and unsafe_cbegin

```
iterator unsafe_begin();

const_iterator unsafe_begin() const;

const_iterator unsafe_cbegin() const;
```

Returns: an iterator to the first element in the container.

unsafe_end and unsafe_cend

```
iterator unsafe_end();

const_iterator unsafe_end() const;

const_iterator unsafe_cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Other

Deduction guides

Where possible, constructors of `tbb::concurrent_queue` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Allocator = cache_aligned_allocator<iterator_value_t<InputIterator>
↪>
concurrent_queue( InputIterator, InputIterator, const Allocator& = Allocator() )
-> concurrent_queue<iterator_value_t<InputIterator>, Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

Example

```
#include <tbb/concurrent_queue.h>
#include <vector>
#include <memory>

int main() {
    std::vector<int> vec;

    // Deduces cq1 as tbb::concurrent_queue<int>
    tbb::concurrent_queue cq1(vec.begin(), vec.end());

    // Deduces cq2 as tbb::concurrent_queue<int, std::allocator<int>>
    tbb::concurrent_queue cq2(vec.begin(), vec.end(), std::allocator<int>{})
}
```

concurrent_bounded_queue

[containers.concurrent_bounded_queue]

`tbb::concurrent_bounded_queue` is a class template for a bounded first-in-first-out data structure that permits multiple threads to concurrently push and pop items.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_queue.h>

namespace tbb {

    template <typename T, typename Allocator = cache_aligned_allocator<T>>
    class concurrent_bounded_queue {
    public:
        using value_type = T;
        using reference = T&;
        using const_reference = const T&;
```

(continues on next page)

(continued from previous page)

```
        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

        using allocator_type = Allocator;

        using size_type = <implementation-defined signed integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        concurrent_bounded_queue();

        explicit concurrent_bounded_queue( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_bounded_queue( InputIterator first, InputIterator last,
                                  const allocator_type& alloc = allocator_type() );

        concurrent_bounded_queue( const concurrent_bounded_queue& other );
        concurrent_bounded_queue( const concurrent_bounded_queue& other,
                                  const allocator_type& alloc );

        concurrent_bounded_queue( concurrent_bounded_queue&& other );
        concurrent_bounded_queue( concurrent_bounded_queue&& other,
                                  const allocator_type& alloc );

        ~concurrent_bounded_queue();

        allocator_type get_allocator() const;

        void push( const value_type& value );
        void push( value_type&& value );

        bool try_push( const value_type& value );
        bool try_push( value_type&& value );

        template <typename... Args>
        void emplace( Args&&... args );

        template <typename... Args>
        bool try_emplace( Args&&... args );

        void pop( value_type& result );

        bool try_pop( value_type& result );

        void abort();

        size_type size() const;

        bool empty() const;

        size_type capacity() const;
        void set_capacity( size_type new_capacity );
```

(continues on next page)

(continued from previous page)

```
        void clear();

        iterator unsafe_begin();
        const_iterator unsafe_begin() const;
        const_iterator unsafe_cbegin() const;

        iterator unsafe_end();
        const_iterator unsafe_end() const;
        const_iterator unsafe_cend() const;
    }; // class concurrent_bounded_queue

} // namespace tbb
```

Requirements:

- The type `T` must meet the `Erasable` requirements from the [container.requirements] ISO C++ Standard section. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_bounded_queue();

explicit concurrent_bounded_queue( const allocator_type& alloc );
```

Constructs an empty `concurrent_bounded_queue` with an unbounded capacity. If provided, uses the allocator `alloc` to allocate the memory.

Constructor from the sequence of elements

```
template <typename InputIterator>
concurrent_bounded_queue( InputIterator first, InputIterator last,
                          const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_bounded_queue` with an unbounded capacity and containing all elements from the half-open interval `[first, last)` using the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

Copying constructors

```
concurrent_bounded_queue( const concurrent_bounded_queue& other );

concurrent_bounded_queue( const concurrent_bounded_queue& other,
                          const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_bounded_queue( concurrent_bounded_queue&& other );

concurrent_bounded_queue( concurrent_bounded_queue&& other,
                  const allocator_type& alloc );
```

Constructs a `concurrent_bounded_queue` with the content of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_bounded_queue();
```

Destroys the `concurrent_bounded_queue`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Concurrently safe member functions

All member functions in this section can be performed concurrently with each other.

Pushing elements

```
void push( const value_type& value );
```

Waits until the number of items in the queue is less than the capacity and pushes a copy of `value` into the container.

Requirements: the type `T` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
bool try_push( const value_type& value );
```

If the number of items in the queue is less than the capacity, pushes a copy of `value` into the container.

Requirements: the type `T` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if the item was pushed; `false`, otherwise.

```
void push( value_type&& value );
```

Waits until the number of items in the queue is less than `capacity()` and pushes `value` into the container using move semantics.

Requirements: the type `T` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

`value` is left in a valid, but unspecified state.

```
bool try_push( value_type&& value );
```

If the number of items in the queue is less than the capacity, pushes `value` into the container using move semantics.

Requirements: the type `T` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

`value` is left in a valid, but unspecified state.

Returns: `true` if the item was pushed; `false`, otherwise.

```
template <typename... Args>
void emplace( Args&&... args );
```

Waits until the number of items in the queue is less than `capacity()` and pushes a new element constructed from `args` into the container.

Requirements: the type `T` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename... Args>
bool try_emplace( Args&&... args );
```

If the number of items in the queue is less than the capacity, pushes a new element constructed from `args` into the container.

Requirements: the type `T` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if the item was pushed; `false`, otherwise.

Popping elements

```
void pop( value_type& value );
```

Waits until the item becomes available, copies it from the container, and assigns it to the `value`. The popped element is destroyed.

Requirements: the type `T` must meet the `MoveAssignable` requirements from the [moveassignable] ISO C++ Standard section.

```
bool try_pop( value_type& value );
```

If the container is empty, does nothing.

Otherwise, copies the last element from the container and assigns it to the `value`. The popped element is destroyed.

Requirements: the type `T` must meet the `MoveAssignable` requirements from the [moveassignable] ISO C++ Standard section.

Returns: `true` if the element was popped; `false`, otherwise.

abort

```
void abort();
```

Wakes up any threads that are waiting on the queue via `push`, `pop`, or `emplace` operations and raises the `tbb::user_abort` exception on those threads.

Capacity of the queue

```
size_type capacity() const;
```

Returns: the maximum number of items that the queue can hold.

```
void set_capacity( size_type new_capacity ) const;
```

Sets the maximum number of items that the queue can hold to `new_capacity`.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator, associated with `*this`.

Concurrently unsafe member functions

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

The number of elements

```
size_type size() const;
```

Returns: the number of elements in the container.

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

clear

```
void clear();
```

Removes all elements from the container.

Iterators

The types `concurrent_bounded_queue::iterator` and `concurrent_bounded_queue::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

unsafe_begin and unsafe_cbegin

```
iterator unsafe_begin();

const_iterator unsafe_begin() const;

const_iterator unsafe_cbegin() const;
```

Returns: an iterator to the first element in the container.

unsafe_end and unsafe_cend

```
iterator unsafe_end();

const_iterator unsafe_end() const;

const_iterator unsafe_cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Other

Deduction guides

Where possible, constructors of `tbb::concurrent_bounded_queue` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Allocator = cache_aligned_allocator<iterator_value_t<InputIterator>
↪>
concurrent_bounded_queue( InputIterator, InputIterator, const Allocator& =␣
↪Allocator() )
-> concurrent_bounded_queue<iterator_value_t<InputIterator>, Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

Example

```
#include <tbb/concurrent_queue.h>
#include <vector>
#include <memory>

int main() {
    std::vector<int> vec;

    // Deduces cq1 as tbb::concurrent_bounded_queue<int>
    tbb::concurrent_bounded_queue cq1(vec.begin(), vec.end());

    // Deduces cq2 as tbb::concurrent_bounded_queue<int, std::allocator<int>>
    tbb::concurrent_bounded_queue cq2(vec.begin(), vec.end(), std::allocator<int>{})
}
```

concurrent_priority_queue

[containers.concurrent_priority_queue]

`tbb::concurrent_priority_queue` is a class template for an unbounded priority queue that permits multiple threads to concurrently push and pop items. Items are popped in a priority order.

Class Template Synopsis

```
namespace tbb {

    template <typename T, typename Compare = std::less<T>,
              typename Allocator = cache_aligned_allocator<T>>
    class concurrent_priority_queue {
    public:
        using value_type = T;
        using reference = T&;
        using const_reference = const T&;
        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;
        using allocator_type = Allocator;

        concurrent_priority_queue();
        explicit concurrent_priority_queue( const allocator_type& alloc );

        explicit concurrent_priority_queue( const Compare& compare,
                                            const allocator_type& alloc = allocator_
↪type() );

        explicit concurrent_priority_queue( size_type init_capacity, const allocator_
↪type& alloc = allocator_type() );

        explicit concurrent_priority_queue( size_type init_capacity, const Compare&␣
↪compare,
                                            const allocator_type& alloc = allocator_
↪type() );

        template <typename InputIterator>
        concurrent_priority_queue( InputIterator first, InputIterator last,
                                   const allocator_type& alloc = allocator_type() );

        template <typename InputIterator>
        concurrent_priority_queue( InputIterator first, InputIterator last,
                                   const Compare& compare, const allocator_type&␣
↪alloc = allocator_type() );

        concurrent_priority_queue( std::initializer_list<value_type> init,
                                   const allocator_type& alloc = allocator_type() );

        concurrent_priority_queue( std::initializer_list<value_type> init,
                                   const Compare& compare, const allocator_type&␣
↪alloc = allocator_type() );

        concurrent_priority_queue( const concurrent_priority_queue& other );
        concurrent_priority_queue( const concurrent_priority_queue& other, const␣
↪allocator_type& alloc );
```

(continues on next page)

(continued from previous page)

```
        concurrent_priority_queue( concurrent_priority_queue&& other );
        concurrent_priority_queue( concurrent_priority_queue&& other, const allocator_
↪type& alloc );

        ~concurrent_priority_queue();

        concurrent_priority_queue& operator=( const concurrent_priority_queue& other␣
↪);
        concurrent_priority_queue& operator=( concurrent_priority_queue&& other );
        concurrent_priority_queue& operator=( std::initializer_list<value_type> init␣
↪);

        template <typename InputIterator>
        void assign( InputIterator first, InputIterator last );

        void assign( std::initializer_list<value_type> init );

        void swap( concurrent_priority_queue& other );

        allocator_type get_allocator() const;

        void clear();

        bool empty() const;
        size_type size() const;

        void push( const value_type& value );
        void push( value_type&& value );

        template <typename... Args>
        void emplace( Args&&... args );

        bool try_pop( value_type& value );
    }; // class concurrent_priority_queue

}; // namespace tbb
```

Requirements:

- The type `T` must meet the `Erasable` requirements from [container.requirements] ISO C++ Standard section. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_priority_queue();

explicit concurrent_priority_queue( const allocator_type& alloc );

explicit concurrent_priority_queue( const Compare& compare, const allocator_
↪type& alloc );
```

Constructs an empty `concurrent_priority_queue`. The initial capacity is unspecified. If provided, uses the predicate `compare` for priority comparisons and the allocator `alloc` to allocate the memory.

```
concurrent_priority_queue( size_type init_capacity,
                           const allocator_type& alloc = allocator_type() );

concurrent_priority_queue( size_type init_capacity,
                           const Compare& compare,
                           const allocator_type& alloc = allocator_type() );
```

Constructs an empty `concurrent_priority_queue` with the initial capacity `init_capacity`. If provided, uses the predicate `compare` for priority comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_priority_queue( InputIterator first, InputIterator last,
                           const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_priority_queue( InputIterator first, InputIterator last,
                           const Compare& compare,
                           const allocator_type& alloc = allocator_type() );
```

Constructs a `concurrent_priority_queue` containing all elements from the half-open interval `[first, last)`.

If provided, uses the predicate `compare` for priority comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_priority_queue( std::initializer_list<value_type> init,
                           const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_priority_queue(init.begin(), init.end(), alloc)`.

```
concurrent_priority_queue( std::initializer_list<value_type> init,
                           const Compare& compare,
                           const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_priority_queue(init.begin(), init.end(), compare, alloc)`.

Copying constructors

```
concurrent_priority_queue( const concurrent_priority_queue& other );

concurrent_priority_queue( const concurrent_priority_queue& other,
                           const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_priority_queue( concurrent_priority_queue&& other );

concurrent_priority_queue( concurrent_priority_queue&& other,
                           const allocator_type& alloc );
```

Constructs a copy of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_priority_queue();
```

Destroys the `concurrent_priority_queue`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_priority_queue& operator=( const concurrent_priority_queue& other␣
↪);
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_priority_queue& operator=( concurrent_priority_queue&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_priority_queue& operator=( std::initializer_list<value_type> init␣
↪);
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

assign

```
template <typename InputIterator>
void assign( InputIterator first, InputIterator last );
```

Replaces all elements in `*this` be the elements in the half-open interval `[first, last)`.

The behavior is undefined in case of concurrent operations with `*this`.

Requirements: the type `InputIterator` must meet the *InputIterator* requirements from the `[input.iterators]` ISO C++ Standard section.

```
void assign( std::initializer_list<value_type> init );
```

Equivalent to `assign(init.begin(), init.end())`.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent `push` or `try_pop` operations.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual number of elements in case of pending concurrent `push` or `try_pop` operations.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other.

Pushing elements

```
void push( const value_type& value );
```

Pushes a copy of `value` into the container.

Requirements: the type `T` must meet the `CopyInsertable` requirements from [container.requirements] and the `CopyAssignable` requirements from [copyassignable] ISO C++ Standard sections.

```
void push( value_type&& value );
```

Pushes `value` into the container using move semantics.

Requirements: the type `T` must meet the `MoveInsertable` requirements from [container.requirements] and the `MoveAssignable` requirements from [moveassignable] ISO C++ Standard sections.

`value` is left in a valid, but unspecified state.

```
template <typename... Args>
void emplace( Args&&... args );
```

Pushes a new element constructed from `args` into the container.

Requirements: the type `T` must meet the `EmplaceConstructible` requirements from [container.requirements] and the `MoveAssignable` requirements from [moveassignable] ISO C++ Standard sections.

Popping elements

```
bool try_pop( value_type& value )
```

If the container is empty, does nothing.

Otherwise, copies the highest priority element from the container and assigns it to `value`. The popped element is destroyed.

Requirements: the type `T` must meet the `MoveAssignable` requirements from the [moveassignable] ISO C++ Standard section.

Returns: `true` if the element was popped; `false`, otherwise.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

clear

```
void clear();
```

Removes all elements from the container.

swap

```
void swap( concurrent_priority_queue& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

Non-member functions

These functions provides binary comparison and swap operations on `tbb::concurrent_priority_queue` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_priority_queue` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_priority_queue<T, Compare, Allocator>& lhs,
           concurrent_priority_queue<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
```

(continues on next page)

(continued from previous page)

```
                const concurrent_priority_queue<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                const concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_priority_queue<T, Compare, Allocator>& lhs,
           concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

```
template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                const concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Checks if `lhs` is equal to `rhs`, that is they have the same number of elements and `lhs` contains all elements from `rhs` with the same priority.

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                const concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

Where possible, constructors of `tbb::concurrent_priority_queue` support class template argument deduction (since C++17):

```
template <typename InputIterator>
concurrent_priority_queue( InputIterator, InputIterator)
-> concurrent_priority_queue<iterator_value_t<InputIterator>>>;

template <typename InputIterator, typename Compare>
concurrent_priority_queue( InputIterator, InputIterator, const Compare& )
-> concurrent_priority_queue<iterator_value_t<InputIterator>,
                             Compare>;

template <typename InputIterator, typename Allocator>
concurrent_priority_queue( InputIterator, InputIterator, const Allocator& alloc )
```

(continues on next page)

(continued from previous page)

```
-> concurrent_priority_queue<iterator_value_t<InputIterator>,
                             std::less<iterator_value_t<InputIterator>,
                             Allocator>;

template <typename InputIterator, typename Compare, typename Allocator>
concurrent_priority_queue( InputIterator, InputIterator, const Compare&,
                           const Allocator& )
-> concurrent_priority_queue<iterator_value_t<InputIterator>,
                             Compare, Allocator>;

template <typename T>
concurrent_priority_queue( std::initializer_list<T> )
-> concurrent_priority_queue<T>;

template <typename T, typename Compare>
concurrent_priority_queue( std::initializer_list<T>, const Compare& )
-> concurrent_priority_queue<T, Compare>;

template <typename T, typename Allocator>
concurrent_priority_queue( std::initializer_list<T>, const Allocator& )
-> concurrent_priority_queue<T, std::less<T>, Allocator>;

template <typename T, typename Compare, typename Allocator>
concurrent_priority_queue( std::initializer_list<T>, const Compare&, const Allocator&␣
↪)
-> concurrent_priority_queue<T, Compare, Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

Example

```
#include <tbb/concurrent_priority_queue.h>
#include <vector>
#include <functional>

int main() {
    std::vector<int> vec;

    // Deduces cpq1 as tbb::concurrent_priority_queue<int>
    tbb::concurrent_priority_queue cpq1(vec.begin(), vec.end());

    // Deduces cpq2 as tbb::concurrent_priority_queue<int, std::greater>
    tbb::concurrent_priority_queue cpq2(vec.begin(), vec.end(), std::greater{});
}
```

Unordered associative containers

concurrent_hash_map

[containers.concurrent_hash_map]

`concurrent_hash_map` is a class template for an unordered associative container that holds key-value pairs with unique keys and supports concurrent insertion, lookup, and erasure.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_hash_map.h>

namespace tbb {

   template <typename Key, typename T,
             typename HashCompare = tbb_hash_compare<Key>,
             typename Allocator = tbb_allocator<std::pair<const Key, T>>>
   class concurrent_hash_map {
   public:
         using key_type = Key;
         using mapped_type = T;
         using value_type = std::pair<const Key, T>;

         using reference = value_type&;
         using const_reference = const value_type&;
         using pointer = typename std::allocator_traits<Allocator>::pointer;
         using const_pointer = typename std::allocator_traits<Allocator>
↪::const_pointer;

         using allocator_type = Allocator;

         using size_type = <implementation-defined unsigned integer type>;
         using difference_type = <implementation-defined signed integer type>
↪;

         using iterator = <implementation-defined ForwardIterator>;
         using const_iterator = <implementation-defined constant␣
↪ForwardIterator>;

         using range_type = <implementation-defined ContainerRange>;
         using const_range_type = <implementation-defined constant␣
↪ContainerRange>;

         class accessor;
         class const_accessor;

         // Construction, destruction, copying
         concurrent_hash_map();

         explicit concurrent_hash_map( const HashCompare& compare,
                                       const allocator_type& alloc =␣
↪allocator_type() );

         explicit concurrent_hash_map( const allocator_type& alloc );
```

(continues on next page)

(continued from previous page)

```
        concurrent_hash_map( size_type n, const HashCompare& compare,
                             const allocator_type& alloc = allocator_type()␣
↪);

        concurrent_hash_map( size_type n, const allocator_type& alloc =␣
↪allocator_type() );

        template <typename InputIterator>
        concurrent_hash_map( InputIterator first, InputIterator last,
                             const HashCompare& compare,
                             const allocator_type& alloc = allocator_type()␣
↪);

        template <typename InputIterator>
        concurrent_hash_map( InputIterator first, InputIterator last,
                             const allocator_type& alloc = allocator_type()␣
↪);

        concurrent_hash_map( std::initializer_list<value_type> init,
                             const HashCompare& compare,
                             const allocator_type& alloc = allocator_type()␣
↪);

        concurrent_hash_map( std::initializer_list<value_type> init,
                             const allocator_type& alloc = allocator_type()␣
↪);

        concurrent_hash_map( const concurrent_hash_map& other );
        concurrent_hash_map( const concurrent_hash_map& other,
                             const allocator_type& alloc );

        concurrent_hash_map( concurrent_hash_map&& other );
        concurrent_hash_map( concurrent_hash_map&& other,
                             const allocator_type& alloc );

        ~concurrent_hash_map();

        concurrent_hash_map& operator=( const concurrent_hash_map& other );
        concurrent_hash_map& operator=( concurrent_hash_map&& other );
        concurrent_hash_map& operator=( std::initializer_list<value_type>␣
↪init );

        allocator_type get_allocator() const;

        // Concurrently unsafe modifiers
        void clear();

        void swap( concurrent_hash_map& other );

        // Hash policy
        void rehash( size_type sz = 0 );
        size_type bucket_count() const;

        // Size and capacity
        size_type size() const;
        bool empty() const;
        size_type max_size() const;
```

(continues on next page)

(continued from previous page)

```
        // Lookup
        bool find( const_accessor& result, const key_type& key ) const;
        bool find( accessor& result, const key_type& key );

        size_type count( const key_type& key ) const;

        // Modifiers
        bool insert( const_accessor& result, const key_type& key );
        bool insert( accessor& result, const key_type& key );

        bool insert( const_accessor& result, const value_type& value );
        bool insert( accessor& result, const value_type& value );

        bool insert( const_accessor& result, value_type&& value );
        bool insert( accessor& result, value_type&& value );

        bool insert( const value_type& value );
        bool insert( value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        template <typename... Args>
        bool emplace( const_accessor& result, Args&&... args );

        template <typename... Args>
        bool emplace( accessor& result, Args&&... args );

        template <typename... Args>
        bool emplace( Args&&... args );

        bool erase( const key_type& key );

        bool erase( const_accessor& item_accessor );
        bool erase( accessor& item_accessor );

        // Iterators
        iterator begin();
        const_iterator begin() const;
        const_iterator cbegin() const;

        iterator end();
        const_iterator end() const;
        const_iterator cend() const;

        std::pair<iterator, iterator> equal_range( const key_type& key );
        std::pair<const_iterator, const_iterator> equal_range( const key_
↪type& key ) const;

        // Parallel iteration
        range_type range( std::size_t grainsize = 1 );
        const_range_type range( std::size_t grainsize = 1 ) const;
    }; // class concurrent_hash_map
```

(continues on next page)

(continued from previous page)

```
} // namespace tbb
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `HashCompare` must meet the *HashCompare requirements*.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member classes

accessor and const_accessor

Member classes `concurrent_hash_map::accessor` and `concurrent_hash_map::const_accessor` are called *accessors*. Accessors allow multiple threads to concurrently access the key-value pairs in `concurrent_hash_map`. An accessor is called *empty* if it does not point to any item.

`accessor` member class

Member class `concurrent_hash_map::accessor` provides read-write access to the key-value pair in `concurrent_hash_map`.

```
namespace tbb {

    template <typename Key, typename T, typename HashCompare, typename Allocator>
    class concurrent_hash_map<Key, T, HashCompare, Allocator>::accessor {
        using value_type = std::pair<const Key, T>;

        accessor();
        ~accessor();

        bool empty() const;
        value_type& operator*() const;
        value_type* operator->() const;

        void release();
    }; // class accessor

} // namespace tbb
```

`const_accessor` member class

Member class `concurrent_hash_map::const_accessor` provides read only access to the key-value pair in `concurrent_hash_map`.

```
namespace tbb {

    template <typename Key, typename T, typename HashCompare, typename Allocator>
    class concurrent_hash_map<Key, T, HashCompare, Allocator>::const_accessor {
        using value_type = const std::pair<const Key, T>;

        const_accessor();
        ~const_accessor();

        bool empty() const;
        value_type& operator*() const;
        value_type* operator->() const;

        void release();
    }; // class const_accessor

} // namespace tbb
```

Member functions

Construction and destruction

```
accessor();

const_accessor();
```

Constructs an empty accessor.

```
~accessor();

~const_accessor();
```

Destroys the accessor. If `*this` is not empty, releases the ownership of the element.

Emptiness

```
bool empty() const;
```

Returns: `true` if the accessor is empty; `false`, otherwise.

Key-value pair access

```
value_type& operator*() const;
```

Returns: a reference to the key-value pair to which the accessor points.

The behavior is undefined if the accessor is empty.

```
value_type* operator->() const;
```

Returns: a pointer to the key-value pair to which the accessor points.

The behavior is undefined if the accessor is empty.

Releasing

```
void release();
```

If `*this` is not empty, releases the ownership of the element. `*this` becomes empty.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_hash_map();

explicit concurrent_hash_map( const HashCompare& compare,
                              const allocator_type& alloc = allocator_type()␣
↪);

explicit concurrent_hash_map( const allocator_type& alloc );
```

Constructs an empty `concurrent_hash_map`. The initial number of buckets is unspecified.

If provided, uses the comparator `compare` to calculate hash codes and compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

```
concurrent_hash_map( size_type n, const HashCompare& compare,
                     const allocator_type& alloc = allocator_type() );

concurrent_hash_map( size_type n, const allocator_type& alloc = allocator_
↪type() );
```

Constructs an empty `concurrent_hash_map` with `n` preallocated buckets.

If provided, uses the comparator `compare` to calculate hash codes and compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_hash_map( InputIterator first, InputIterator last,
                     const HashCompare& compare,
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_hash_map( InputIterator first, InputIterator last,
                     const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_hash_map` which contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple elements with equal keys, it is unspecified which element would be inserted.

If provided, uses the comparator `compare` to calculate hash codes and compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_hash_map( std::initializer_list<value_type> init,
                     const HashCompare& compare,
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_hash_map(init.begin(), init.end(), compare, alloc)`.

```
concurrent_hash_map( std::initializer_list<value_type> init,
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_hash_map(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_hash_map( const concurrent_hash_map& other );

concurrent_hash_map( const concurrent_hash_map& other,
                     const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_hash_map( concurrent_hash_map&& other );

concurrent_hash_map( concurrent_hash_map&& other,
                     const allocator_type& alloc );
```

Constructs a `concurrent_hash_map` with the content of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_hash_map();
```

Destroys the `concurrent_hash_map`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_hash_map& operator=( const concurrent_hash_map& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_hash_map& operator=( concurrent_hash_map&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_hash_map& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element is inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

clear

```
void clear();
```

Removes all elements from the container.

swap

```
void swap( concurrent_hash_map& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Hash policy

Rehashing

```
void rehash( size_type n = 0 );
```

If `n > 0`, sets the number of buckets to the value that is not less than `n`.

bucket_count

```
size_type bucket_count() const;
```

Returns: the number of buckets in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ with the actual container state in case of pending concurrent insertions or erasures.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ with the actual container state in case of pending concurrent insertions or erasures.

max_size

```
size_type max_size() const;
```

Returns: The maximum number of elements that container can hold.

Lookup

All methods in this section can be executed concurrently with each other and concurrently-safe modifiers.

find

```
bool find( const_accessor& result, const key_type& key ) const;

bool find( accessor& result, const key_type& key );
```

If the accessor `result` is not empty, releases the `result`.

If an element with the key that is equivalent to `key` exists, sets the `result` to provide access to this element.

Returns: `true` if an element with the key equivalent to `key` is found; `false`, otherwise.

count

```
size_type count( const key_type& key ) const;
```

Returns: `1` if an element with the equivalent to `key` exists; `0`, otherwise.

Concurrently safe modifiers

All methods in this section can be executed concurrently with each other and lookup methods.

Inserting values

```
bool insert( const_accessor& result, const key_type& key );

bool insert( accessor& result, const key_type& key );
```

If the accessor `result` is not empty, releases the `result` and attempts to insert the value constructed from `key, mapped_type()` into the container.

Sets the `result` to provide access to the inserted element or to the element with equal key that was already presented in the container.

Requirements:

- the type `value_type` must meet the `EmplaceConstructible` requirements the from [container.requirements] ISO C++ Standard section.
- the type `mapped_type` must meet the `DefaultConstructible` requirements from the [defaultconstructible] ISO C++ Standard section.

Returns: `true` if an element was inserted; `false`, otherwise.

```
bool insert( const_accessor& result, const value_type& value );

bool insert( accessor& result, const value_type& value );
```

If the accessor `result` is not empty, releases the `result` and attempts to insert the value `value` into the container.

Sets the `result` to provide access to the inserted element or to the element with equal key that was already presented in the container.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if an element was inserted; `false`, otherwise.

```
bool insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if an element was inserted; `false`, otherwise.

```
bool insert( const_accessor& result, value_type&& value );

bool insert( accessor& result, value_type&& value );
```

If the accessor `result` is not empty, releases the `result` and attempts to insert the value `value` into the container using move semantics.

Sets the `result` to provide access to the inserted element or to the element with equal key that was already presented in the container.

`value` is left in a valid, but unspecified state.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if an element was inserted; `false`, otherwise.

```
bool insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if an element was inserted; `false`, otherwise.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple elements with equal keys, it is unspecified which element should be inserted.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Emplacing elements

```
template <typename... Args>
bool emplace( const_accessor& result, Args&&... args );

template <typename... Args>
bool emplace( accessor& result, Args&&... args );
```

If the accessor `result` is not empty, releases the `result` and attempts to insert an element constructed in-place from `args` into the container.

Sets the `result` to provide access to the inserted element or to the element with equal key that was already presented in the container.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if an element was inserted; `false`, otherwise

```
template <typename... Args>
bool emplace( Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Returns: `true` if an element was inserted; `false`, otherwise

Erasing elements

```
bool erase( const key_type& key );
```

If an element with the key equivalent to `key` exists, removes it from the container.

Returns: `true` if an element was removed; `false`, otherwise.

```
bool erase( const_accessor& item_accessor );
bool erase( accessor& item_accessor );
```

Removes an element owned by `item_accessor` from the container.

Requirements: `item_accessor` should not be empty.

Returns: `true` if an element was removed by the current thread; `false` if it was removed by another thread.

Iterators

The types `concurrent_hash_map::iterator` and `concurrent_hash_map::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

If an element with the key that is equivalent to `key` exists in the container, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

Parallel iteration

Member types `concurrent_hash_map::range_type` and `concurrent_hash_map::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_hash_map::const_range_type` are of type `concurrent_hash_map::const_iterator`, whereas the bounds for a `concurrent_hash_map::range_type` are of type `concurrent_hash_map::iterator`.

Traversing the `concurrent_hash_map` is not thread safe. The behavior is undefined in case of concurrent execution of any member functions while traversing the `range_type` or `const_range_type`.

range member function

```
range_type range( std::size_t grainsize = 1 );

const_range_type range( std::size_t grainsize = 1 ) const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary comparison and swap operations on `tbb::concurrent_hash_map` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_hash_map` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator==( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                 const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );

template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator!=( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                 const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );

template <typename Key, typename T, typename HashCompare, typename Allocator>
void swap( concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
           concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Non-member swap

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
void swap( concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
           concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_hash_map` are equal if the following conditions are `true`:

- They contain equal number of elements.
- Each element from one container is also available in the other.

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator==( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                 const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Returns: `true` if `lhs` is equivalent to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator!=( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                 const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

Where possible, constructors of `concurrent_hash_map` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename HashCompare,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_hash_map( InputIterator, InputIterator,
                     const HashCompare&,
                     const Allocator& = Allocator() )
-> concurrent_hash_map<iterator_key_t<InputIterator>,
                       iterator_mapped_t<InputIterator>,
                       HashCompare,
                       Allocator>;

template <typename InputIterator,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_hash_map( InputIterator, InputIterator,
                     const Allocator& = Allocator() )
-> concurrent_hash_map<iterator_key_t<InputIterator>,
                       iterator_mapped_t<InputIterator>,
                       tbb_hash_compare<iterator_key_t<InputIterator>>,
                       Allocator>;

template <typename Key,
          typename T,
          typename HashCompare,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_hash_map( std::initializer_list<std::pair<const Key, T>>,
                     const HashCompare&,
                     const Allocator& = Allocator() )
-> concurrent_hash_map<Key,
                       T,
                       HashCompare,
                       Allocator>;

template <typename Key,
          typename T,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_hash_map( std::initializer_list<std::pair<const Key, T>>,
                     const Allocator& = Allocator() )
-> concurrent_hash_map<Key,
                       T,
                       tbb_hash_compare<Key>,
                       Allocator>;
```

Where the type aliases `iterator_key_t`, `iterator_mapped_t`, and `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits
↪<InputIterator>::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
↪<InputIterator>,
                                         iterator_mapped_t<InputIterator>>>;
```

Example

```
#include <tbb/concurrent_hash_map.h>
#include <vector>

int main() {
    std::vector<std::pair<const int, float>> v;

    // Deduces chmap1 as tbb::concurrent_hash_map<int, float>
    tbb::concurrent_hash_map chmap1(v.begin(), v.end());

    std::allocator<std::pair<const int, float>> alloc;
    // Deduces chmap2 as tbb::concurrent_hash_map<int, float,
    //                                            tbb_hash_compare<int>,
    //                                            std::allocator<std::pair<const int,␣
↪float>>>
    tbb::concurrent_hash_map chmap2(v.begin(), v.end(), alloc);
}
```

concurrent_unordered_map

[containers.concurrent_unordered_map]

`tbb::concurrent_unordered_map` is a class template that represents an unordered associative container. It stores key-value pairs with unique keys and supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_unordered_map.h>

namespace tbb {

    template <typename Key,
              typename T,
              typename Hash = std::hash<Key>,
              typename KeyEqual = std::equal_to<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>>
```

(continues on next page)

(continued from previous page)

```
    class concurrent_unordered_map {
    public:
        using key_type = Key;
        using mapped_type = T;
        using value_type = std::pair<const Key, T>;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using hasher = Hash;
        using key_equal = /*See below*/;

        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;

        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        using local_iterator = <implementation-defined ForwardIterator>;
        using const_local_iterator = <implementation-defined constant ForwardIterator>
↪;

        using node_type = <implementation-defined node handle>;

        using range_type = <implementation-defined ContainerRange>;
        using const_range_type = <implementation-defined constant ContainerRange>;

        // Construction, destruction, copying
        concurrent_unordered_map();

        explicit concurrent_unordered_map( size_type bucket_count, const hasher& hash␣
↪= hasher(),
                                           const key_equal& equal = key_equal(),
                                           const allocator_type& alloc = allocator_
↪type() );

        concurrent_unordered_map( size_type bucket_count, const allocator_type& alloc␣
↪);

        concurrent_unordered_map( size_type bucket_count, const hasher& hash,
                                  const allocator_type& alloc );

        explicit concurrent_unordered_map( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_unordered_map( InputIterator first, InputIterator last,
                                  size_type bucket_count = /*implementation-defined*/,
                                  const hasher& hash = hasher(),
                                  const key_equal& equal = key_equal(),
                                  const allocator_type& alloc = allocator_type() );
```

(continues on next page)

(continued from previous page)

```
        template <typename Inputiterator>
        concurrent_unordered_map( InputIterator first, InputIterator last,
                                  size_type bucket_count, const allocator_type& alloc␣
↪);

        template <typename InputIterator>
        concurrent_unordered_map( InputIterator first, InputIterator last,
                                  size_type bucket_count, const hasher& hash,
                                  const allocator_type& alloc );

        concurrent_unordered_map( std::initializer_list<value_type> init,
                                  size_type bucket_count = /*implementation-defined*/,
                                  const hasher& hash = hasher(),
                                  const key_equal& equal = key_equal(),
                                  const allocator_type& alloc = allocator_type() );

        concurrent_unordered_map( std::initializer_list<value_type> init,
                                  size_type bucket_count, const allocator_type& alloc␣
↪);

        concurrent_unordered_map( std::initializer_list<value_type> init,
                                  size_type bucket_count, const hasher& hash,
                                  const allocator_type& alloc );

        concurrent_unordered_map( const concurrent_unordered_map& other );
        concurrent_unordered_map( const concurrent_unordered_map& other,
                                  const allocator_type& alloc );

        concurrent_unordered_map( concurrent_unordered_map&& other );
        concurrent_unordered_map( concurrent_unordered_map&& other,
                                  const allocator_type& alloc );

        ~concurrent_unordered_map();

        concurrent_unordered_map& operator=( const concurrent_unordered_map& other );
        concurrent_unordered_map& operator=( concurrent_unordered_map&& other )␣
↪noexcept(/*See details*/);

        concurrent_unordered_map& operator=( std::initializer_list<value_type> init );

        allocator_type get_allocator() const;

        // Iterators
        iterator begin() noexcept;
        const_iterator begin() const noexcept;
        const_iterator cbegin() const noexcept;

        iterator end() noexcept;
        const_iterator end() const noexcept;
        const_iterator cend() const noexcept;

        // Size and capacity
        bool empty() const noexcept;
        size_type size() const noexcept;
        size_type max_size() const noexcept;

        // Concurrently safe modifiers
```

(continues on next page)

(continued from previous page)

```
        std::pair<iterator, bool> insert( const value_type& value );
        iterator insert( const_iterator hint, const value_type& value );

        template <typename P>
        std::pair<iterator, bool> insert( P&& value );

        template <typename P>
        iterator insert( const_iterator hint, P&& value );

        std::pair<iterator, bool> insert( value_type&& value );
        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&
↪ source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&
↪& source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>& source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>&& source );

        // Concurrently unsafe modifiers
        void clear() noexcept;

        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );

        size_type unsafe_erase( const key_type& key );

        template <typename K>
        size_type unsafe_erase( const K& key );

        node_type unsafe_extract( const_iterator pos );
        node_type unsafe_extract( iterator pos );
```

(continues on next page)

(continued from previous page)

```
        node_type unsafe_extract( const key_type& key );

        template <typename K>
        node_type unsafe_extract( const K& key );

        void swap( concurrent_unordered_map& other );

        // Element access
        mapped_type& at( const key_type& key );
        const mapped_type& at( const key_type& key ) const;

        mapped_type& operator[]( const key_type& key );
        mapped_type& operator[]( key_type&& key );

        // Lookup
        size_type count( const key_type& key ) const;

        template <typename K>
        size_type count( const K& key ) const;

        iterator find( const key_type& key );
        const_iterator find( const key_type& key ) const;

        template <typename K>
        iterator find( const K& key );

        template <typename K>
        const_iterator find( const K& key ) const;

        bool contains( const key_type& key ) const;

        template <typename K>
        bool contains( const K& key ) const;

        std::pair<iterator, iterator> equal_range( const key_type& key );
        std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

        template <typename K>
        std::pair<iterator, iterator> equal_range( const K& key );

        template <typename K>
        std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

        // Bucket interface
        local_iterator unsafe_begin( size_type n );
        const_local_iterator unsafe_begin( size_type n ) const;
        const_local_iterator unsafe_cbegin( size_type n ) const;

        local_iterator unsafe_end( size_type n );
        const_local_iterator unsafe_end( size_type n ) const;
        const_local_iterator unsafe_cend( size_type n ) const;

        size_type unsafe_bucket_count() const;
        size_type unsafe_max_bucket_bount() const;
```

(continues on next page)

(continued from previous page)

```
        size_type unsafe_bucket_size( size_type n ) const;

        size_type unsafe_bucket( const key_type& key ) const;

        // Hash policy
        float load_factor() const;

        float max_load_factor() const;
        void max_load_factor( float ml );

        void rehash( size_type count );

        void reserve( size_type count );

        // Observers
        hasher hash_function() const;
        key_equal key_eq() const;

        // Parallel iteration
        range_type range();
        const_range_type range() const;
    }; // class concurrent_unordered_map

} // namespace tbb
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` must meet the `Hash` requirements from the [hash] ISO C++ Standard section.
- The type `KeyEqual` must meet the `BinaryPredicate` requirements from the [algorithms.general] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Description

`tbb::concurrent_unordered_map` is an unordered associative container, which elements are organized into buckets. The value of the hash function `Hash` for a `Key` object determines the number of the bucket in which the corresponding element will be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, the member type `concurrent_unordered_map::key_equal` is defined as the value of this qualified-id. In this case, the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type.

Otherwise, the member type `concurrent_unordered_map::key_equal` is defined as the value of the template parameter `KeyEqual`.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_unordered_map();

explicit concurrent_unordered_map( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_map`. The initial number of buckets is unspecified.

If provided, uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_map( size_type bucket_count,
                                   const hasher& hash = hasher(),
                                   const key_equal& equal = key_equal(),
                                   const allocator_type& alloc = allocator_
↪type() );

concurrent_unordered_map( size_type bucket_count, const allocator_type&␣
↪alloc );

concurrent_unordered_map( size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_map` with `bucket_count` buckets.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
                          size_type bucket_count = /*implementation-defined*/
↪,
                          const hasher& hash = hasher(),
                          const key_equal& equal = key_equal(),
                          const allocator_type& alloc = allocator_type() );

template <typename Inputiterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
                          size_type bucket_count, const allocator_type&␣
↪alloc );

template <typename InputIterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
                          size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Constructs the `concurrent_unordered_map` that contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple elements with equal keys, it is unspecified which element would be inserted.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_map( std::initializer_list<value_type> init,
                          size_type bucket_count = /*implementation-defined*/
↪,
                          const hasher& hash = hasher(),
                          const key_equal& equal = key_equal(),
                          const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_unordered_map(init.begin(), init.end(), bucket_count, hash, equal, alloc)`.

```
concurrent_unordered_map( std::initializer_list<value_type> init,
                          size_type bucket_count, const allocator_type&␣
↪alloc );
```

Equivalent to `concurrent_unordered_map(init.begin(), init.end(), bucket_count, alloc)`.

```
concurrent_unordered_map( std::initializer_list<value_type> init,
                          size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Equivalent to `concurrent_unordered_map(init.begin(), init.end(), bucket_count, hash, alloc)`.

Copying constructors

```
concurrent_unordered_map( const concurrent_unordered_map& other );

concurrent_unordered_map( const concurrent_unordered_map& other,
                          const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_unordered_map( concurrent_unordered_map&& other );

concurrent_unordered_map( concurrent_unordered_map&& other,
                          const allocator_type& alloc );
```

Constructs a `concurrent_unordered_map` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_unordered_map();
```

Destroys the `concurrent_unordered_map`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_unordered_map& operator=( const concurrent_unordered_map& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_unordered_map& operator=( concurrent_unordered_map&& other )␣
↪noexcept(/*See below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
↪equal::value &&
         std::is_nothrow_move_assignable<hasher>::value &&
         std::is_nothrow_move_assignable<key_equal>::value)
```

```
concurrent_unordered_map& operator=( std::initializer_list<value_type> init␣
↪);
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_unordered_map::iterator` and `concurrent_unordered_map::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& value );
```

Attempts to insert `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value );
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value );
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple elements with equal keys, it is unspecified which element should be inserted.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with key equal to `nh.key()`. Boolean value is `true` if insertion took place; `false`, otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element or to an existing element with key equal to `nh.key()`.

Merging containers

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪&& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>&& source );
```

Transfers those elements from `source` which keys do not exist in the container.

In case of merging with the container with multiple elements with equal keys, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: `1` if an element with the key equivalent to `key` exists; `0`, otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: `1` if an element with the key equivalent to `key` exists; `0`, otherwise.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

swap

```
void swap( concurrent_unordered_map& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
↪equal::value &&
         std::is_nothrow_swappable<hasher>::value &&
         std::is_nothrow_swappable<key_equal>::value
```

Element access

at

```
value_type& at( const key_type& key );

const value_type& at( const key_type& key ) const;
```

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

Throws: `std::out_of_range` exception if the element with the key equivalent to `key` is not presented in the container.

operator[]

```
value_type& operator[]( const key_type& key );
```

If the element with the key equivalent to `key` is not presented in the container, inserts a new element constructed in-place from `std::piecewise_construct, std::forward_as_tuple(key), std::tuple<>()`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

```
value_type& operator[]( key_type&& key );
```

If the element with the key equivalent to `key` is not presented in the container, inserts a new element constructed in-place from `std::piecewise_construct, std::forward_as_tuple(std::move(key)), std::tuple<>()`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Returns: a reference to `item.second` where `item` is the element with the key equivalent to `key`.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements with the key equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements with the key that is equivalent to `key`.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element with the key that is equivalent to `key`, or `end()` if no such element exists.

These overloads only participate in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if an element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if an element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

These overloads only participate in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

Bucket interface

The types `concurrent_unordered_map::local_iterator` and `concurrent_unordered_map::const_local_ite` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

Use these iterators to traverse a certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );

const_local_iterator unsafe_begin( size_type n ) const;

const_local_iterator unsafe_cbegin( size_type n ) const;
```

Returns: an iterator to the first element in the bucket number `n`.

```
local_iterator unsafe_end( size_type n );

const_local_iterator unsafe_end( size_type n ) const;

const_local_iterator unsafe_cend( size_type n ) const;
```

Returns: an iterator to the element that follows the last element in the bucket number `n`.

The number of buckets

```
size_type unsafe_bucket_count() const;
```

Returns: the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

Returns: the maximum number of buckets that container can hold.

Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

Returns: the number of elements in the bucket number `n`.

Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

Returns: the number of the bucket in which the element with the key `key` is stored.

Hash policy

Hash policy of `concurrent_unordered_map` manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

Load factor

```
float load_factor() const;
```

Returns: the average number of elements per bucket, which is `size()/unsafe_bucket_count()`.

```
float max_load_factor() const;
```

Returns: the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to `ml`.

Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to `n` and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value that is needed to store `n` elements.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

hash_function

```
hasher hash_function() const;
```

Returns: a copy of the hash function associated with `*this`.

key_eq

```
key_equal key_eq() const;
```

Returns: a copy of the key equality predicate associated with `*this`.

Parallel iteration

Member types `concurrent_unordered_map::range_type` and `concurrent_unordered_map::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_map::const_range_type` are of type `concurrent_unordered_map::const_iterator`, whereas the bounds for a `concurrent_unordered_map::range_type` are of type `concurrent_unordered_map::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary comparison and swap operations on `tbb::concurrent_unordered_map` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_unordered_map` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>&␣
↪rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>&␣
↪rhs );
```

Non-member swap

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs )␣
↪noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_unordered_map` are equal if the following conditions are `true`:

- They contains an equal number of elements.
- Each element from the one container is also available in the other.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>&␣
↪rhs );
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>&␣
↪rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

Where possible, constructors of `concurrent_unordered_map` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_key_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_key_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_unordered_map( InputIterator, InputIterator,
                          map_size_type = /*implementation_defined*/,
                          Hash = Hash(), KeyEqual = KeyEqual(),
                          Allocator = Allocator() )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                            iterator_mapped_t<InputIterator>,
                            Hash, KeyEqual, Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_map( InputIterator, InputIterator,
                          map_size_type,
                          Allocator )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                            iterator_mapped_t<InputIterator>,
                            std::hash<iterator_key_t<InputIterator>>,
```

(continues on next page)

(continued from previous page)

```
                            std::equal_to<iterator_key_t<InputIterator>>,
                            Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_map( InputIterator, InputIterator, Allocator )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                            iterator_mapped_t<InputIterator>,
                            std::hash<iterator_key_t<InputIterator>>,
                            std::equal_to<iterator_key_t<InputIterator>>,
                            Allocator>;

template <typename InputIterator,
          typename Hash,
          typename Allocator>
concurrent_unordered_map( InputIterator, InputIterator,
                          Hash, Allocator )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                            iterator_mapped_t<InputIterator>,
                            Hash,
                            std::equal_to<iterator_key_t<InputIterator>>,
                            Allocator>;

template <typename Key,
          typename T,
          typename Hash = std::hash<Key>,
          typename KeyEqual = std::equal_to<Key>,
          typename Allocator = tbb_allocator<std::pair<Key, T>>>
concurrent_unordered_map( std::initializer_list<value_type>,
                          map_size_type = /*implementation-defined*/,
                          Hash = Hash(),
                          KeyEqual = KeyEqual(),
                          Allocator = Allocator() )
-> concurrent_unordered_map<Key, T,
                            Hash,
                            KeyEqual,
                            Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_unordered_map( std::initializer_list<value_type>,
                          map_size_type, Allocator )
-> concurrent_unordered_map<Key, T,
                            std::hash<Key>,
                            std::equal_to<Key>,
                            Allocator>;

template <typename Key,
          typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_map( std::initializer_list<value_type>,
                          map_size_type, Hash, Allocator )
-> concurrent_unordered_map<Key, T,
                            Hash,
                            std::equal_to<Key>,
```

(continues on next page)

(continued from previous page)

```
                                Allocator>;
```

where the type `map_size_type` refers to the `size_type` member type of the deduced `concurrent_unordered_map` and the type aliases `iterator_key_t`, `iterator_mapped_t`, and `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits
↪<InputIterator>::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
↪<InputIterator>,
                                         iterator_mapped_t<InputIterator>>>;
```

Example

```
#include <tbb/concurrent_unordered_map.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces m1 as concurrent_unordered_map<int, float>
    tbb::concurrent_unordered_map m1(v.begin(), v.end());

    // Deduces m2 as concurrent_unordered_map<int, float, CustomHasher>;
    tbb::concurrent_unordered_map m2(v.begin(), v.end(), CustomHasher{});
}
```

concurrent_unordered_multimap

[containers.concurrent_unordered_multimap]

`tbb::concurrent_unordered_multimap` is a class template that represents an unordered associative container. It stores key-value pairs and supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure. In this container, multiple elements with equal keys can be stored.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_unordered_map.h>

namespace tbb {
    template <typename Key,
              typename T,
              typename Hash = std::hash<Key>,
              typename KeyEqual = std::equal_to<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>>
    class concurrent_unordered_multimap {
    public:
        using key_type = Key;
        using mapped_type = T;
        using value_type = std::pair<const Key, T>;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using hasher = Hash;
        using key_equal = /*See below*/;

        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;

        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        using local_iterator = <implementation-defined ForwardIterator>;
        using const_local_iterator = <implementation-defined constant ForwardIterator>
↪;

        using node_type = <implementation-defined node handle>;

        using range_type = <implementation-defined ContainerRange>;
        using const_range_type = <implementation-defined constant ContainerRange>;

        // Construction, destruction, copying
        concurrent_unordered_multimap();

        explicit concurrent_unordered_multimap( size_type bucket_count, const hasher&␣
↪hash = hasher(),
                                                const key_equal& equal = key_equal(),
                                                const allocator_type& alloc =␣
↪allocator_type() );

        concurrent_unordered_multimap( size_type bucket_count, const allocator_type&␣
↪alloc );

        concurrent_unordered_multimap( size_type bucket_count, const hasher& hash,
                                       const allocator_type& alloc );
```

(continues on next page)

(continued from previous page)

```
        explicit concurrent_unordered_multimap( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_unordered_multimap( InputIterator first, InputIterator last,
                                       size_type bucket_count = /*implementation-
↪defined*/,
                                       const hasher& hash = hasher(),
                                       const key_equal& equal = key_equal(),
                                       const allocator_type& alloc = allocator_type()␣
↪);

        template <typename Inputiterator>
        concurrent_unordered_multimap( InputIterator first, InputIterator last,
                                       size_type bucket_count, const allocator_type&␣
↪alloc );

        template <typename InputIterator>
        concurrent_unordered_multimap( InputIterator first, InputIterator last,
                                       size_type bucket_count, const hasher& hash,
                                       const allocator_type& alloc );

        concurrent_unordered_multimap( std::initializer_list<value_type> init,
                                       size_type bucket_count = /*implementation-
↪defined*/,
                                       const hasher& hash = hasher(),
                                       const key_equal& equal = key_equal(),
                                       const allocator_type& alloc = allocator_type()␣
↪);

        concurrent_unordered_multimap( std::initializer_list<value_type> init,
                                       size_type bucket_count, const allocator_type&␣
↪alloc );

        concurrent_unordered_multimap( std::initializer_list<value_type> init,
                                       size_type bucket_count, const hasher& hash,
                                       const allocator_type& alloc );

        concurrent_unordered_multimap( const concurrent_unordered_multimap& other );
        concurrent_unordered_multimap( const concurrent_unordered_multimap& other,
                                       const allocator_type& alloc );

        concurrent_unordered_multimap( concurrent_unordered_multimap&& other );
        concurrent_unordered_multimap( concurrent_unordered_multimap&& other,
                                       const allocator_type& alloc );

        ~concurrent_unordered_multimap();

        concurrent_unordered_multimap& operator=( const concurrent_unordered_multimap&
↪ other );
        concurrent_unordered_multimap& operator=( concurrent_unordered_multimap&&␣
↪other ) noexcept(/*See details*/);

        concurrent_unordered_multimap& operator=( std::initializer_list<value_type>␣
↪init );

        allocator_type get_allocator() const;
```

(continues on next page)

(continued from previous page)

```
        // Iterators
        iterator begin() noexcept;
        const_iterator begin() const noexcept;
        const_iterator cbegin() const noexcept;

        iterator end() noexcept;
        const_iterator end() const noexcept;
        const_iterator cend() const noexcept;

        // Size and capacity
        bool empty() const noexcept;
        size_type size() const noexcept;
        size_type max_size() const noexcept;

        // Concurrently safe modifiers
        std::pair<iterator, bool> insert( const value_type& value );
        iterator insert( const_iterator hint, const value_type& value );

        template <typename P>
        std::pair<iterator, bool> insert( P&& value );

        template <typename P>
        iterator insert( const_iterator hint, P&& value );

        std::pair<iterator, bool> insert( value_type&& value );
        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&
↪ source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&
↪& source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>& source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>&& source );
```

(continues on next page)

(continued from previous page)

```
        // Concurrently unsafe modifiers
        void clear() noexcept;

        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );

        size_type unsafe_erase( const key_type& key );

        template <typename K>
        size_type unsafe_erase( const K& key );

        node_type unsafe_extract( const_iterator pos );
        node_type unsafe_extract( iterator pos );

        node_type unsafe_extract( const key_type& key );

        template <typename K>
        node_type unsafe_extract( const K& key );

        void swap( concurrent_unordered_multimap& other );

        // Lookup
        size_type count( const key_type& key ) const;

        template <typename K>
        size_type count( const K& key ) const;

        iterator find( const key_type& key );
        const_iterator find( const key_type& key ) const;

        template <typename K>
        iterator find( const K& key );

        template <typename K>
        const_iterator find( const K& key ) const;

        bool contains( const key_type& key ) const;

        template <typename K>
        bool contains( const K& key ) const;

        std::pair<iterator, iterator> equal_range( const key_type& key );
        std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

        template <typename K>
        std::pair<iterator, iterator> equal_range( const K& key );

        template <typename K>
        std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

        // Bucket interface
        local_iterator unsafe_begin( size_type n );
        const_local_iterator unsafe_begin( size_type n ) const;
```

(continues on next page)

(continued from previous page)

```
        const_local_iterator unsafe_cbegin( size_type n ) const;

        local_iterator unsafe_end( size_type n );
        const_local_iterator unsafe_end( size_type n ) const;
        const_local_iterator unsafe_cend( size_type n ) const;

        size_type unsafe_bucket_count() const;
        size_type unsafe_max_bucket_bount() const;

        size_type unsafe_bucket_size( size_type n ) const;

        size_type unsafe_bucket( const key_type& key ) const;

        // Hash policy
        float load_factor() const;

        float max_load_factor() const;
        void max_load_factor( float ml );

        void rehash( size_type count );

        void reserve( size_type count );

        // Observers
        hasher hash_function() const;
        key_equal key_eq() const;

        // Parallel iteration
        range_type range();
        const_range_type range() const;
    }; // class concurrent_unordered_multimap
} // namespace tbb
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` must meet the `Hash` requirements from the [hash] ISO C++ Standard section.
- The type `KeyEqual` must meet the `BinaryPredicate` requirements from the [algorithms.general] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Description

`tbb::concurrent_unordered_multimap` is an unordered associative container, which elements are organized into buckets. The value of the hash function `Hash` for a `Key` object determines the number of the bucket in which the corresponding element will be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, the member type `concurrent_unordered_multimap::key_equal` is defined as the value of this qualified-id. In this case, the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type.

Otherwise, the type `concurrent_unordered_multimap::key_equal` is defined as the value of the template parameter `KeyEqual`.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_unordered_multimap();

explicit concurrent_unordered_multimap( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multimap`. The initial number of buckets is unspecified.

If provided uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_multimap( size_type bucket_count,
                                        const hasher& hash = hasher(),
                                        const key_equal& equal = key_equal(),
                                        const allocator_type& alloc =␣
↪allocator_type() );

concurrent_unordered_multimap( size_type bucket_count, const allocator_type&␣
↪alloc );

concurrent_unordered_multimap( size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multimap` with `bucket_count` buckets.

If provided uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
                               size_type bucket_count = /*implementation-
↪defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_
↪type() );

template <typename Inputiterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
                               size_type bucket_count, const allocator_type&␣
↪alloc );

template <typename InputIterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs the `concurrent_unordered_multimap` that contains all elements from the half-open interval `[first, last)`.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count = /*implementation-
↪defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_
↪type() );
```

Equivalent to `concurrent_unordered_multimap(init.begin(), init.end(), bucket_count, hash, equal, alloc)`.

```
concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count, const allocator_type&␣
↪alloc );
```

Equivalent to `concurrent_unordered_multimap(init.begin(), init.end(), bucket_count, alloc)`.

```
concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Equivalent to `concurrent_unordered_multimap(init.begin(), init.end(), bucket_count, hash, alloc)`.

Copying constructors

```
concurrent_unordered_multimap( const concurrent_unordered_multimap& other );

concurrent_unordered_multimap( const concurrent_unordered_multimap& other,
                               const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_unordered_multimap( concurrent_unordered_multimap&& other );

concurrent_unordered_multimap( concurrent_unordered_multimap&& other,
                               const allocator_type& alloc );
```

Constructs a `concurrent_unordered_multimap` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_unordered_multimap();
```

Destroys the `concurrent_unordered_multimap`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_unordered_multimap& operator=( const concurrent_unordered_
↪multimap& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_unordered_multimap& operator=( concurrent_unordered_multimap&&␣
↪other ) noexcept(/*See below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
↪equal::value &&
         std::is_nothrow_move_assignable<hasher>::value &&
         std::is_nothrow_move_assignable<key_equal>::value)
```

```
concurrent_unordered_multimap& operator=( std::initializer_list<value_type>␣
↪init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_unordered_multimap::iterator` and `concurrent_unordered_multimap::const_itera` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args )
```

Inserts an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args )
```

Inserts an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value )
```

Inserts the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, const value_type& other )
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value )
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value )
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value )
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, value_type&& other )
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last )
```

Inserts all items from the half-open interval `[first, last)` into the container.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init )
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh )
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, node_type&& nh )
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element.

Merging containers

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪&& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual,␣
↪Allocator>&& source );
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes all elements with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed elements.

Returns: the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes all elements with the key equivalent to `key` if they exist in the container.

Invalidates all iterators and references to the removed elements.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the number of removed elements.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If at least one element with the key equivalent to `key` exists, transfers ownership of one of these element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If at least one element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key equivalent to `key` exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

swap

```
void swap( concurrent_unordered_multimap& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
↪equal::value &&
         std::is_nothrow_swappable<hasher>::value &&
         std::is_nothrow_swappable<key_equal>::value
```

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements with the key equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements with the key equivalent to `key`.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be found.

These overloads only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if at least one element with the key equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if at least one element with the key equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element with the key equivalent to `key`, `l` is an iterator to the element which follows the last element with the key equivalent to `key`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if at least one element with the key equivalent to `key` exists - a pair of iterators `{f, l}`, where `f` is an iterator to the first element with the key equivalent to `key`, `l` is an iterator to the element that follows the last element with the key equivalent to `key`. Otherwise,``{end(), end()}``.

These overloads only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

Bucket interface

The types `concurrent_unordered_multimap::local_iterator` and `concurrent_unordered_multimap::const_local_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

These iterators are used to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );

const_local_iterator unsafe_begin( size_type n ) const;

const_local_iterator unsafe_cbegin( size_type n ) const;
```

Returns: an iterator to the first element in the bucket number `n`.

```
local_iterator unsafe_end( size_type n );

const_local_iterator unsafe_end( size_type n ) const;

const_local_iterator unsafe_cend( size_type n ) const;
```

Returns: an iterator to the element that follows the last element in the bucket number `n`.

The number of buckets

```
size_type unsafe_bucket_count() const;
```

Returns: the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

Returns: the maximum number of buckets that container can hold.

Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

Returns: the number of elements in the bucket number `n`.

Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

Returns: the number of the bucket in which the element with the key `key` is stored.

Hash policy

Hash policy of `concurrent_unordered_multimap` manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

Load factor

```
float load_factor() const;
```

Returns: the average number of elements per bucket, which is `size()/unsafe_bucket_count()`.

```
float max_load_factor() const;
```

Returns: the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to `ml`.

Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to `n` and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value that is needed to store `n` elements.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

hash_function

```
hasher hash_function() const;
```

Returns: a copy of the hash function associated with `*this`.

key_eq

```
key_equal key_eq() const;
```

Returns: a copy of the key equality predicate associated with `*this`.

Parallel iteration

Member types `concurrent_unordered_multimap::range_type` and `concurrent_unordered_multimap::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_multimap::const_range_type` are of type `concurrent_unordered_multimap::const_iterator`, whereas the bounds for a `concurrent_unordered_multimap::range_type` are of type `concurrent_unordered_multimap::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provides binary comparison and swap operations on `tbb::concurrent_unordered_multimap` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_unordered_multimap` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,␣
↪Allocator>& lhs,
                 const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,␣
↪Allocator>& rhs );
```

(continues on next page)

(continued from previous page)

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,␣
↪Allocator>& lhs,
                 const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,␣
↪Allocator>& rhs );
```

Non-member swap

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& rhs )␣
↪noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_unordered_multimap` are equal if the following conditions are `true`:

- They contain an equal number of elements.
- Each group of elements with the same key in one container has the corresponding group of equivalent elements in the other container (not necessary in the same order).

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,␣
↪Allocator>& lhs,
                 const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,␣
↪Allocator>& rhs );
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,␣
↪Allocator>& lhs,
                 const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,␣
↪Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

Where possible, constructors of `concurrent_unordered_multimap` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_key_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_key_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_unordered_multimap( InputIterator, InputIterator,
                               map_size_type = /*implementation_defined*/,
                               Hash = Hash(), KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                 iterator_mapped_t<InputIterator>,
                                 Hash, KeyEqual, Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_multimap( InputIterator, InputIterator,
                           map_size_type,
                           Allocator )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                 iterator_mapped_t<InputIterator>,
                                 std::hash<iterator_key_t<InputIterator>>,
                                 std::equal_to<iterator_key_t<InputIterator>>,
                                 Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_multimap( InputIterator, InputIterator, Allocator )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                 iterator_mapped_t<InputIterator>,
                                 std::hash<iterator_key_t<InputIterator>>,
                                 std::equal_to<iterator_key_t<InputIterator>>,
                                 Allocator>;

template <typename InputIterator,
          typename Hash,
          typename Allocator>
concurrent_unordered_multimap( InputIterator, InputIterator,
                           Hash, Allocator )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                 iterator_mapped_t<InputIterator>,
                                 Hash,
                                 std::equal_to<iterator_key_t<InputIterator>>,
                                 Allocator>;

template <typename Key,
          typename T,
          typename Hash = std::hash<Key>,
          typename KeyEqual = std::equal_to<Key>,
          typename Allocator = tbb_allocator<std::pair<Key, T>>>
concurrent_unordered_multimap( std::initializer_list<value_type>,
                               map_size_type = /*implementation-defined*/,
```

(continues on next page)

(continued from previous page)

```
                                Hash = Hash(),
                                KeyEqual = KeyEqual(),
                                Allocator = Allocator() )
-> concurrent_unordered_multimap<Key, T,
                                 Hash,
                                 KeyEqual,
                                 Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_unordered_multimap( std::initializer_list<value_type>,
                                map_size_type, Allocator )
-> concurrent_unordered_multimap<Key, T,
                                 std::hash<Key>,
                                 std::equal_to<Key>,
                                 Allocator>;

template <typename Key,
          typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_multimap( std::initializer_list<value_type>,
                                map_size_type, Hash, Allocator )
-> concurrent_unordered_multimap<Key, T,
                                 Hash,
                                 std::equal_to<Key>,
                                 Allocator>;
```

Where the type `map_size_type` refers to the `size_type` member type of the deduced `concurrent_multimap` and the type aliases `iterator_key_t`, `iterator_mapped_t` and `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits
↪<InputIterator>::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
↪<InputIterator>,
                         iterator_mapped_t<InputIterator>>>;
```

Example

```
#include <tbb/concurrent_unordered_map.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<std::pair<int, float>> v;
```

(continues on next page)

(continued from previous page)

```
    // Deduces m1 as concurrent_unordered_multimap<int, float>
    tbb::concurrent_unordered_multimap m1(v.begin(), v.end());

    // Deduces m2 as concurrent_unordered_multimap<int, float, CustomHasher>;
    tbb::concurrent_unordered_multimap m2(v.begin(), v.end(), CustomHasher{});
}
```

concurrent_unordered_set

[containers.concurrent_unordered_set]

`tbb::concurrent_unordered_set` is a class template that represents an unordered sequence of unique elements. It supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_unordered_set.h>

namespace tbb {
    template <typename T,
              typename Hash = std::hash<Key>,
              typename KeyEqual = std::equal_to<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>>
    class concurrent_unordered_set {
    public:
        using key_type = Key;
        using value_type = Key;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using hasher = Hash;
        using key_equal = /*See below*/;

        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;

        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        using local_iterator = <implementation-defined ForwardIterator>;
        using const_local_iterator = <implementation-defined constant ForwardIterator>
↪;

        using node_type = <implementation-defined node handle>;
```

(continues on next page)

(continued from previous page)

```
        using range_type = <implementation-defined ContainerRange>;
        using const_range_type = <implementation-defined constant ContainerRange>;

        // Construction, destruction, copying
        concurrent_unordered_set();

        explicit concurrent_unordered_set( size_type bucket_count, const hasher& hash␣
↪= hasher(),
                                           const key_equal& equal = key_equal(),
                                           const allocator_type& alloc = allocator_
↪type() );

        concurrent_unordered_set( size_type bucket_count, const allocator_type& alloc␣
↪);

        concurrent_unordered_set( size_type bucket_count, const hasher& hash,
                                  const allocator_type& alloc );

        explicit concurrent_unordered_set( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_unordered_set( InputIterator first, InputIterator last,
                                  size_type bucket_count = /*implementation-defined*/,
                                  const hasher& hash = hasher(),
                                  const key_equal& equal = key_equal(),
                                  const allocator_type& alloc = allocator_type() );

        template <typename Inputiterator>
        concurrent_unordered_set( InputIterator first, InputIterator last,
                                  size_type bucket_count, const allocator_type& alloc␣
↪);

        template <typename InputIterator>
        concurrent_unordered_set( InputIterator first, InputIterator last,
                                  size_type bucket_count, const hasher& hash,
                                  const allocator_type& alloc );

        concurrent_unordered_set( std::initializer_list<value_type> init,
                                  size_type bucket_count = /*implementation-defined*/,
                                  const hasher& hash = hasher(),
                                  const key_equal& equal = key_equal(),
                                  const allocator_type& alloc = allocator_type() );

        concurrent_unordered_set( std::initializer_list<value_type> init,
                                  size_type bucket_count, const allocator_type& alloc␣
↪);

        concurrent_unordered_set( std::initializer_list<value_type> init,
                                  size_type bucket_count, const hasher& hash,
                                  const allocator_type& alloc );

        concurrent_unordered_set( const concurrent_unordered_set& other );
        concurrent_unordered_set( const concurrent_unordered_set& other,
                                  const allocator_type& alloc );

        concurrent_unordered_set( concurrent_unordered_set&& other );
        concurrent_unordered_set( concurrent_unordered_set&& other,
```

(continues on next page)

(continued from previous page)

```
                                const allocator_type& alloc );

        ~concurrent_unordered_set();

        concurrent_unordered_set& operator=( const concurrent_unordered_set& other );
        concurrent_unordered_set& operator=( concurrent_unordered_set&& other )␣
↪noexcept(/*See details*/);

        concurrent_unordered_set& operator=( std::initializer_list<value_type> init );

        allocator_type get_allocator() const;

        // Iterators
        iterator begin() noexcept;
        const_iterator begin() const noexcept;
        const_iterator cbegin() const noexcept;

        iterator end() noexcept;
        const_iterator end() const noexcept;
        const_iterator cend() const noexcept;

        // Size and capacity
        bool empty() const noexcept;
        size_type size() const noexcept;
        size_type max_size() const noexcept;

        // Concurrently safe modifiers
        std::pair<iterator, bool> insert( const value_type& value );
        iterator insert( const_iterator hint, const value_type& value );

        std::pair<iterator, bool> insert( value_type&& value );
        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&␣
↪source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&
↪ source );
```

(continues on next page)

(continued from previous page)

```
        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&
↪& source );

        // Concurrently unsafe modifiers
        void clear() noexcept;

        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );

        size_type unsafe_erase( const key_type& key );

        template <typename K>
        size_type unsafe_erase( const K& key );

        node_type unsafe_extract( const_iterator pos );
        node_type unsafe_extract( iterator pos );

        node_type unsafe_extract( const key_type& key );

        template <typename K>
        node_type unsafe_extract( const K& key );

        void swap( concurrent_unordered_set& other );

        // Lookup
        size_type count( const key_type& key ) const;

        template <typename K>
        size_type count( const K& key ) const;

        iterator find( const key_type& key );
        const_iterator find( const key_type& key ) const;

        template <typename K>
        iterator find( const K& key );

        template <typename K>
        const_iterator find( const K& key ) const;

        bool contains( const key_type& key ) const;

        template <typename K>
        bool contains( const K& key ) const;

        std::pair<iterator, iterator> equal_range( const key_type& key );
        std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

        template <typename K>
        std::pair<iterator, iterator> equal_range( const K& key );

        template <typename K>
        std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;
```

(continues on next page)

(continued from previous page)

```
        // Bucket interface
        local_iterator unsafe_begin( size_type n );
        const_local_iterator unsafe_begin( size_type n ) const;
        const_local_iterator unsafe_cbegin( size_type n ) const;

        local_iterator unsafe_end( size_type n );
        const_local_iterator unsafe_end( size_type n ) const;
        const_local_iterator unsafe_cend( size_type n ) const;

        size_type unsafe_bucket_count() const;
        size_type unsafe_max_bucket_bount() const;

        size_type unsafe_bucket_size( size_type n ) const;

        size_type unsafe_bucket( const key_type& key ) const;

        // Hash policy
        float load_factor() const;

        float max_load_factor() const;
        void max_load_factor( float ml );

        void rehash( size_type count );

        void reserve( size_type count );

        // Observers
        hasher hash_function() const;
        key_equal key_eq() const;

        // Parallel iteration
        range_type range();
        const_range_type range() const;
    }; // class concurrent_unordered_set
} // namespace tbb
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` must meet the `Hash` requirements from the [hash] ISO C++ Standard section.
- The type `KeyEqual` must meet the `BinaryPredicate` requirements from the [algorithms.general] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Description

`tbb::concurrent_unordered_set` is an unordered sequence, which elements are organized into buckets. The value of the hash function `Hash` for `Key` object determines the number of the bucket in which the corresponding element will be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, the member type `concurrent_unordered_set::key_equal` is defined as the value of this qualified-id. In this case, the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type.

Otherwise, the member type `concurrent_unordered_set::key_equal` is defined as the value of the template parameter `KeyEqual`.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_unordered_set();

explicit concurrent_unordered_set( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_set`. The initial number of buckets is unspecified.

If provided, uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_set( size_type bucket_count,
                                   const hasher& hash = hasher(),
                                   const key_equal& equal = key_equal(),
                                   const allocator_type& alloc = allocator_
↪type() );

concurrent_unordered_set( size_type bucket_count, const allocator_type&␣
↪alloc );

concurrent_unordered_set( size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_set` with `bucket_count` buckets.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                          size_type bucket_count = /*implementation-defined*/
↪,
                          const hasher& hash = hasher(),
                          const key_equal& equal = key_equal(),
                          const allocator_type& alloc = allocator_type() );

template <typename Inputiterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                          size_type bucket_count, const allocator_type&␣
↪alloc );

template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                          size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Constructs the `concurrent_unordered_set` that contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple equal elements, it is unspecified which element would be inserted.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_set( std::initializer_list<value_type> init,
                          size_type bucket_count = /*implementation-defined*/
↪,
                          const hasher& hash = hasher(),
                          const key_equal& equal = key_equal(),
                          const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_unordered_set(init.begin(), init.end(), bucket_count, hash, equal, alloc)`.

```
concurrent_unordered_set( std::initializer_list<value_type> init,
                          size_type bucket_count, const allocator_type&␣
↪alloc );
```

Equivalent to `concurrent_unordered_set(init.begin(), init.end(), bucket_count, alloc)`.

```
concurrent_unordered_set( std::initializer_list<value_type> init,
                          size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );
```

Equivalent to `concurrent_unordered_set(init.begin(), init.end(), bucket_count, hash, alloc)`.

Copying constructors

```
concurrent_unordered_set( const concurrent_unordered_set& other );

concurrent_unordered_set( const concurrent_unordered_set& other,
                          const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_unordered_set( concurrent_unordered_set&& other );

concurrent_unordered_set( concurrent_unordered_set&& other,
                          const allocator_type& alloc );
```

Constructs a *concurrent_unordered_set* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_unordered_set();
```

Destroys the `concurrent_unordered_set`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_unordered_set& operator=( const concurrent_unordered_set& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_unordered_set& operator=( concurrent_unordered_set&& other )␣
↪noexcept(/*See below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
↪equal::value &&
         std::is_nothrow_move_assignable<hasher>::value &&
         std::is_nothrow_move_assignable<key_equal>::value)
```

```
concurrent_unordered_set& operator=( std::initializer_list<value_type> init␣
↪);
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple equal elements, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_unordered_set::iterator` and `concurrent_unordered_set::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing equal element. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Attempts to insert the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element or to an existing equal element.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing equal element. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element or to an existing equal element.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple equal elements, it is unspecified which element should be inserted.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element equal to `nh.value()`. Boolean value is `true` if insertion took place; `false`, otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element or to an existing element equal to `nh.value()`.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing equal element. Boolean value is `true` if insertion took place, `false` otherwise.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element or to an existing equal element.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ Standard section.

Merging containers

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&␣
↪source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>
↪& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>
↪&& source );
```

Transfers those elements from `source` that do not exist in the container.

In case of merging with the container with multiple equal elements, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable, and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: `1` if an element equivalent to `key` exists; `0`, otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: `1` if an element equivalent to `key` exists; `0`, otherwise.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable, and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload participates in overload resolution only if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

swap

```
void swap( concurrent_unordered_set& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
↪equal::value &&
         std::is_nothrow_swappable<hasher>::value &&
         std::is_nothrow_swappable<key_equal>::value
```

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements that is equivalent to `key`.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element equivalent to `key`, or `end()` if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element that is equivalent to `key`, or `end()` if no such element exists.

These overloads only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if an element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if an element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

These overloads participate in overload resolution only if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

Bucket interface

The types `concurrent_unordered_set::local_iterator` and `concurrent_unordered_set::const_local_ite` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

Use these iterators to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );

const_local_iterator unsafe_begin( size_type n ) const;

const_local_iterator unsafe_cbegin( size_type n ) const;
```

Returns: an iterator to the first element in the bucket number `n`.

```
local_iterator unsafe_end( size_type n );

const_local_iterator unsafe_end( size_type n ) const;

const_local_iterator unsafe_cend( size_type n ) const;
```

Returns: an iterator to the element that follows the last element in the bucket number `n`.

The number of buckets

```
size_type unsafe_bucket_count() const;
```

Returns: the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

Returns: the maximum number of buckets that container can hold.

Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

Returns: the number of elements in the bucket number `n`.

Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

Returns: the number of the bucket in which the element with the key `key` is stored.

Hash policy

Hash policy of `concurrent_unordered_set` manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

Load factor

```
float load_factor() const;
```

Returns: the average number of elements per bucket, which is `size()/unsafe_bucket_count()`.

```
float max_load_factor() const;
```

Returns: the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to `ml`.

Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to `n` and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value that is needed to store `n` elements.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

hash_function

```
hasher hash_function() const;
```

Returns: a copy of the hash function associated with `*this`.

key_eq

```
key_equal key_eq() const;
```

Returns: a copy of the key equality predicate associated with `*this`.

Parallel iteration

Member types `concurrent_unordered_set::range_type` and `concurrent_unordered_set::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_set::const_range_type` are of type `concurrent_unordered_set::const_iterator`, whereas the bounds for a `concurrent_unordered_set::range_type` are of type `concurrent_unordered_set::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary comparison and swap operations on `tbb::concurrent_unordered_set` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_unordered_set` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs )␣
↪noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_unordered_set` are equal if the following conditions are `true`:

- They contain an equal number of elements.
- Each element from one container is also available in the other.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );
```

Returns: `true` if `lhs` is equal to `rhs`, `false` otherwise.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                 const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Other

Deduction guides

Where possible, constructors of `concurrent_unordered_set` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_value_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_value_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_unordered_set( InputIterator, InputIterator,
                          map_size_type = /*implementation_defined*/,
                          Hash = Hash(), KeyEqual = KeyEqual(),
                          Allocator = Allocator() )
-> concurrent_unordered_set<iterator_value_t<InputIterator>,
                            Hash, KeyEqual, Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_set( InputIterator, InputIterator,
                          map_size_type,
                          Allocator )
-> concurrent_unordered_set<iterator_value_t<InputIterator>,
                            std::hash<iterator_value_t<InputIterator>>,
                            std::equal_to<iterator_value_t<InputIterator>>,
                            Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_set( InputIterator, InputIterator, Allocator )
```

(continues on next page)

(continued from previous page)

```
-> concurrent_unordered_set<iterator_value_t<InputIterator>,
                            std::hash<iterator_value_t<InputIterator>>,
                            std::equal_to<iterator_key_t<InputIterator>>,
                            Allocator>;

template <typename InputIterator,
          typename Hash,
          typename Allocator>
concurrent_unordered_set( InputIterator, InputIterator,
                          Hash, Allocator )
-> concurrent_unordered_set<iterator_value_t<InputIterator>,
                            Hash,
                            std::equal_to<iterator_value_t<InputIterator>>,
                            Allocator>;

template <typename T,
          typename Hash = std::hash<Key>,
          typename KeyEqual = std::equal_to<Key>,
          typename Allocator = tbb_allocator<std::pair<Key, T>>>
concurrent_unordered_set( std::initializer_list<value_type>,
                          map_size_type = /*implementation-defined*/,
                          Hash = Hash(),
                          KeyEqual = KeyEqual(),
                          Allocator = Allocator() )
-> concurrent_unordered_set<T,
                            Hash,
                            KeyEqual,
                            Allocator>;

template <typename T,
          typename Allocator>
concurrent_unordered_set( std::initializer_list<value_type>,
                          map_size_type, Allocator )
-> concurrent_unordered_set<T,
                            std::hash<Key>,
                            std::equal_to<Key>,
                            Allocator>;

template <typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_set( std::initializer_list<value_type>,
                          map_size_type, Hash, Allocator )
-> concurrent_unordered_set<T,
                            Hash,
                            std::equal_to<Key>,
                            Allocator>;
```

Where the type `map_size_type` refers to the `size_type` member type of the deduced `concurrent_unordered_set` and the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

Example

```
#include <tbb/concurrent_unordered_set.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<int> v;

    // Deduces s1 as concurrent_unordered_set<int>
    tbb::concurrent_unordered_set s1(v.begin(), v.end());

    // Deduces s2 as concurrent_unordered_set<int, CustomHasher>;
    tbb::concurrent_unordered_set s2(v.begin(), v.end(), CustomHasher{});
}
```

concurrent_unordered_multiset

[containers.concurrent_unordered_multiset]

`tbb::concurrent_unordered_multiset` is a class template that represents an unordered sequence of elements, It supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure. In this container, multiple equivalent elements can be stored.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_unordered_set.h>

namespace tbb {
    template <typename T,
              typename Hash = std::hash<Key>,
              typename KeyEqual = std::equal_to<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>>
    class concurrent_unordered_multiset {
    public:
        using key_type = Key;
        using value_type = Key;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using hasher = Hash;
        using key_equal = /*See below*/;

        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;

        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
↪pointer;

        using iterator = <implementation-defined ForwardIterator>;
```

(continues on next page)

(continued from previous page)

```
        using const_iterator = <implementation-defined constant ForwardIterator>;

        using local_iterator = <implementation-defined ForwardIterator>;
        using const_local_iterator = <implementation-defined constant ForwardIterator>
↪;

        using node_type = <implementation-defined node handle>;

        using range_type = <implementation-defined ContainerRange>;
        using const_range_type = <implementation-defined constant ContainerRange>;

        // Construction, destruction, copying
        concurrent_unordered_multiset();

        explicit concurrent_unordered_multiset( size_type bucket_count, const hasher&␣
↪hash = hasher(),
                                                const key_equal& equal = key_equal(),
                                                const allocator_type& alloc =␣
↪allocator_type() );

        concurrent_unordered_multiset( size_type bucket_count, const allocator_type&␣
↪alloc );

        concurrent_unordered_multiset( size_type bucket_count, const hasher& hash,
                                       const allocator_type& alloc );

        explicit concurrent_unordered_multiset( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_unordered_multiset( InputIterator first, InputIterator last,
                                       size_type bucket_count = /*implementation-
↪defined*/,
                                       const hasher& hash = hasher(),
                                       const key_equal& equal = key_equal(),
                                       const allocator_type& alloc = allocator_type()␣
↪);

        template <typename Inputiterator>
        concurrent_unordered_multiset( InputIterator first, InputIterator last,
                                       size_type bucket_count, const allocator_type&␣
↪alloc );

        template <typename InputIterator>
        concurrent_unordered_multiset( InputIterator first, InputIterator last,
                                       size_type bucket_count, const hasher& hash,
                                       const allocator_type& alloc );

        concurrent_unordered_multiset( std::initializer_list<value_type> init,
                                       size_type bucket_count = /*implementation-
↪defined*/,
                                       const hasher& hash = hasher(),
                                       const key_equal& equal = key_equal(),
                                       const allocator_type& alloc = allocator_type()␣
↪);

        concurrent_unordered_multiset( std::initializer_list<value_type> init,
                                       size_type bucket_count, const allocator_type&␣
↪alloc );
```

(continues on next page)

(continued from previous page)

```
        concurrent_unordered_multiset( std::initializer_list<value_type> init,
                                       size_type bucket_count, const hasher& hash,
                                       const allocator_type& alloc );

        concurrent_unordered_multiset( const concurrent_unordered_multiset& other );
        concurrent_unordered_multiset( const concurrent_unordered_multiset& other,
                                       const allocator_type& alloc );

        concurrent_unordered_multiset( concurrent_unordered_multiset&& other );
        concurrent_unordered_multiset( concurrent_unordered_multiset&& other,
                                       const allocator_type& alloc );

        ~concurrent_unordered_multiset();

        concurrent_unordered_multiset& operator=( const concurrent_unordered_multiset&
↪ other );
        concurrent_unordered_multiset& operator=( concurrent_unordered_multiset&&␣
↪other ) noexcept(/*See details*/);

        concurrent_unordered_multiset& operator=( std::initializer_list<value_type>␣
↪init );

        allocator_type get_allocator() const;

        // Iterators
        iterator begin() noexcept;
        const_iterator begin() const noexcept;
        const_iterator cbegin() const noexcept;

        iterator end() noexcept;
        const_iterator end() const noexcept;
        const_iterator cend() const noexcept;

        // Size and capacity
        bool empty() const noexcept;
        size_type size() const noexcept;
        size_type max_size() const noexcept;

        // Concurrently safe modifiers
        std::pair<iterator, bool> insert( const value_type& value );
        iterator insert( const_iterator hint, const value_type& value );

        std::pair<iterator, bool> insert( value_type&& value );
        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );
```

(continues on next page)

(continued from previous page)

```
        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&␣
↪source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&
↪ source );

        template <typename SrcHash, typename SrcKeyEqual>
        void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&
↪& source );

        // Concurrently unsafe modifiers
        void clear() noexcept;

        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );

        size_type unsafe_erase( const key_type& key );

        template <typename K>
        size_type unsafe_erase( const K& key );

        node_type unsafe_extract( const_iterator pos );
        node_type unsafe_extract( iterator pos );

        node_type unsafe_extract( const key_type& key );

        template <typename K>
        node_type unsafe_extract( const K& key );

        void swap( concurrent_unordered_multiset& other );

        // Lookup
        size_type count( const key_type& key ) const;

        template <typename K>
        size_type count( const K& key ) const;

        iterator find( const key_type& key );
        const_iterator find( const key_type& key ) const;

        template <typename K>
        iterator find( const K& key );

        template <typename K>
        const_iterator find( const K& key ) const;
```

(continues on next page)

(continued from previous page)

```
        bool contains( const key_type& key ) const;

        template <typename K>
        bool contains( const K& key ) const;

        std::pair<iterator, iterator> equal_range( const key_type& key );
        std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

        template <typename K>
        std::pair<iterator, iterator> equal_range( const K& key );

        template <typename K>
        std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

        // Bucket interface
        local_iterator unsafe_begin( size_type n );
        const_local_iterator unsafe_begin( size_type n ) const;
        const_local_iterator unsafe_cbegin( size_type n ) const;

        local_iterator unsafe_end( size_type n );
        const_local_iterator unsafe_end( size_type n ) const;
        const_local_iterator unsafe_cend( size_type n ) const;

        size_type unsafe_bucket_count() const;
        size_type unsafe_max_bucket_bount() const;

        size_type unsafe_bucket_size( size_type n ) const;

        size_type unsafe_bucket( const key_type& key ) const;

        // Hash policy
        float load_factor() const;

        float max_load_factor() const;
        void max_load_factor( float ml );

        void rehash( size_type count );

        void reserve( size_type count );

        // Observers
        hasher hash_function() const;
        key_equal key_eq() const;

        // Parallel iteration
        range_type range();
        const_range_type range() const;
    }; // class concurrent_unordered_multiset
} // namespace tbb
```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` must meet the `Hash` requirements from the [hash] ISO C++ Standard section.

- The type `KeyEqual` must meet the `BinaryPredicate` requirements from the [algorithms.general] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Description

`tbb::concurrent_unordered_multiset` is an unordered sequence, which elements are organized into buckets. The value of the hash function `Hash` for `Key` object determines the number of the bucket in which the corresponding element will be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, the member type `concurrent_unordered_multiset::key_equal` is defined as the value of this qualified-id. In this case, the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type.

Otherwise, the member type `concurrent_unordered_multiset::key_equal` is defined as the value of the template parameter `KeyEqual`.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_unordered_multiset();

explicit concurrent_unordered_multiset( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multiset`. The initial number of buckets is unspecified.

If provided, uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_multiset( size_type bucket_count,
                                        const hasher& hash = hasher(),
                                        const key_equal& equal = key_equal(),
                                        const allocator_type& alloc =␣
↪allocator_type() );

concurrent_unordered_multiset( size_type bucket_count, const allocator_type&␣
↪alloc );

concurrent_unordered_multiset( size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multiset` with `bucket_count` buckets.

If provided, uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
                               size_type bucket_count = /*implementation-
↪defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_
↪type() );

template <typename Inputiterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
                               size_type bucket_count, const allocator_type&␣
↪alloc );

template <typename InputIterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs the `concurrent_unordered_multiset`, which contains the elements from the half-open interval `[first, last)`.

If provided uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality, and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of `InputIterator` from the [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count = /*implementation-
↪defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_
↪type() );
```

Equivalent to `concurrent_unordered_multiset(init.begin(), init.end(), bucket_count, hash, equal, alloc)`.

```
concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count, const allocator_type&␣
↪alloc );
```

Equivalent to `concurrent_unordered_multiset(init.begin(), init.end(), bucket_count, alloc)`.

```
concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Equivalent to `concurrent_unordered_multiset(init.begin(), init.end(), bucket_count, hash, alloc)`.

Copying constructors

```
concurrent_unordered_multiset( const concurrent_unordered_multiset& other );

concurrent_unordered_multiset( const concurrent_unordered_multiset& other,
                               const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_unordered_multiset( concurrent_unordered_multiset&& other );

concurrent_unordered_multiset( concurrent_unordered_multiset&& other,
                               const allocator_type& alloc );
```

Constructs a `concurrent_unordered_multiset` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_unordered_multiset();
```

Destroys the `concurrent_unordered_multiset`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_unordered_multiset& operator=( const concurrent_unordered_
↪multiset& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_unordered_multiset& operator=( concurrent_unordered_multiset&&␣
↪other ) noexcept(/*See below*/);
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
↪equal::value &&
         std::is_nothrow_move_assignable<hasher>::value &&
         std::is_nothrow_move_assignable<key_equal>::value)
```

```
concurrent_unordered_multiset& operator=( std::initializer_list<value_type>␣
↪init );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_unordered_multiset::iterator` and `concurrent_unordered_multiset::const_itera` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value )
```

Inserts the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, const value_type& other )
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element.

```
std::pair<iterator, bool> insert( value_type&& value )
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, value_type&& other )
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last )
```

Inserts all items from the half-open interval `[first, last)` into the container.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init )
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh )
```

If the node handle `nh` is empty, does nothing.

Otherwise - inserts the node, owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, node_type&& nh )
```

If the node handle `nh` is empty, does nothing.

Otherwise - inserts the node, owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args )
```

Inserts an element ,constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always `true`.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args )
```

Inserts an element ,constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element.

Merging containers

```
template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&␣
↪source )

template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&␣
↪source )

template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>
↪& source )
```

(continues on next page)

(continued from previous page)

```
template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>
 ↪&& source )
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element that is equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the number of removed elements.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements which are equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

swap

```
void swap( concurrent_unordered_multiset& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Exceptions: `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
↪equal::value &&
         std::is_nothrow_swappable<hasher>::value &&
         std::is_nothrow_swappable<key_equal>::value
```

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements that are equivalent to `key`.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements equivalent to `key`, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements equivalent to `key`, it is unspecified which element should be found.

These overloads only participate in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if at least one element equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if at least one element equal to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element equivalent to `key`, `l` is an iterator to the element that follows the last element equivalent to `key`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element equivalent to `key`, `l` is an iterator to the element that follows the last element equivalent to `key`. Otherwise, `{end(), end()}`.

These overloads participate in overload resolution only if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

Bucket interface

The types `concurrent_unordered_multiset::local_iterator` and `concurrent_unordered_multiset::const_local_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ Standard section.

These iterators are used to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );

const_local_iterator unsafe_begin( size_type n ) const;

const_local_iterator unsafe_cbegin( size_type n ) const;
```

Returns: an iterator to the first element in the bucket number `n`.

```
local_iterator unsafe_end( size_type n );

const_local_iterator unsafe_end( size_type n ) const;

const_local_iterator unsafe_cend( size_type n ) const;
```

Returns: an iterator to the element that follows the last element in the bucket number `n`.

The number of buckets

```
size_type unsafe_bucket_count() const;
```

Returns: the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

Returns: the maximum number of buckets that container can hold.

Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

Returns: the number of elements in the bucket number `n`.

Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

Returns: the number of the bucket in which the element with the key `key` is stored.

Hash policy

Hash policy of `concurrent_unordered_multiset` manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

Load factor

```
float load_factor() const;
```

Returns: the average number of elements per bucket, which is `size()/unsafe_bucket_count()`.

```
float max_load_factor() const;
```

Returns: the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to `ml`.

Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to `n` and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value that is needed to store `n` elements.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

hash_function

```
hasher hash_function() const;
```

Returns: a copy of the hash function associated with `*this`.

key_eq

```
key_equal key_eq() const;
```

Returns: a copy of the key equality predicate associated with `*this`.

Parallel iteration

Member types `concurrent_unordered_multiset::range_type` and `concurrent_unordered_multiset::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_multiset::const_range_type` are of type `concurrent_unordered_multiset::const_iterator`, whereas the bounds for a `concurrent_unordered_multiset::range_type` are of type `concurrent_unordered_multiset::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary comparison and swap operations on `tbb::concurrent_unordered_multiset` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_unordered_multiset` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>&␣
↪rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>&␣
↪rhs );
```

Non-member swap

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs )␣
↪noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two objects of `concurrent_unordered_multiset` are equal if the following conditions are `true`:

- They contain an equal number of elements.
- Each group of elements with the same key in one container has the corresponding group of equivalent elements in the other container (not necessarily in the same order).

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>&␣
↪rhs );
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>&␣
↪lhs,
                 const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>&␣
↪rhs );
```

Equivalent to `!(lhs == rhs)`.

Returns: `true` if `lhs` is not equal to `rhs`, `false` otherwise.

Other

Deduction guides

Where possible, constructors of `concurrent_unordered_multiset` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_value_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_value_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_unordered_multiset( InputIterator, InputIterator,
                               map_size_type = /*implementation_defined*/,
                               Hash = Hash(), KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                 Hash, KeyEqual, Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_multiset( InputIterator, InputIterator,
                               map_size_type,
                               Allocator )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                 std::hash<iterator_value_t<InputIterator>>,
                                 std::equal_to<iterator_value_t<InputIterator>>,
```

(continues on next page)

(continued from previous page)

```
                                Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_multiset( InputIterator, InputIterator, Allocator )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                 std::hash<iterator_value_t<InputIterator>>,
                                 std::equal_to<iterator_key_t<InputIterator>>,
                                 Allocator>;

template <typename InputIterator,
          typename Hash,
          typename Allocator>
concurrent_unordered_multiset( InputIterator, InputIterator,
                               Hash, Allocator )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                 Hash,
                                 std::equal_to<iterator_value_t<InputIterator>>,
                                 Allocator>;

template <typename T,
          typename Hash = std::hash<Key>,
          typename KeyEqual = std::equal_to<Key>,
          typename Allocator = tbb_allocator<std::pair<Key, T>>>
concurrent_unordered_multiset( std::initializer_list<value_type>,
                               map_size_type = /*implementation-defined*/,
                               Hash = Hash(),
                               KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multiset<T,
                                 Hash,
                                 KeyEqual,
                                 Allocator>;

template <typename T,
          typename Allocator>
concurrent_unordered_multiset( std::initializer_list<value_type>,
                               map_size_type, Allocator )
-> concurrent_unordered_multiset<T,
                                 std::hash<Key>,
                                 std::equal_to<Key>,
                                 Allocator>;

template <typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_multiset( std::initializer_list<value_type>,
                           map_size_type, Hash, Allocator )
-> concurrent_unordered_multiset<T,
                                 Hash,
                                 std::equal_to<Key>,
                                 Allocator>;
```

where the type `map_size_type` refers to the `size_type` member type of the deduced `concurrent_unordered_multiset` and the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

Example

```
#include <tbb/concurrent_unordered_set.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<int> v;

    // Deduces s1 as concurrent_unordered_multiset<int>
    tbb::concurrent_unordered_multiset s1(v.begin(), v.end());

    // Deduces s2 as concurrent_unordered_multiset<int, CustomHasher>;
    tbb::concurrent_unordered_multiset s2(v.begin(), v.end(), CustomHasher{});
}
```

Ordered associative containers

concurrent_map

[containers.concurrent_map]

`tbb::concurrent_map` is a class template that represents a sorted associative container. It stores unique elements and supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure.

Class Template Synopsis

```
namespace tbb {

    template <typename Key,
              typename T,
              typename Compare = std::less<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>
    class concurrent_map {
    public:
        using key_type = Key;
        using mapped_type = T;
        using value_type = std::pair<const Key, T>;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using key_compare = Compare;
        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;
        using pointer = std::allocator_traits<Allocator>::pointer;
        using const_pointer = std::allocator_traits<Allocator>::const_pointer;
```

(continues on next page)

(continued from previous page)

```
        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        using node_type = <implementation-defined node handle>;

        using range_type = <implementation-defined range>;
        using const_range_type = <implementation-defined constant node handle>;

        class value_compare;

        // Construction, destruction, copying
        concurrent_map();
        explicit concurrent_map( const key_compare& comp,
                                 const allocator_type& alloc = allocator_type() );

        explicit concurrent_map( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_map( InputIterator first, InputIterator last,
                        const key_compare& comp = key_compare(),
                        const allocator_type& alloc = allocator_type() );

        template <typename InputIterator>
        concurrent_map( InputIterator first, InputIterator last,
                        const allocator_type& alloc );

        concurrent_map( std::initializer_list<value_type> init,
                        const key_compare& comp = key_compare(),
                        const allocator_type& alloc = allocator_type() );

        concurrent_map( std::initializer_list<value_type> init, const allocator_type&␣
↪alloc );

        concurrent_map( const concurrent_map& other );
        concurrent_map( const concurrent_map& other,
                        const allocator_type& alloc );

        concurrent_map( concurrent_map&& other );
        concurrent_map( concurrent_map&& other,
                        const allocator_type& alloc );

        ~concurrent_map();

        concurrent_map& operator=( const concurrent_map& other );
        concurrent_map& operator=( concurrent_map&& other );
        concurrent_map& operator=( std::initializer_list<value_type> init );

        allocator_type get_allocator() const;

        // Element access
        value_type& at( const key_type& key );
        const value_type& at( const key_type& key ) const;

        value_type& operator[]( const key_type& key );
        value_type& operator[]( key_type&& key );
```

(continues on next page)

(continued from previous page)

```
        // Iterators
        iterator begin();
        const_iterator begin() const;
        const_iterator cbegin() const;

        iterator end();
        const_iterator end() const;
        const_iterator cend() const;

        // Size and capacity
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

        // Concurrently safe modifiers
        std::pair<iterator, bool> insert( const value_type& value );

        iterator insert( const_iterator hint, const value_type& value );

        template <typename P>
        std::pair<iterator, bool> insert( P&& value );

        template <typename P>
        iterator insert( const_iterator hint, P&& value );

        std::pair<iterator, bool> insert( value_type&& value );

        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcCompare>
        void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

        template <typename SrcCompare>
        void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );

        // Concurrently unsafe modifiers
        void clear();
```

(continues on next page)

(continued from previous page)

```
        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );

        size_type unsafe_erase( const key_type& key );

        template <typename K>
        size_type unsafe_erase( const K& key );

        node_type unsafe_extract( const_iterator pos );
        node_type unsafe_extract( iterator pos );

        node_type unsafe_extract( const key_type& key );

        template <typename K>
        node_type unsafe_extract( const K& key );

        void swap( concurrent_map& other );

        // Lookup
        size_type count( const key_type& key );

        template <typename K>
        size_type count( const K& key );

        iterator find( const key_type& key );
        const_iterator find( const key_type& key ) const;

        template <typename K>
        iterator find( const K& key );

        template <typename K>
        const_iterator find( const K& key ) const;

        bool contains( const key_type& key ) const;

        template <typename K>
        bool contains( const K& key ) const;

        std::pair<iterator, iterator> equal_range( const key_type& key );
        std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

        template <typename K>
        std::pair<iterator, iterator>  equal_range( const K& key );
        std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

        iterator lower_bound( const key_type& key );
        const_iterator lower_bound( const key_type& key ) const;

        template <typename K>
        iterator lower_bound( const K& key );

        template <typename K>
        const_iterator lower_bound( const K& key ) const;
```

(continues on next page)

(continued from previous page)

```
        iterator upper_bound( const key_type& key );
        const_iterator upper_bound( const key_type& key ) const;

        template <typename K>
        iterator upper_bound( const K& key );

        template <typename K>
        const_iterator upper_bound( const K& key ) const;

        // Observers
        key_compare key_comp() const;

        value_compare value_comp() const;

        // Parallel iteration
        range_type range();
        const_range_type range() const;
    }; // class concurrent_map

} // namespace tbb
```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from the [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member classes

value_compare

`concurrent_map::value_compare` is a function object that is used to compare `concurrent_map::value_type` objects by comparing their first components.

Class Synopsis

```
namespace tbb {

    template <typename Key, typename T,
              typename Compare, typename Allocator>
    class concurrent_map<Key, T, Compare, Allocator>::value_compare {
    protected:
        key_compare comp;

        value_compare( key_compare c );

    public:
```

(continues on next page)

(continued from previous page)

```
        bool operator()( const value_type& lhs, const value_type& rhs )␣
↪const;
    }; // class value_compare

} // namespace tbb
```

Member objects

```
key_compare comp;
```

The key comparison function object.

Member functions

```
value_compare( key_compare c );
```

Constructs a `value_compare` with the stored key comparison function object `c`.

```
bool operator()( const value_type& lhs, const value_type& rhs ) const;
```

Compares `lhs.first` and `rhs.first` by calling the stored key comparison function `comp`.

Returns: `true` if first components of `lhs` and `rhs` are equal; `false`, otherwise.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_map();

explicit concurrent_map( const key_compare& comp,
                         const allocator_type& alloc = allocator_type() );

explicit concurrent_map( const allocator_type& alloc );
```

Constructs an empty `concurrent_map`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_map( InputIterator first, InputIterator last,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_map( InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_map`, which contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple elements with equal keys, it is unspecified which element would be inserted.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_map( std::initializer_list<value_type> init, const key_compare&␣
↪comp = key_compare(),
                const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_map(init.begin(), init.end(), comp, alloc)`.

```
concurrent_map( std::initializer_list<value_type> init,
                const allocator_type& alloc );
```

Equivalent to `concurrent_map(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_map( const concurrent_map& other );

concurrent_map( const concurrent_map& other, const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_map( concurrent_map&& other );

concurrent_map( concurrent_map&& other, const allocator_type& alloc );
```

Constructs a *concurrent_map* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_map();
```

Destroys the `concurrent_map`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_map& operator=( const concurrent_map& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_map& operator=( concurrent_map&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_map& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Element access

at

```
value_type& at( const key_type& key );

const value_type& at( const key_type& key ) const;
```

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

Throws: `std::out_of_range` exception if the element with the key equivalent to `key` is not present in the container.

operator[]

```
value_type& operator[]( const key_type& key );
```

If the element with the key equivalent to `key` is not present in the container, inserts a new element constructed in-place from `std::piecewise_construct, std::forward_as_tuple(key), std::tuple<>()`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

```
value_type& operator[]( key_type&& key );
```

If the element with the key equivalent to `key` is not present in the container, inserts a new element, constructed in-place from `std::piecewise_construct, std::forward_as_tuple(std::move(key)), std::tuple<>()`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Returns: a reference to `item.second`, where `item` is the element with the key equivalent to `key`.

Iterators

The types `concurrent_map::iterator` and `concurrent_map::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Attempts to insert the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value );
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value );
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple elements with equal keys, it is unspecified which element should be inserted.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with key equivalent to `nh.key()`. Boolean value is `true` if insertion took place; `false`, otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise - `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element or to an existing element with key equivalent to `nh.key()`.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element ,constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Merging containers

```
template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );
```

(continues on next page)

(continued from previous page)

```
template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );
```

Transfers those elements from `source` which keys do not exist in the container.

In case of merging with the container with multiple elements with equal keys, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` which follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: `1` if an element with the key equivalent to `key` exists; `0`, otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element with the key that is equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: `1` if an element with the key equivalent to `key` exists; `0`, otherwise.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

swap

```
void swap( concurrent_map& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers, and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements with the key equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements with the key that is equivalent to `key`.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element with the key that is equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

lower_bound

```
iterator lower_bound( const key_type& key );

const_iterator lower_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container with the key that is *not less* than `key`.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

Returns: an iterator to the first element in the container with the key that is *not less* than `key`.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

upper_bound

```
iterator upper_bound( const key_type& key );

const_iterator upper_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container with the key that compares *greater* than `key`.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

Returns: an iterator to the first element in the container with the key that compares `greater` than `key`.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if an element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if an element with the key that is equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

key_comp

```
key_compare key_comp() const;
```

Returns: a copy of the key comparison functor associated with `*this`.

value_comp

```
value_compare value_comp() const;
```

Returns: an object of the `value_compare` class that is used to compare `value_type` objects.

Parallel iteration

Member types `concurrent_map::range_type` and `concurrent_map::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_map::const_range_type` are of type `concurrent_map::const_iterator`, whereas the bounds for a `concurrent_map::range_type` are of type `concurrent_map::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary and lexicographical comparison and swap operations on `tbb::concurrent_map` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_map` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_map<Key, T, Compare, Allocator>& lhs,
           concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs );
```

(continues on next page)

(continued from previous page)

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_map<Key, T, Compare, Allocator>& lhs,
           concurrent_map<Key, T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two `tbb::concurrent_map` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Non-member lexicographical comparisons

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                 const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`.

Other

Deduction guides

Where possible, constructors of `concurrent_map` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Compare = std::less<iterator_key_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_map( InputIterator, InputIterator, Compare = Compare(), Allocator =␣
↪Allocator() )
-> concurrent_map<iterator_key_t<InputIterator>,
                  iterator_mapped_t<InputIterator>,
                  Compare,
                  Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_map( InputIterator, InputIterator, Allocator )
-> concurrent_map<iterator_key_t<InputIterator>,
                  iterator_mapped_t<InputIterator>,
                  std::less<iterator_key_t<InputIterator>>,
                  Allocator>;
```

(continues on next page)

(continued from previous page)

```
template <typename Key,
          typename T,
          typename Compare = std::less<Key>,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_map( std::initializer_list<std::pair<Key, T>>, Compare = Compare(),␣
↪Allocator = Allocator() )
-> concurrent_map<Key, T, Compare, Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_map( std::initializer_list<std::pair<Key, T>>, Allocator )
-> concurrent_map<Key, T, std::less<Key>, Allocator>;
```

where the type aliases `iterator_key_t`, `iterator_mapped_t`, `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits
↪<InputIterator>::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
↪<InputIterator>>,
                                         iterator_mapped_t<InputIterator>>;
```

Example

```
#include <tbb/concurrent_map.h>
#include <vector>

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces cm1 as concurrent_map<int, float>
    tbb::concurrent_map cm1(v.begin(), v.end());

    // Deduces cm2 as concurrent_map<int, float>
    tbb::concurrent_map cm2({std::pair(1, 2f), std::pair(2, 3f)});
}
```

concurrent_multimap

[containers.concurrent_multimap]

`tbb::concurrent_multimap` is a class template that represents a sorted associative container. It supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure. In this container, multiple elements with equal keys can be stored.

Class Template Synopsis

```
namespace tbb {

    template <typename Key,
              typename T,
              typename Compare = std::less<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>
    class concurrent_multimap {
    public:
        using key_type = Key;
        using mapped_type = T;
        using value_type = std::pair<const Key, T>;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using key_compare = Compare;
        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;
        using pointer = std::allocator_traits<Allocator>::pointer;
        using const_pointer = std::allocator_traits<Allocator>::const_pointer;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        using node_type = <implementation-defined node handle>;

        using range_type = <implementation-defined range>;
        using const_range_type = <implementation-defined constant node handle>;

        class value_compare;

        // Construction, destruction, copying
        concurrent_multimap();
        explicit concurrent_multimap( const key_compare& comp,
                                      const allocator_type& alloc = allocator_type()␣
↪);

        explicit concurrent_multimap( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_multimap( InputIterator first, InputIterator last,
                             const key_compare& comp = key_compare(),
                             const allocator_type& alloc = allocator_type() );
```

(continues on next page)

(continued from previous page)

```
        template <typename InputIterator>
        concurrent_multimap( InputIterator first, InputIterator last,
                             const allocator_type& alloc );

        concurrent_multimap( std::initializer_list<value_type> init,
                             const key_compare& comp = key_compare(),
                             const allocator_type& alloc = allocator_type() );

        concurrent_multimap( std::initializer_list<value_type> init, const allocator_
↪type& alloc );

        concurrent_multimap( const concurrent_multimap& other );
        concurrent_multimap( const concurrent_multimap& other,
                        const allocator_type& alloc );

        concurrent_multimap( concurrent_multimap&& other );
        concurrent_multimap( concurrent_multimap&& other,
                        const allocator_type& alloc );

        ~concurrent_multimap();

        concurrent_multimap& operator=( const concurrent_multimap& other );
        concurrent_multimap& operator=( concurrent_multimap&& other );
        concurrent_multimap& operator=( std::initializer_list<value_type> init );

        allocator_type get_allocator() const;

        // Iterators
        iterator begin();
        const_iterator begin() const;
        const_iterator cbegin() const;

        iterator end();
        const_iterator end() const;
        const_iterator cend() const;

        // Size and capacity
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

        // Concurrently safe modifiers
        std::pair<iterator, bool> insert( const value_type& value );

        iterator insert( const_iterator hint, const value_type& value );

        template <typename P>
        std::pair<iterator, bool> insert( P&& value );

        template <typename P>
        iterator insert( const_iterator hint, P&& value );

        std::pair<iterator, bool> insert( value_type&& value );

        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
```

(continues on next page)

(continued from previous page)

```
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcCompare>
        void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

        template <typename SrcCompare>
        void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );

        // Concurrently unsafe modifiers
        void clear();

        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );

        size_type unsafe_erase( const key_type& key );

        template <typename K>
        size_type unsafe_erase( const K& key );

        node_type unsafe_extract( const_iterator pos );
        node_type unsafe_extract( iterator pos );

        node_type unsafe_extract( const key_type& key );

        template <typename K>
        node_type unsafe_extract( const K& key );

        void swap( concurrent_multimap& other );

        // Lookup
        size_type count( const key_type& key );

        template <typename K>
        size_type count( const K& key );

        iterator find( const key_type& key );
        const_iterator find( const key_type& key ) const;
```

(continues on next page)

(continued from previous page)

```
        template <typename K>
        iterator find( const K& key );

        template <typename K>
        const_iterator find( const K& key ) const;

        bool contains( const key_type& key ) const;

        template <typename K>
        bool contains( const K& key ) const;

        std::pair<iterator, iterator> equal_range( const key_type& key );
        std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

        template <typename K>
        std::pair<iterator, iterator>  equal_range( const K& key );
        std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

        iterator lower_bound( const key_type& key );
        const_iterator lower_bound( const key_type& key ) const;

        template <typename K>
        iterator lower_bound( const K& key );

        template <typename K>
        const_iterator lower_bound( const K& key ) const;

        iterator upper_bound( const key_type& key );
        const_iterator upper_bound( const key_type& key ) const;

        template <typename K>
        iterator upper_bound( const K& key );

        template <typename K>
        const_iterator upper_bound( const K& key ) const;

        // Observers
        key_compare key_comp() const;

        value_compare value_comp() const;

        // Parallel iteration
        range_type range();
        const_range_type range() const;
    }; // class concurrent_multimap

} // namespace tbb
```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from the [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++

Standard section.

Member classes

value_compare

`concurrent_multimap::value_compare` is a function object that is used to compare `concurrent_multimap::value_type` objects by comparing their first components.

Class Synopsis

```
namespace tbb {

    template <typename Key, typename T,
              typename Compare, typename Allocator>
    class concurrent_multimap<Key, T, Compare, Allocator>::value_compare {
    protected:
        key_compare comp;

        value_compare( key_compare c );

    public:
        bool operator()( const value_type& lhs, const value_type& rhs )␣
↪const;
    }; // class value_compare

} // namespace tbb
```

Member objects

```
key_compare comp;
```

The key comparison function object.

Member functions

```
value_compare( key_compare c );
```

Constructs a `value_compare` with the stored key comparison function object `c`.

```
bool operator()( const value_type& lhs, const value_type& rhs ) const;
```

Compares `lhs.first` and `rhs.first` by calling the stored key comparison function `comp`.

Returns: `true` if first components of `lhs` and `rhs` are equal; `false`, otherwise.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_multimap();

explicit concurrent_multimap( const key_compare& comp,
                              const allocator_type& alloc = allocator_type()␣
↪);

explicit concurrent_multimap( const allocator_type& alloc );
```

Constructs an empty `concurrent_multimap`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_multimap( InputIterator first, InputIterator last,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_multimap( InputIterator first, InputIterator last,
                 const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_multimap`, which contains all elements from the half-open interval `[first, last)`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from `[input.iterators]` ISO C++ Standard section.

```
concurrent_multimap( std::initializer_list<value_type> init, const key_
↪compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_multimap(init.begin(), init.end(), comp, alloc)`.

```
concurrent_multimap( std::initializer_list<value_type> init,
                     const allocator_type& alloc );
```

Equivalent to `concurrent_multimap(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_multimap( const concurrent_multimap& other );

concurrent_multimap( const concurrent_multimap& other, const allocator_type&␣
↪alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_multimap( concurrent_multimap&& other );

concurrent_multimap( concurrent_multimap&& other, const allocator_type&␣
↪alloc );
```

Constructs a *concurrent_multimap* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_multimap();
```

Destroys the `concurrent_multimap`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_multimap& operator=( const concurrent_multimap& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_multimap& operator=( concurrent_multimap&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_multimap& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_multimap::iterator` and `concurrent_multimap::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty, `false` otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Inserts an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Inserts an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Inserts the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value );
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value );
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Inserts all items from the half-open interval `[first, last)` into the container.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from `[input.iterators]` the ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element.

Merging containers

```
template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` which follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes all element with the key equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed elements.

Returns: the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes all elements with the key that is equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed elements.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the number of removed elements.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If at least one element with the key equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If at least one element with the key that is equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key that is equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element with the key that is equivalent to `key` was not found.

swap

```
void swap( concurrent_multimap& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements with the key equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements with the key equivalent to `key`.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element with the key equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements with the key equivalent to `key`, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element with the key that is equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements with the key that is equivalent to `key`, it is unspecified which element should be found.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element with the key equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

lower_bound

```
iterator lower_bound( const key_type& key );

const_iterator lower_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container with the key that is *not less* than `key`.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

Returns: an iterator to the first element in the container with the key that is *not less* than `key`.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

upper_bound

```
iterator upper_bound( const key_type& key );

const_iterator upper_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container with the key that compares *greater* than `key`.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

Returns: an iterator to the first element in the container with the key that compares `greater` than `key`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element with the key equivalent to `key`, `l` is an iterator to the element that follows the last element with the key equivalent to `key`. Otherwise - `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if at least one element with the key equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element with the key that is equivalent to `key`, `l` is an iterator to the element that follows the last element with the key that is equivalent to `key`. Otherwise, `{end(), end()}`.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

key_comp

```
key_compare key_comp() const;
```

Returns: a copy of the key comparison functor associated with `*this`.

value_comp

```
value_compare value_comp() const;
```

Returns: an object of the `value_compare` class that is used to compare `value_type` objects.

Parallel iteration

Member types `concurrent_multimap::range_type` and `concurrent_multimap::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_multimap::const_range_type` are of type `concurrent_multimap::const_iterator`, whereas the bounds for a `concurrent_multimap::range_type` are of type `concurrent_multimap::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary and lexicographical comparison and swap operations on `tbb::concurrent_multimap` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_multimap` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_multimap<Key, T, Compare, Allocator>& lhs,
           concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_mutlimap<Key, T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_multimap<Key, T, Compare, Allocator>& lhs,
           concurrent_multimap<Key, T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two `tbb::concurrent_multimap` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Non-member lexicographical comparisons

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                 const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`.

Other

Deduction guides

Where possible, constructors of `concurrent_multimap` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Compare = std::less<iterator_key_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_multimap( InputIterator, InputIterator, Compare = Compare(), Allocator =␣
↪Allocator() )
-> concurrent_multimap<iterator_key_t<InputIterator>,
                       iterator_mapped_t<InputIterator>,
                       Compare,
                       Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_multimap( InputIterator, InputIterator, Allocator )
-> concurrent_multimap<iterator_key_t<InputIterator>,
                       iterator_mapped_t<InputIterator>,
                       std::less<iterator_key_t<InputIterator>>,
                       Allocator>;

template <typename Key,
          typename T,
          typename Compare = std::less<Key>,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_multimap( std::initializer_list<std::pair<Key, T>>, Compare = Compare(),␣
↪Allocator = Allocator() )
-> concurrent_multimap<Key, T, Compare, Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_multimap( std::initializer_list<std::pair<Key, T>>, Allocator )
-> concurrent_multimap<Key, T, std::less<Key>, Allocator>;
```

where the type aliases `iterator_key_t`, `iterator_mapped_t`, `iterator_alloc_value_t` are defined as follows:

```
template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits
↪<InputIterator>::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
↪<InputIterator>>,
                                         iterator_mapped_t<InputIterator>>;
```

Example

```
#include <tbb/concurrent_map.h>
#include <vector>

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces cm1 as concurrent_multimap<int, float>
    tbb::concurrent_multimap cm1(v.begin(), v.end());

    // Deduces cm2 as concurrent_multimap<int, float>
    tbb::concurrent_multimap cm2({std::pair(1, 2f), std::pair(2, 3f)});
}
```

concurrent_set

[containers.concurrent_set]

`tbb::concurrent_set` is a class template that represents an unordered sequence of unique elements. It supports concurrent insertion, lookup and traversal, but does not support concurrent erasure.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_set.h>

namespace tbb {

    template <typename T,
              typename Compare = std::less<T>,
              typename Allocator = tbb_allocator<T>>
    class concurrent_set {
    public:
        using key_type = T;
        using value_type = T;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using key_compare = Compare;
        using value_compare = Compare;

        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;
        using pointer = std::allocator_traits<Allocator>::pointer;
        using const_pointer = std::allocator_traits<Allocator>::const_pointer;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        using node_type = <implementation-defined node handle>;

        using range_type = <implementation-defined range>;
        using const_range_type = <implementation-defined constant node handle>;
```

(continues on next page)

(continued from previous page)

```
        // Construction, destruction, copying
        concurrent_set();
        explicit concurrent_set( const key_compare& comp,
                                 const allocator_type& alloc = allocator_type() );

        explicit concurrent_set( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_set( InputIterator first, InputIterator last,
                        const key_compare& comp = key_compare(),
                        const allocator_type& alloc = allocator_type() );

        template <typename InputIterator>
        concurrent_set( InputIterator first, InputIterator last,
                        const allocator_type& alloc );

        concurrent_set( std::initializer_list<value_type> init,
                        const key_compare& comp = key_compare(),
                        const allocator_type& alloc = allocator_type() );

        concurrent_set( std::initializer_list<value_type> init, const allocator_type&␣
↪alloc );

        concurrent_set( const concurrent_set& other );
        concurrent_set( const concurrent_set& other,
                        const allocator_type& alloc );

        concurrent_set( concurrent_set&& other );
        concurrent_set( concurrent_set&& other,
                        const allocator_type& alloc );

        ~concurrent_set();

        concurrent_set& operator=( const concurrent_set& other );
        concurrent_set& operator=( concurrent_set&& other );
        concurrent_set& operator=( std::initializer_list<value_type> init );

        allocator_type get_allocator() const;

        // Iterators
        iterator begin();
        const_iterator begin() const;
        const_iterator cbegin() const;

        iterator end();
        const_iterator end() const;
        const_iterator cend() const;

        // Size and capacity
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

        // Concurrently safe modifiers
        std::pair<iterator, bool> insert( const value_type& value );
```

(continues on next page)

(continued from previous page)

```
        iterator insert( const_iterator hint, const value_type& value );

        std::pair<iterator, bool> insert( value_type&& value );

        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );

        template <typename... Args>
        std::pair<iterator, bool> emplace( Args&&... args );

        template <typename... Args>
        iterator emplace_hint( const_iterator hint, Args&&... args );

        template <typename SrcCompare>
        void merge( concurrent_set<T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

        template <typename SrcCompare>
        void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

        template <typename SrcCompare>
        void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );

        // Concurrently unsafe modifiers
        void clear();

        iterator unsafe_erase( const_iterator pos );
        iterator unsafe_erase( iterator pos );

        iterator unsafe_erase( const_iterator first, const_iterator last );

        size_type unsafe_erase( const key_type& key );

        template <typename K>
        size_type unsafe_erase( const K& key );

        node_type unsafe_extract( const_iterator pos );
        node_type unsafe_extract( iterator pos );

        node_type unsafe_extract( const key_type& key );

        template <typename K>
        node_type unsafe_extract( const K& key );

        void swap( concurrent_set& other );

        // Lookup
        size_type count( const key_type& key );
```

(continues on next page)

(continued from previous page)

```
        template <typename K>
        size_type count( const K& key );

        iterator find( const key_type& key );
        const_iterator find( const key_type& key ) const;

        template <typename K>
        iterator find( const K& key );

        template <typename K>
        const_iterator find( const K& key ) const;

        bool contains( const key_type& key ) const;

        template <typename K>
        bool contains( const K& key ) const;

        std::pair<iterator, iterator> equal_range( const key_type& key );
        std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

        template <typename K>
        std::pair<iterator, iterator>  equal_range( const K& key );
        std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

        iterator lower_bound( const key_type& key );
        const_iterator lower_bound( const key_type& key ) const;

        template <typename K>
        iterator lower_bound( const K& key );

        template <typename K>
        const_iterator lower_bound( const K& key ) const;

        iterator upper_bound( const key_type& key );
        const_iterator upper_bound( const key_type& key ) const;

        template <typename K>
        iterator upper_bound( const K& key );

        template <typename K>
        const_iterator upper_bound( const K& key ) const;

        // Observers
        key_compare key_comp() const;

        value_compare value_comp() const;

        // Parallel iteration
        range_type range();
        const_range_type range() const;
    }; // class concurrent_set

} // namespace tbb
```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type`, must be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from the [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_set();

explicit concurrent_set( const key_compare& comp,
                         const allocator_type& alloc = allocator_type() );

explicit concurrent_set( const allocator_type& alloc );
```

Constructs an empty `concurrent_set`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_set( InputIterator first, InputIterator last,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_set( InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_set` that contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple equal elements, it is unspecified which element would be inserted.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_set( std::initializer_list<value_type> init, const key_compare&␣
↪comp = key_compare(),
                const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_set(init.begin(), init.end(), comp, alloc)`.

```
concurrent_set( std::initializer_list<value_type> init,
                const allocator_type& alloc );
```

Equivalent to `concurrent_set(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_set( const concurrent_set& other );

concurrent_set( const concurrent_set& other, const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_set( concurrent_set&& other );

concurrent_set( concurrent_set&& other, const allocator_type& alloc );
```

Constructs a *concurrent_set* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_set();
```

Destroys the `concurrent_set`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_set& operator=( const concurrent_set& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_set& operator=( concurrent_set&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_set& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_set::iterator` and `concurrent_set::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Attempts to insert the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple equal elements, it is unspecified which element should be inserted.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

If the insertion fails, node handle `nh` keeps ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool`, where `iterator` points to the inserted element or to an existing element equal to `nh.value()`. Boolean value is `true` if insertion took place; `false`, otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, attempts to insert the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` remains ownership of the node.

Otherwise, `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element or to an existing element equal to `nh.value()`.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool`, where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place; `false`, otherwise.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element or to an existing element with equal key.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Merging containers

```
template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );
```

Transfers those elements from `source` which keys do not exist in the container.

In case of merging with the container with multiple equal elements, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

Returns: `1` if an element equivalent to `key` exists; `0`, otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element that is equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: `1` if an element equivalent to `key` exists; `0`, otherwise.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

swap

```
void swap( concurrent_set& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements that are equivalent to `key`.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element equivalent to `key`, or `end()` if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element that is equivalent to `key`, or `end()` if no such element exists.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if an element equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if an element equivalent to `key` exists in the container; `false`, otherwise.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

lower_bound

```
iterator lower_bound( const key_type& key );

const_iterator lower_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container that is *not less* than `key`.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

Returns: an iterator to the first element in the container that is *not less* than `key`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

upper_bound

```
iterator upper_bound( const key_type& key );

const_iterator upper_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container that compares *greater* than `key`.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

Returns: an iterator to the first element in the container that compares `greater` than `key`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if an element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if an element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise, `{end(), end()}`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

key_comp

```
key_compare key_comp() const;
```

Returns: a copy of the key comparison functor associated with `*this`.

value_comp

```
value_compare value_comp() const;
```

Returns: an object of the `value_compare` class that is used to compare `value_type` objects.

Parallel iteration

Member types `concurrent_set::range_type` and `concurrent_set::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_set::const_range_type` are of type `concurrent_set::const_iterator`, whereas the bounds for a `concurrent_set::range_type` are of type `concurrent_set::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provide binary and lexicographical comparison and swap operations on `tbb::concurrent_set` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_set` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_set<T, Compare, Allocator>& lhs,
           concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_set<T, Compare, Allocator>& lhs,
                const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_set<T, Compare, Allocator>& lhs,
                const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_set<T, Compare, Allocator>& lhs,
           concurrent_set<T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two `tbb::concurrent_set` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Non-member lexicographical comparisons

```
template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_set<T, Compare, Allocator>& lhs,
                const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_set<T, Compare, Allocator>& lhs,
                const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_set<T, Compare, Allocator>& lhs,
                 const concurrent_set<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`.

Other

Deduction guides

Where possible, constructors of `concurrent_set` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Compare = std::less<iterator_value_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_set( InputIterator, InputIterator, Compare = Compare(), Allocator =␣
↪Allocator() )
-> concurrent_set<iterator_value_t<InputIterator>,
                  Compare,
                  Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_set( InputIterator, InputIterator, Allocator )
-> concurrent_set<iterator_value_t<InputIterator>,
                  std::less<iterator_key_t<InputIterator>>,
                  Allocator>;

template <typename T,
          typename Compare = std::less<T>,
          typename Allocator = tbb_allocator<T>>
concurrent_set( std::initializer_list<T>, Compare = Compare(), Allocator =␣
↪Allocator() )
-> concurrent_set<T, Compare, Allocator>;

template <typename T,
          typename Allocator>
concurrent_set( std::initializer_list<T>, Allocator )
-> concurrent_set<T, std::less<Key>, Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

Example

```
#include <tbb/concurrent_set.h>
#include <vector>

int main() {
    std::vector<int> v;

    // Deduces cs1 as concurrent_set<int>
    tbb::concurrent_set cs1(v.begin(), v.end());

    // Deduces cs2 as concurrent_set<int>
    tbb::concurrent_set cs2({1, 2, 3});
}
```

concurrent_multiset

[containers.concurrent_multiset]

`tbb::concurrent_multiset` is a class template that represents an unordered sequence of unique elements. It supports concurrent insertion, lookup, and traversal, but does not support concurrent erasure. In this container, multiple equivalent elements can be stored.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_set.h>

namespace tbb {

    template <typename T,
              typename Compare = std::less<T>,
              typename Allocator = tbb_allocator<T>>
    class concurrent_multiset {
    public:
        using key_type = T;
        using value_type = T;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using key_compare = Compare;
        using value_compare = Compare;

        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;
        using pointer = std::allocator_traits<Allocator>::pointer;
        using const_pointer = std::allocator_traits<Allocator>::const_pointer;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        using node_type = <implementation-defined node handle>;

        using range_type = <implementation-defined range>;
        using const_range_type = <implementation-defined constant node handle>;

        // Construction, destruction, copying
        concurrent_multiset();
        explicit concurrent_multiset( const key_compare& comp,
                                      const allocator_type& alloc = allocator_type()␣
↪);

        explicit concurrent_multiset( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_multiset( InputIterator first, InputIterator last,
                             const key_compare& comp = key_compare(),
                             const allocator_type& alloc = allocator_type() );
```

(continues on next page)

(continued from previous page)

```
        template <typename InputIterator>
        concurrent_multiset( InputIterator first, InputIterator last,
                             const allocator_type& alloc );

        concurrent_multiset( std::initializer_list<value_type> init,
                             const key_compare& comp = key_compare(),
                             const allocator_type& alloc = allocator_type() );

        concurrent_multiset( std::initializer_list<value_type> init, const allocator_
↪type& alloc );

        concurrent_multiset( const concurrent_multiset& other );
        concurrent_multiset( const concurrent_multiset& other,
                             const allocator_type& alloc );

        concurrent_multiset( concurrent_multiset&& other );
        concurrent_multiset( concurrent_multiset&& other,
                             const allocator_type& alloc );

        ~concurrent_multiset();

        concurrent_multiset& operator=( const concurrent_multiset& other );
        concurrent_multiset& operator=( concurrent_multiset&& other );
        concurrent_multiset& operator=( std::initializer_list<value_type> init );

        allocator_type get_allocator() const;

        // Iterators
        iterator begin();
        const_iterator begin() const;
        const_iterator cbegin() const;

        iterator end();
        const_iterator end() const;
        const_iterator cend() const;

        // Size and capacity
        bool empty() const;
        size_type size() const;
        size_type max_size() const;

        // Concurrently safe modifiers
        std::pair<iterator, bool> insert( const value_type& value );

        iterator insert( const_iterator hint, const value_type& value );

        std::pair<iterator, bool> insert( value_type&& value );

        iterator insert( const_iterator hint, value_type&& value );

        template <typename InputIterator>
        void insert( InputIterator first, InputIterator last );

        void insert( std::initializer_list<value_type> init );

        std::pair<iterator, bool> insert( node_type&& nh );
        iterator insert( const_iterator hint, node_type&& nh );
```

(continues on next page)

(continued from previous page)

```
    template <typename... Args>
    std::pair<iterator, bool> emplace( Args&&... args );

    template <typename... Args>
    iterator emplace_hint( const_iterator hint, Args&&... args );

    template <typename SrcCompare>
    void merge( concurrent_set<T, SrcCompare, Allocator>& source );

    template <typename SrcCompare>
    void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

    template <typename SrcCompare>
    void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

    template <typename SrcCompare>
    void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );

    // Concurrently unsafe modifiers
    void clear();

    iterator unsafe_erase( const_iterator pos );
    iterator unsafe_erase( iterator pos );

    iterator unsafe_erase( const_iterator first, const_iterator last );

    size_type unsafe_erase( const key_type& key );

    template <typename K>
    size_type unsafe_erase( const K& key );

    node_type unsafe_extract( const_iterator pos );
    node_type unsafe_extract( iterator pos );

    node_type unsafe_extract( const key_type& key );

    template <typename K>
    node_type unsafe_extract( const K& key );

    void swap( concurrent_multiset& other );

    // Lookup
    size_type count( const key_type& key );

    template <typename K>
    size_type count( const K& key );

    iterator find( const key_type& key );
    const_iterator find( const key_type& key ) const;

    template <typename K>
    iterator find( const K& key );

    template <typename K>
    const_iterator find( const K& key ) const;
```

(continues on next page)

(continued from previous page)

```
        bool contains( const key_type& key ) const;

        template <typename K>
        bool contains( const K& key ) const;

        std::pair<iterator, iterator> equal_range( const key_type& key );
        std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;

        template <typename K>
        std::pair<iterator, iterator>  equal_range( const K& key );
        std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

        iterator lower_bound( const key_type& key );
        const_iterator lower_bound( const key_type& key ) const;

        template <typename K>
        iterator lower_bound( const K& key );

        template <typename K>
        const_iterator lower_bound( const K& key ) const;

        iterator upper_bound( const key_type& key );
        const_iterator upper_bound( const key_type& key ) const;

        template <typename K>
        iterator upper_bound( const K& key );

        template <typename K>
        const_iterator upper_bound( const K& key ) const;

        // Observers
        key_compare key_comp() const;

        value_compare value_comp() const;

        // Parallel iteration
        range_type range();
        const_range_type range() const;
    }; // class concurrent_multiset

} // namespace tbb
```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)`, where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type`, should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` must meet the `Compare` requirements from the [alg.sorting] ISO C++ Standard section.
- The type `Allocator` must meet the `Allocator` requirements from the [allocator.requirements] ISO C++ Standard section.

Member functions

Construction, destruction, copying

Empty container constructors

```
concurrent_multiset();

explicit concurrent_multiset( const key_compare& comp,
                              const allocator_type& alloc = allocator_type()␣
↪);

explicit concurrent_multiset( const allocator_type& alloc );
```

Constructs an empty `concurrent_multiset`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_multiset( InputIterator first, InputIterator last,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_multiset( InputIterator first, InputIterator last,
                     const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_multiset`, which contains all elements from the half-open interval `[first, last)`.

If provided, uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
concurrent_multiset( std::initializer_list<value_type> init, const key_
↪compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_multiset(init.begin(), init.end(), comp, alloc)`.

```
concurrent_multiset( std::initializer_list<value_type> init,
                     const allocator_type& alloc );
```

Equivalent to `concurrent_multiset(init.begin(), init.end(), alloc)`.

Copying constructors

```
concurrent_multiset( const concurrent_multiset& other );

concurrent_multiset( const concurrent_multiset& other, const allocator_type&␣
↪alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Moving constructors

```
concurrent_multiset( concurrent_multiset&& other );

concurrent_multiset( concurrent_multiset&& other, const allocator_type&␣
↪alloc );
```

Constructs a *concurrent_multiset* with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

Destructor

```
~concurrent_multiset();
```

Destroys the `concurrent_multiset`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

Assignment operators

```
concurrent_multiset& operator=( const concurrent_multiset& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_multiset& operator=( concurrent_multiset&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move-assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_as` is `true`.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

Returns: a reference to `*this`.

```
concurrent_multiset& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

Returns: a reference to `*this`.

Iterators

The types `concurrent_multiset::iterator` and `concurrent_multiset::const_iterator` meet the requirements of `ForwardIterator` from the [forward.iterators] ISO C++ standard section.

begin and cbegin

```
iterator begin();

const_iterator begin() const;

const_iterator cbegin() const;
```

Returns: an iterator to the first element in the container.

end and cend

```
iterator end();

const_iterator end() const;

const_iterator cend() const;
```

Returns: an iterator to the element that follows the last element in the container.

Size and capacity

empty

```
bool empty() const;
```

Returns: `true` if the container is empty; `false`, otherwise.

The result may differ from the actual container state in case of pending concurrent insertions.

size

```
size_type size() const;
```

Returns: the number of elements in the container.

The result may differ from the actual container size in case of pending concurrent insertions.

max_size

```
size_type max_size() const;
```

Returns: the maximum number of elements that container can hold.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Inserts the value `value` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `CopyInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `MoveInsertable` requirements from the [container.requirements] ISO C++ Standard section.

Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Inserts all items from the half-open interval `[first, last)` into the container.

Requirements: the type `InputIterator` must meet the requirements of *InputIterator* from the `[input.iterators]` ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise, inserts the node owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

Returns: an iterator pointing to the inserted element.

Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Inserts an element, constructed in-place from `args` into the container.

Returns: `std::pair<iterator, bool>`, where `iterator` points to the inserted element. Boolean value is always `true`.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Inserts an element constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

Returns: an `iterator` to the inserted element.

Requirements: the type `value_type` must meet the `EmplaceConstructible` requirements from the [container.requirements] ISO C++ section.

Merging containers

```
template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

Clearing

```
void clear();
```

Removes all elements from the container.

Erasing elements

```
iterator unsafe_erase( const_iterator pos );

iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

Returns: `iterator` that follows the removed element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes all elements equivalent to `key` if they exist in the container.

Invalidates all iterators and references to the removed element.

Returns: the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes all elements that are equivalent to `key` if they exist in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the number of removed elements.

Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

Returns: `iterator` that follows the last removed element.

Requirements: the range `[first, last)` must be a valid subrange in `*this`.

Extracting nodes

```
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element.

Requirements: the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element that is equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements equivalent to `key`, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

Returns: the node handle that owns the extracted element or an empty node handle if an element equivalent to `key` was not found.

swap

```
void swap( concurrent_multiset& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise, if `get_allocator() != other.get_allocator()`, the behavior is undefined.

Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

count

```
size_type count( const key_type& key );
```

Returns: the number of elements equivalent to `key`.

```
template <typename K>
size_type count( const K& key );
```

Returns: the number of elements that are equivalent to `key`.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

find

```
iterator find( const key_type& key );

const_iterator find( const key_type& key ) const;
```

Returns: an iterator to the element equivalent to `key` or `end()` if no such element exists.

If there are multiple elements equivalent to `key`, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

Returns: an iterator to the element that is equivalent to `key`, or `end()` if no such element exists.

If there are multiple elements that are equivalent to `key`, it is unspecified which element should be found.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

contains

```
bool contains( const key_type& key ) const;
```

Returns: `true` if at least one element equivalent to `key` exists in the container; `false`, otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

Returns: `true` if at least one element that is equivalent to `key` exists in the container; `false`, otherwise.

This overload participates in overload resolution only if qualified-id `key_compare::is_transparent` is valid and denotes a type.

lower_bound

```
iterator lower_bound( const key_type& key );

const_iterator lower_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container that is *not less* than `key`.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

Returns: an iterator to the first element in the container that is *not less* than `key`.

These overloads only participate in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

upper_bound

```
iterator upper_bound( const key_type& key );

const_iterator upper_bound( const key_type& key ) const;
```

Returns: an iterator to the first element in the container that compares *greater* than `key`.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

Returns: an iterator to the first element in the container that compares `greater` than `key`.

These overloads participate in overload resolution only if qualified-id `key_compare::is_transparent` is valid and denotes a type.

equal_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )␣
↪const;
```

Returns: if at least one element equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element equivalent to `key`, `l` is an iterator to the element that follows the last element equivalent to `key`. Otherwise, `{end(), end()}`.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

Returns: if at least one element that is equivalent to `key` exists, a pair of iterators `{f, l}`, where `f` is an iterator to the first element that is equivalent to `key`, `l` is an iterator to the element that follows the last element that is equivalent to `key`. Otherwise, `{end(), end()}`.

These overloads participate in overload resolution only if qualified-id `key_compare::is_transparent` is valid and denotes a type.

Observers

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator associated with `*this`.

key_comp

```
key_compare key_comp() const;
```

Returns: a copy of the key comparison functor associated with `*this`.

value_comp

```
value_compare value_comp() const;
```

Returns: an object of the `value_compare` class that is used to compare `value_type` objects.

Parallel iteration

Member types `concurrent_multiset::range_type` and `concurrent_multiset::const_range_type` meet the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_multiset::const_range_type` are of type `concurrent_multiset::const_iterator`, whereas the bounds for a `concurrent_multiset::range_type` are of type `concurrent_multiset::iterator`.

range member function

```
range_type range();

const_range_type range() const;
```

Returns: a range object representing all elements in the container.

Non-member functions

These functions provides binary and lexicographical comparison and swap operations on `tbb::concurrent_multiset` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_multiset` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_multiset<T, Compare, Allocator>& lhs,
           concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multiset<T, Compare, Allocator>& lhs,
                const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multiset<T, Compare, Allocator>& lhs,
                const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs );
```

Non-member swap

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_multiset<T, Compare, Allocator>& lhs,
           concurrent_multiset<T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

Non-member binary comparisons

Two `tbb::concurrent_multiset` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is equal to `rhs`; `false`, otherwise.

```
template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is not equal to `rhs`; `false`, otherwise.

Non-member lexicographical comparisons

```
template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multiset<T, Compare, Allocator>& lhs,
                const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multiset<T, Compare, Allocator>& lhs,
                const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                 const concurrent_multiset<T, Compare, Allocator>& rhs )
```

Returns: `true` if `lhs` is lexicographically *greater or equal* than `rhs`.

Other

Deduction guides

Where possible, constructors of `concurrent_multiset` support class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Compare = std::less<iterator_value_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_multiset( InputIterator, InputIterator, Compare = Compare(), Allocator =␣
↪Allocator() )
-> concurrent_multiset<iterator_value_t<InputIterator>,
                       Compare,
                       Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_multiset( InputIterator, InputIterator, Allocator )
-> concurrent_multiset<iterator_value_t<InputIterator>,
                       std::less<iterator_key_t<InputIterator>>,
                       Allocator>;
```

(continues on next page)

(continued from previous page)

```
template <typename T,
          typename Compare = std::less<T>,
          typename Allocator = tbb_allocator<T>>
concurrent_multiset( std::initializer_list<T>, Compare = Compare(), Allocator =␣
↪Allocator() )
-> concurrent_multiset<T, Compare, Allocator>;

template <typename T,
          typename Allocator>
concurrent_multiset( std::initializer_list<T>, Allocator )
-> concurrent_multiset<T, std::less<Key>, Allocator>;
```

Where the type alias `iterator_value_t` is defined as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

Example

```
#include <tbb/concurrent_set.h>
#include <vector>

int main() {
    std::vector<int> v;

    // Deduces cs1 as concurrent_multiset<int>
    tbb::concurrent_multiset cs1(v.begin(), v.end());

    // Deduces cs2 as concurrent_multiset<int>
    tbb::concurrent_multiset cs2({1, 2, 3});
}
```

Auxiliary classes

tbb_hash_compare

[containers.tbb_hash_compare]

`tbb::tbb_hash_compare` is a class template for hash support. Use it with the `tbb::concurrent_hash_map` associative container to calculate hash codes and compare keys for equality.

`tbb_hash_compare` meets the *HashCompare requirements*.

Class Template Synopsis

```
// Defined in header <tbb/concurrent_hash_map.h>

namespace tbb {

    template <typename Key>
    class tbb_hash_compare {
        static std::size_t hash( const Key& k );
```

(continues on next page)

(continued from previous page)

```
        static bool equal( const Key& k1, const Key& k2 );
    }; // class tbb_hash_compare

} // namespace tbb
```

Member functions

```
static std::size_t hash( const Key& k );
```

Returns: a hash code for a key `k`.

```
static bool equal( const Key& k1, const Key& k2 );
```

Equivalent to `k1 == k2`.

Returns: `true` if the keys are equal; `false`, otherwise.

Node handles

[containers.node_handles]

Concurrent associative containers (`concurrent_map`, `concurrent_multimap`, `concurrent_set`, `concurrent_multiset`, `concurrent_unordered_map`, `concurrent_unordered_multimap`, `concurrent_unordered_set`, and `concurrent_unordered_multiset`) store elements in individually allocated, connected nodes. These containers support data transfer between containers with compatible node types by changing the connections without copying or moving the actual data.

Class synopsis

```
class node-handle { // Exposition-only name
public:
    using key_type = <container-specific>;    // Only for maps
    using mapped_type = <container-specific>; // Only for maps
    using value_type = <container-specific>;  // Only for sets
    using allocator_type = <container-specific>;

    node-handle();
    node-handle( node-handle&& other );

    ~node-handle();

    node-handle& operator=( node-handle&& other );

    void swap( node-handle& nh );

    bool empty() const;
    explicit operator bool() const;

    key_type& key() const;       // Only for maps
    mapped_type& mapped() const; // Only for maps
```

(continues on next page)

(continued from previous page)

```
    value_type& value() const;   // Only for sets

    allocator_type get_allocator() const;
};
```

A `node handle` is a container-specific move-only nested type (exposed as *container::node_type*) that represents a node outside of any container instance. It allows reading and modifying the data stored in the node, and inserting the node into a compatible container instance. The following containers have compatible node types and may exchange nodes:

- `concurrent_map` and `concurrent_multimap` with the same `key_type`, `mapped_type` and `allocator_type`.
- `concurrent_set` and `concurrent_multiset` with the same `value_type` and `allocator_type`.
- `concurrent_unordered_map` and `concurrent_unordered_multimap` with the same `key_type`, `mapped_type` and `allocator_type`.
- `concurrent_unordered_set` and `concurrent_unordered_multiset` with the same `value_type` and `allocator_type`.

Default or moved-from node handles are *empty* and do not represent a valid node. A non-empty node handle is typically created when a node is extracted out of a container, for example, with the `unsafe_extract` method. It stores the node along with a copy of the container's allocator. Upon assignment or destruction a non-empty node handle destroys the stored data and deallocates the node.

Member functions

Constructors

```
node-handle();
```

Constructs an empty node handle.

```
node-handle( node-handle&& other );
```

Constructs a node handle that takes ownership of the node from `other`.

`other` is left in an empty state.

Assignment

```
node-handle& operator=( node-handle&& other );
```

Transfers ownership of the node from `other` to `*this`. If `*this` was not empty before transferring, destroys and deallocates the stored node.

Move assigns the stored allocator if `std::allocator_traits<allocator_type>::propagate_on_container_m` is `true`.

`other` is left in an empty state.

Destructor

```
~node-handle();
```

Destroys the node handle. If it is not empty, destroys and deallocates the owned node.

Swap

```
void swap( node-handle& other )
```

Exchanges the nodes owned by `*this` and `other`.

Swaps the stored allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap:` is `true`.

State

```
bool empty() const;
```

Returns: `true` if the node handle is empty, `false` otherwise.

```
explicit operator bool() const;
```

Equivalent to `!empty()`.

Access to the stored element

```
key_type& key() const;
```

Available only for map node handles.

Returns: a reference to the key of the element stored in the owned node.

The behavior is undefined if the node handle is empty.

```
mapped_type& mapped() const;
```

Available only for map node handles.

Returns: a reference to the value of the element stored in the owned node.

The behavior is undefined if the node handle is empty.

```
value_type& value() const;
```

Available only for set node handles.

Returns: a reference to the element stored in the owned node.

The behavior is undefined if the node handle is empty.

get_allocator

```
allocator_type get_allocator() const;
```

Returns: a copy of the allocator stored in the node handle.

The behavior is undefined if the node handle is empty.

9.2.6 Thread Local Storage

[thread_local_storage]

oneAPI Threading Building Blocks provides class templates for thread local storage (TLS). Each provides a thread-local element per thread and lazily creates elements on demand.

combinable

[tls.combinable]

A class template for holding thread-local values during a parallel computation that will be merged into a final value.

A `combinable` provides each thread with its own instance of type `T`.

```
// Defined in header <tbb/combinable.h>

namespace tbb {
    template <typename T>
    class combinable {
    public:
        combinable();

        combinable(const combinable& other);
        combinable(combinable&& other);

        template <typename FInit>
        explicit combinable(FInit finit);

        ~combinable();

        combinable& operator=( const combinable& other);
        combinable& operator=( combinable&& other);

        void clear();

        T& local();
        T& local(bool & exists);

        template<typename BinaryFunc> T combine(BinaryFunc f);
        template<typename UnaryFunc> void combine_each(UnaryFunc f);
    };
}
```

Member functions

combinable()
Constructs `combinable` such that thread-local instances of `T` will be default-constructed.

template<typename **FInit**>
explicit combinable(*FInit finit*)
Constructs `combinable` such that thread-local elements will be created by copying the result of *finit()*.

> **Caution:** The expression *finit()* must be safe to evaluate concurrently by multiple threads. It is evaluated each time a new thread-local element is created.

combinable(**const** *combinable* &*other*)
Constructs a copy of *other*, so that it has copies of each element in *other* with the same thread mapping.

combinable(*combinable* &&*other*)
Constructs `combinable` by moving the content of *other* intact. *other* is left in an unspecified state but can be safely destroyed.

~combinable()
Destroys all elements in `*this`.

combinable &**operator=**(**const** combinable &*other*)
Sets `*this` to be a copy of *other*. Returns a reference to `*this`.

combinable &**operator=**(combinable &&*other*)
Moves the content of *other* to `*this` intact. *other* is left in an unspecified state but can be safely destroyed. Returns a reference to `*this`.

void **clear**()
Removes all elements from `*this`.

T &**local**()
If an element does not exist for the current thread, creates it.

Returns: Reference to thread-local element.

T &**local**(bool &*exists*)
Similar to `local()`, except that *exists* is set to true if an element was already present for the current thread; false, otherwise.

Returns: Reference to thread-local element.

template<typename **BinaryFunc**>
T **combine**(*BinaryFunc f*)
Requires: A `BinaryFunc` must meet the *Function Objects* requirements from the [function.objects] ISO C++ Standard section. Specifically, the type should be an associative binary functor with the signature `T BinaryFunc(T,T)` or `T BinaryFunc(const T&,const T&)`. A `T` type must be the same as a corresponding template parameter for the `combinable` object.

Effects: Computes a reduction over all elements using binary functor *f*. All evaluations of *f* are done sequentially in the calling thread. If there are no elements, creates the result using the same rules as for creating a new element.

Returns: Result of the reduction.

template<typename **UnaryFunc**>
void **combine_each**(*UnaryFunc f*)
Requires: An `UnaryFunc` must meet the *Function Objects* requirements from the [function.objects] ISO

C++ Standard section. Specifically, the type should be an unary functor with the one of the signatures: `void UnaryFunc(T)`, `void UnaryFunc(T&)`, or `void UnaryFunc(const T&)` A `T` type must be the same as a corresponding template parameter for the `enumerable_thread_specific` object.

Effects: Evaluates *f(x)* for each thread-local element *x* in `*this`. All evaluations are done sequentially in the calling thread.

Note: Methods of `class combinable` are not thread-safe, except for `local`.

---

enumerable_thread_specific

[tls.enumerable_thread_specific]

A class template for thread local storage (TLS).

```
// Defined in header <tbb/enumerable_thread_specific.h>

namespace tbb {

    enum ets_key_usage_type {
        ets_key_per_instance,
        ets_no_key,
        ets_suspend_aware
    };

    template <typename T,
        typename Allocator=cache_aligned_allocator<T>,
        ets_key_usage_type ETS_key_type=ets_no_key >
    class enumerable_thread_specific {
    public:
        // Basic types
        using value_type = T;
        using reference = T&;
        using const_reference = const T&;
        using pointer = T*;
        using size_type = /* implementation-defined */;
        using difference_type = /* implementation-defined */;
        using allocator_type = Allocator;

        // Iterator types
        using iterator = /* implementation-defined */;
        using const_iterator = /* implementation-defined */;

        // Parallel range types
        using range_type = /* implementation-defined */;
        using const_range_type = /* implementation-defined */;

        // Construction
        enumerable_thread_specific();
        template <typename Finit>
        explicit enumerable_thread_specific( Finit finit );
        explicit enumerable_thread_specific( const T& exemplar );
        explicit enumerable_thread_specific( T&& exemplar );
        template <typename... Args>
        enumerable_thread_specific( Args&&... args );
```

(continues on next page)

(continued from previous page)

```
        // Destruction
        ~enumerable_thread_specific();

        // Copy constructors
        enumerable_thread_specific( const enumerable_thread_specific& other);
        template<typename Alloc, ets_key_usage_type Cachetype>
        enumerable_thread_specific( const enumerable_thread_specific<T, Alloc,␣
↪Cachetype>& other);
        // Copy assignments
        enumerable_thread_specific& operator=( const enumerable_thread_specific&␣
↪other );
        template<typename Alloc, ets_key_usage_type Cachetype>
        enumerable_thread_specific& operator=( const enumerable_thread_specific<T,␣
↪Alloc, Cachetype>& other );

        // Move constructors
        enumerable_thread_specific( enumerable_thread_specific&& other);
        template<typename Alloc, ets_key_usage_type Cachetype>
        enumerable_thread_specific( enumerable_thread_specific<T, Alloc, Cachetype>&&␣
↪other);
        // Move assignments
        enumerable_thread_specific& operator=( enumerable_thread_specific&& other );
        template<typename Alloc, ets_key_usage_type Cachetype>
        enumerable_thread_specific& operator=( enumerable_thread_specific<T, Alloc,␣
↪Cachetype>&& other );

        // Other whole container operations
        void clear();

        // Concurrent operations
        reference local();
        reference local( bool& exists );
        size_type size() const;
        bool empty() const;

        // Combining
        template<typename BinaryFunc> T combine( BinaryFunc f );
        template<typename UnaryFunc> void combine_each( UnaryFunc f );

        // Parallel iteration
        range_type range( size_t grainsize=1 );
        const_range_type range( size_t grainsize=1 ) const;

        // Iterators
        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;
    };

} // namespace tbb
```

A class template `enumerable_thread_specific` provides TLS for elements of type `T`. A class template `enumerable_thread_specific` acts as a container by providing iterators and ranges across all of the thread-local elements.

The thread-local elements are created lazily. A freshly constructed `enumerable_thread_specific` has no

elements. When a thread requests access to an `enumerable_thread_specific`, it creates an element corresponding to that thread. The number of elements is equal to the number of distinct threads that have accessed the `enumerable_thread_specific` and not necessarily the number of threads in use by the application. Clearing an `enumerable_thread_specific` removes all its elements.

Use the `ETS_key_usage_type` parameter type to select an underlying implementation.

Caution: `enumerable_thread_specific` uses the OS-specific value returned by `std::this_thread::get_id()` to identify threads. This value is not guaranteed to be unique except for the life of the thread. A newly created thread may get an OS-specific ID equal to that of an already destroyed thread. The number of elements of the `enumerable_thread_specific` may therefore be less than the number of actual distinct threads that have called `local()`, and the element returned by the first reference by a thread to the `enumerable_thread_specific` may not be newly-constructed.

Member functions

Construction, destruction, copying

Empty container constructors

```
enumerable_thread_specific();
```

Constructs an `enumerable_thread_specific` where each thread-local element will be default-constructed.

```
template<typename Finit> explicit enumerable_thread_specific( Finit finit );
```

Constructs an `enumerable_thread_specific` such that any thread-local element will be created by copying the result of `finit()`.

Note: The expression `finit()` must be safe to evaluate concurrently by multiple threads. It is evaluated each time a thread-local element is created.

```
explicit enumerable_thread_specific( const T& exemplar );
```

Constructs an `enumerable_thread_specific` where each thread-local element will be copy-constructed from `exemplar`.

```
explicit enumerable_thread_specific( T&& exemplar );
```

Constructs an `enumerable_thread_specific` object, move constructor of `T` can be used to store `exemplar` internally; however, thread-local elements are always copy-constructed.

```
template <typename... Args> enumerable_thread_specific( Args&&... args );
```

Constructs `enumerable_thread_specific` such that any thread-local element will be constructed by invoking `T(args...)`.

Note: This constructor does not participate in overload resolution if the type of the first argument in `args...` is `T`, or `enumerable_thread_specific<T>`, or `foo()` is a valid expression for a value `foo` of that type.

Copying constructors

```
enumerable_thread_specific ( const enumerable_thread_specific& other );

template<typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific(␣
↪const enumerable_thread_specific <T, Alloc, Cachetype>& other );
```

Constructs an `enumerable_thread_specific` as a copy of `other`. The values are copy-constructed from the values in `other` and have same thread correspondence.

Moving constructors

```
enumerable_thread_specific ( enumerable_thread_specific&& other )
```

Constructs an `enumerable_thread_specific` by moving the content of `other` intact. `other` is left in an unspecified state, but can be safely destroyed.

```
template<typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific(␣
↪enumerable_thread_specific <T, Alloc, Cachetype>&& other )
```

Constructs an `enumerable_thread_specific` using per-element move construction from the values in `other`, and keeping their thread correspondence. `other` is left in an unspecified state, but can be safely destroyed.

Destructor

```
~enumerable_thread_specific()
```

Destroys all elements in `*this`. Destroys any native TLS keys that were created for this instance.

Assignment operators

```
enumerable_thread_specific& operator=( const enumerable_thread_specific& other );
```

Copies the content of `other` to `*this`. Returns a reference to `this*`.

```
template<typename Alloc, ets_key_usage_type Cachetype>
enumerable_thread_specific& operator=( const enumerable_thread_specific<T, Alloc,␣
↪Cachetype>& other );
```

Copies the content of `other` to `*this`. Returns a reference to `this*`.

Note: The allocator and key usage specialization is unchanged by this call.

```
enumerable_thread_specific& operator=( enumerable_thread_specific&& other );
```

Moves the content of `other` to `*this` intact. An `other` is left in an unspecified state, but can be safely destroyed. Returns a reference to `this*`.

```
template<typename Alloc, ets_key_usage_type Cachetype>
enumerable_thread_specific& operator=( enumerable_thread_specific<T, Alloc, Cachetype>
↪&& other );
```

Moves the content of `other` to `*this` using per-element move construction and keeping thread correspondence. An `other` is left in an unspecified state, but can be safely destroyed. Returns a reference to `this*`.

Note: The allocator and key usage specialization is unchanged by this call.

Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other.

reference **`local`**`()`
: If there is no current element corresponding to the current thread, this method constructs a new element. A new element is copy-constructed if an exemplar was provided to the constructor for `*this`; otherwise, a new element is default-constructed.

  Returns: A reference to the element of `*this` that corresponds to the current thread.

reference **`local`**`(`bool *&exists*`)`
: Similar to `local()`, except that `exists` is set to true if an element was already present for the current thread; false, otherwise.

  Returns: Reference to the thread-local element.

Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

clear

```
void clear();
```

Destroys all elements in `*this`.

Size and capacity

size_type **`size`**`()` **`const`**
: Returns the number of elements in `*this`. The value is equal to the number of distinct threads that have called `local()` after `*this` was constructed or most recently cleared.

bool **`empty`**`()` **`const`**
: Returns `true` if the container is empty; `false`, otherwise.

Iteration

Class template `enumerable_thread_specific` supports random access iterators, which enable iteration over the set of all elements in the container.

iterator **`begin`**`()`
: Returns `iterator` pointing to the beginning of the set of elements.

iterator **`end`**`()`
: Returns `iterator` pointing to the end of the set of elements.

const_iterator **`begin`**`() const`
: Returns `const_iterator` pointing to the beginning of the set of elements.

const_iterator **`end`**`() const`
: Returns `const_iterator` pointing to the end of the set of elements.

Class template `enumerable_thread_specific` supports `const_range_type` and `range_type` types, which model the *ContainerRange requirement*. The types differ only in that the bounds for a `const_range_type` are of type `const_iterator`, whereas the bounds for a `range_type` are of type `iterator`.

const_range_type **`range`**(size_t *grainsize* = 1) **`const`**
: **Returns**: A `const_range_type` representing all elements in `*this`. The parameter `grainsize` is in units of elements.

range_type **`range`**(size_t *grainsize* = 1)
: **Returns**: A `range_type` representing all elements in `*this`. The parameter `grainsize` is in units of elements.

Combining

The member functions in this section iterate across the entire container sequentially in the calling thread.

template<typename **`BinaryFunc`**>
T **`combine`**(*BinaryFunc f*)
: **Requires**: A `BinaryFunc` must meet the *Function Objects* requirements from the [function.objects] ISO C++ Standard section. Specifically, the type should be an associative binary functor with the signature `T BinaryFunc(T,T)` or `T BinaryFunc(const T&,const T&)`. A `T` type must be the same as a corresponding template parameter for `enumerable_thread_specific` object.

  Effects: Computes reduction over all elements using binary functor `f`. If there are no elements, creates the result using the same rules as for creating a thread-local element.

  Returns: Result of the reduction.

template<typename **`UnaryFunc`**>
void **`combine_each`**(*UnaryFunc f*)
: **Requires**: An `UnaryFunc` must meet the *Function Objects* requirements from the [function.objects] ISO C++ Standard section. Specifically, the type should be an unary functor with one of signatures: `void UnaryFunc(T)`, `void UnaryFunc(T&)`, or `void UnaryFunc(const T&)` A `T` type must be the same as a corresponding template parameter for the `enumerable_thread_specific` object.

  Effects: Evaluates `f(x)` for each instance `x` of `T` in `*this`.

Non-member types and constants

enum ets_key_usage_type::**ets_key_per_instance**

Enumeration parameter type used to select an implementation that consumes 1 native TLS key per enumerable_thread_specific instance. The number of native TLS keys may be limited and can be fairly small.

enum ets_key_usage_type::**ets_no_key**

Enumeration parameter type used to select an implementation that consumes no native TLS keys. If no ETS_key_usage_type parameter type is provided, ets_no_key is used by default.

enum ets_key_usage_type::**ets_suspend_aware**

The tbb::task::suspend function can change the value of the enumerable_thread_specific object. To avoid this problem, use the ets_suspend_aware enumeration parameter type. The local() value can be the same for different threads, but no two distinct threads can access the same value simultaneously.

This section also describes class template flatten2d, which assists a common idiom where an enumerable_thread_specific represents a container partitioner across threads.

flattened2d

[tls.flattened2d]

The class template flattened2d is an adaptor that provides a flattened view of a container of containers.

```
// Defined in header <tbb/enumerable_thread_specific.h>

namespace tbb {

    template<typename Container>
    class flattened2d {
    public:
        // Basic types
        using size_type = /* implementation-defined */;
        using difference_type = /* implementation-defined */;
        using allocator_type = /* implementation-defined */;
        using value_type = /* implementation-defined */;
        using reference = /* implementation-defined */;
        using const_reference = /* implementation-defined */;
        using pointer = /* implementation-defined */;
        using const_pointer = /* implementation-defined */;

        using iterator = /* implementation-defined */;
        using const_iterator = /* implementation-defined */;

        explicit flattened2d( const Container& c );

        flattened2d( const Container& c,
                     typename Container::const_iterator first,
                     typename Container::const_iterator last );

        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;
```

(continues on next page)

(continued from previous page)

```
        size_type size() const;
    };

    template <typename Container>
    flattened2d<Container> flatten2d(const Container &c);

    template <typename Container>
    flattened2d<Container> flatten2d(
        const Container &c,
        const typename Container::const_iterator first,
        const typename Container::const_iterator last);

} // namespace tbb
```

Requirements:

- A `Container` type must meet the container requirements from the [container.requirements.general] ISO C++ section.

Iterating from `begin()` to `end()` visits all of the elements in the inner containers. The class template supports forward iterators only.

The utility function `flatten2d` creates a `flattened2d` object from a specified container.

Member functions

explicit flattened2d(**const** Container &*c*)
: Constructs a `flattened2d` representing the sequence of elements in the inner containers contained by outer container `c`.

  Safety: these operations must not be invoked concurrently on the same `flattened2d`.

flattened2d(**const** Container &*c*, **typename** Container::const_iterator *first*, **typename** Container::const_iterator *last*)
: Constructs a `flattened2d` representing the sequence of elements in the inner containers in the half-open interval `[first, last)` of a container `c`.

  Safety: these operations must not be invoked concurrently on the same `flattened2d`.

size_type **size**() **const**
: Returns the sum of the sizes of the inner containers that are viewable in the `flattened2d`.

  Safety: These operations may be invoked concurrently on the same `flattened2d`.

iterator **begin**()
: Returns `iterator` pointing to the beginning of the set of local copies.

iterator **end**()
: Returns `iterator` pointing to the end of the set of local copies.

const_iterator **begin**() **const**
: Returns `const_iterator` pointing to the beginning of the set of local copies.

const_iterator **end**() **const**
: Returns `const_iterator` pointing to the end of the set of local copies.

Non-member functions

template<typename **Container**>
flattened2d<*Container*> **flatten2d**(**const** *Container* &*c*, **const typename** *Container*::const_iterator *b*,
const typename *Container*::const_iterator *e*)

Constructs and returns a `flattened2d` object that provides iterators that traverse the elements in the containers within the half-open range `[b, e)` of a container `c`.

template<typename **Container**>
flattened2d(**const** *Container* &*c*)

Constructs and returns a `flattened2d` that provides iterators that traverse the elements in all of the containers within a container `c`.

9.3 oneTBB Auxiliary Interfaces

9.3.1 Memory Allocation

[memory_allocation]

This section describes classes and functions related to memory allocation.

Allocators

The oneAPI Threading Building Blocks (oneTBB) library implements several classes that meet the allocator requirements from the [allocator.requirements] ISO C++ Standard section.

tbb_allocator

[memory_allocation.tbb_allocator]

A `tbb_allocator` is a class template that models the allocator requirements from the [allocator.requirements] ISO C++ section.

The `tbb_allocator` allocates and frees memory via the oneTBB malloc library if it is available, otherwise, it reverts to using `std::malloc` and `std::free`.

```
// Defined in header <tbb/tbb_allocator.h>

namespace tbb {
    template<typename T> class tbb_allocator {
    public:
        using value_type = T;
        using size_type = std::size_t;
        using propagate_on_container_move_assignment = std::true_type;
        using is_always_equal = std::true_type;

        enum malloc_type {
            scalable,
            standard
        };

        tbb_allocator() = default;
        template<typename U>
```

(continues on next page)

(continued from previous page)

```
        tbb_allocator(const tbb_allocator<U>&) noexcept;

        T* allocate(size_type);
        void deallocate(T*, size_type);

        static malloc_type allocator_type();
    };
}
```

Member Functions

T ***allocate**(*size_type n*)
: Allocates `n * sizeof(T)` bytes. Returns a pointer to the allocated memory.

void **deallocate**(T **p*, *size_type n*)
: Deallocates memory pointed to by `p`. The behavior is undefined if the pointer `p` is not the result of the `allocate(n)` method. The behavior is undefined if the memory has been already deallocated.

static malloc_type **allocator_type**()
: Returns the enumeration type `malloc_type::scalable` if the oneTBB malloc library is available, and `malloc_type::standard`, otherwise.

Non-member Functions

These functions provide comparison operations between two `tbb_allocator` instances.

```
template<typename T, typename U>
bool operator==(const tbb_allocator<T>&, const tbb_allocator<U>&) noexcept;

template<typename T, typename U>
bool operator!=(const tbb_allocator<T>&, const tbb_allocator<U>&) noexcept;
```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on `tbb_allocator` objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace and define `tbb::tbb_allocator` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

template<typename **T**, typename **U**>
bool **operator==**(**const** tbb_allocator<*T*>&, **const** tbb_allocator<*U*>&) **noexcept**
: Returns **true**.

template<typename **T**, typename **U**>
bool **operator!=**(**const** tbb_allocator<*T*>&, **const** tbb_allocator<*U*>&) **noexcept**
: Returns **false**.

scalable_allocator

[memory_allocation.scalable_allocator]

A `scalable_allocator` is a class template that models the allocator requirements from the [allocator.requirements] ISO C++ section.

The `scalable_allocator` allocates and frees memory in a way that scales with the number of processors. Memory allocated by a `scalable_allocator` should be freed by a `scalable_allocator`, not by a `std::allocator`.

```
// Defined in header <tbb/scalable_allocator.h>

namespace tbb {
    template<typename T> class scalable_allocator {
    public:
        using value_type = T;
        using size_type = std::size_t;
        using propagate_on_container_move_assignment = std::true_type;
        using is_always_equal = std::true_type;

        scalable_allocator() = default;
        template<typename U>
        scalable_allocator(const scalable_allocator<U>&) noexcept;

        T* allocate(size_type);
        void deallocate(T*, size_type);
    };
}
```

Caution: The `scalable_allocator` requires the memory allocator library. If the library is missing, calls to the scalable allocator fail. In contrast to `scalable_allocator`, if the memory allocator library is not available, `tbb_allocator` falls back on `std::malloc` and `std::free`.

Member Functions

value_type ***allocate**(*size_type n*)
: Allocates `n * sizeof(T)` bytes of memory. Returns a pointer to the allocated memory.

void **deallocate**(value_type **p*, *size_type n*)
: Deallocates memory pointed to by `p`. The behavior is undefined if the pointer `p` is not the result of the `allocate(n)` method. The behavior is undefined if the memory has been already deallocated.

Non-member Functions

These functions provide comparison operations between two `scalable_allocator` instances.

```
namespace tbb {
    template<typename T, typename U>
    bool operator==(const scalable_allocator<T>&,
                    const scalable_allocator<U>&) noexcept;

    template<typename T, typename U>
```

(continues on next page)

(continued from previous page)

```
    bool operator!=(const scalable_allocator<T>&,
                    const scalable_allocator<U>&) noexcept;
}
```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on `scalable_allocator` objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define `tbb::scalable_allocator` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

template<typename **T**, typename **U**>
bool **operator==**(**const** scalable_allocator<*T*>&, **const** scalable_allocator<*U*>&) **noexcept**
Returns **true**.

template<typename **T**, typename **U**>
bool **operator!=**(**const** scalable_allocator<*T*>&, **const** scalable_allocator<*U*>&) **noexcept**
Returns **false**.

cache_aligned_allocator

[memory_allocation.cache_aligned_allocator]

A `cache_aligned_allocator` is a class template that models the allocator requirements from the [allocator.requirements] ISO C++ section.

The `cache_aligned_allocator` allocates memory on cache line boundaries, in order to avoid false sharing and potentially improve performance. False sharing is a situation when logically distinct items occupy the same cache line, which can hurt performance if multiple threads attempt to access the different items simultaneously. Even though the items are logically separate, the processor hardware may have to transfer the cache line between the processors as if they were sharing a location. The net result can be much more memory traffic than if the logically distinct items were on different cache lines.

However, this class is sometimes an inappropriate replacement for default allocator, because the benefit of allocating on a cache line comes at the price that `cache_aligned_allocator` implicitly adds pad memory. Therefore allocating many small objects with `cache_aligned_allocator` may increase memory usage.

```
// Defined in header <tbb/cache_aligned_allocator.h>

namespace tbb {
    template<typename T> class cache_aligned_allocator {
    public:
        using value_type = T;
        using size_type = std::size_t;
        using propagate_on_container_move_assignment = std::true_type;
        using is_always_equal = std::true_type;

        cache_aligned_allocator() = default;
        template<typename U>
        cache_aligned_allocator(const cache_aligned_allocator<U>&) noexcept;

        T* allocate(size_type);
        void deallocate(T*, size_type);
        size_type max_size() const noexcept;
    };
}
```

Member Functions

T *__allocate__(*size_type n*)
: Returns a pointer to the allocated `n * sizeof(T)` bytes of memory, aligned on a cache-line boundary. The allocation may include extra hidden padding.

void __deallocate__(T *p, *size_type n*)
: Deallocates memory pointed to by `p`. Deallocation also deallocates any extra hidden padding. The behavior is undefined if the pointer `p` is not the result of the `allocate(n)` method. The behavior is undefined if the memory has been already deallocated.

size_type __max_size__() __const noexcept__
: Returns the largest value `n` for which the call `allocate(n)` might succeed with cache alignment constraints.

Non-member Functions

These functions provide comparison operations between two `cache_aligned_allocator` instances.

```
template<typename T, typename U>
bool operator==(const cache_aligned_allocator<T>&,
                const cache_aligned_allocator<U>&) noexcept;

template<typename T, typename U>
bool operator!=(const cache_aligned_allocator<T>&,
                const cache_aligned_allocator<U>&) noexcept;
```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on `cache_aligned_allocator` objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define `tbb::cache_aligned_allocator` as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

template<typename __T__, typename __U__>
bool __operator==__(__const__ cache_aligned_allocator<*T*>&, __const__ cache_aligned_allocator<*U*>&) __noexcept__
: Returns **true**.

template<typename __T__, typename __U__>
bool __operator!=__(__const__ cache_aligned_allocator<*T*>&, __const__ cache_aligned_allocator<*U*>&) __noexcept__
: Returns **false**.

Memory Resources

Starting from C++17, the standard library provides a `std::pmr::polymorphic_allocator` class that allocates memory from a supplied memory resource (see the [mem.poly.allocator.class] ISO/IEC 14882:2017 section). Class `std::pmr::memory_resource` is an abstract interface for user-side implementation of different allocation strategies. For details, see the [mem.res.class] ISO/IEC 14882:2017 standard section.

oneTBB provides a set of `std::pmr::memory_resource` implementations.

cache_aligned_resource

[memory_allocation.cache_aligned_resource]

A `cache_aligned_resource` is a general-purpose memory resource class, which acts as a wrapper to another memory resource to ensure that all allocations are aligned on cache line boundaries to avoid false sharing.

See the *cache_aligned_allocator template class* section for more information about false sharing avoidance.

```
// Defined in header <tbb/cache_aligned_allocator.h>

namespace tbb {
    class cache_aligned_resource {
    public:
        cache_aligned_resource();
        explicit cache_aligned_resource( std::pmr::memory_resource* );

        std::pmr::memory_resource* upstream_resource() const;

    private:
        void* do_allocate(size_t n, size_t alignment) override;
        void do_deallocate(void* p, size_t n, size_t alignment) override;
        bool do_is_equal(const std::pmr::memory_resource& other) const noexcept␣
↪override;
    };
}
```

Member Functions

cache_aligned_resource()
: Constructs a `cache_aligned_resource` over `std::pmr::get_default_resource()`.

explicit cache_aligned_resource(std::pmr::memory_resource *r)
: Constructs a `cache_aligned_resource` over the memory resource `r`.

std::pmr::memory_resource ***upstream_resource**() **const**
: Returns the pointer to the underlying memory resource.

void ***do_allocate**(size_t *n*, size_t *alignment*) **override**
: Allocates `n` bytes of memory on a cache-line boundary, with alignment not less than requested. The allocation may include extra memory for padding. Returns pointer to the allocated memory.

void **do_deallocate**(void *p, size_t *n*, size_t *alignment*) **override**
: Deallocates memory pointed to by `p` and any extra padding. Pointer `p` must be obtained with `do_allocate(n, alignment)`. The memory must not be deallocated beforehand.

bool **do_is_equal**(**const** std::pmr::memory_resource &*other*) **const noexcept override**
: Compares upstream memory resources of `*this` and `other`. If `other` is not a `cache_aligned_resource`, returns false.

scalable_memory_resource

[memory_allocation.scalable_memory_resource]

A `tbb::scalable_memory_resource()` is a function that returns a memory resource for scalable memory allocation.

The `scalable_memory_resource()` function returns the pointer to the memory resource managed by the oneTBB scalable memory allocator. In particular, its `allocate` method uses `scalable_aligned_malloc()`, and `deallocate` uses `scalable_free()`. The memory resources returned by this function compare equal.

`std::pmr::polymorphic_allocator` instantiated with `tbb::scalable_memory_resource()` behaves like `tbb::scalable_allocator`.

```
// Defined in header <tbb/scalable_allocator.h>

std::pmr::memory_resource* scalable_memory_resource();
```

Library Functions

C Interface to Scalable Allocator

[memory_allocation.scalable_alloc_c_interface]

Low-level interface for scalable memory allocation.

```
// Defined in header <tbb/scalable_allocator.h>

extern "C" {
    // Scalable analogs of C memory allocator
    void* scalable_malloc( size_t size );
    void  scalable_free( void* ptr );
    void* scalable_calloc( size_t nobj, size_t size );
    void* scalable_realloc( void* ptr, size_t size );

    // Analog of _msize/malloc_size/malloc_usable_size.
    size_t scalable_msize( void* ptr );

    // Scalable analog of posix_memalign
    int scalable_posix_memalign( void** memptr, size_t alignment, size_t size );

    // Aligned allocation
    void* scalable_aligned_malloc( size_t size, size_t alignment);
    void scalable_aligned_free( void* ptr );
    void* scalable_aligned_realloc( void* ptr, size_t size, size_t alignment );

    // Return values for scalable_allocation_* functions
    typedef enum {
        TBBMALLOC_OK,
        TBBMALLOC_INVALID_PARAM,
        TBBMALLOC_UNSUPPORTED,
        TBBMALLOC_NO_MEMORY,
        TBBMALLOC_NO_EFFECT
    } ScalableAllocationResult;

    typedef enum {
```

(continues on next page)

(continued from previous page)

```
        // To turn on/off the use of huge memory pages
        TBBMALLOC_USE_HUGE_PAGES,
        // To set a threshold for the allocator memory usage.
        // Exceeding it will forcefully clean internal memory buffers
        TBBMALLOC_SET_SOFT_HEAP_LIMIT,
        // Lower bound for the size (Bytes), that is interpreted as huge
        // and not released during regular cleanup operations
        TBBMALLOC_SET_HUGE_SIZE_THRESHOLD
    } AllocationModeParam;

    // Set allocator-specific allocation modes.
    int scalable_allocation_mode(int param, intptr_t value);

    typedef enum {
        // Clean internal allocator buffers for all threads.
        TBBMALLOC_CLEAN_ALL_BUFFERS,
        // Clean internal allocator buffer for current thread only.
        TBBMALLOC_CLEAN_THREAD_BUFFERS
    } ScalableAllocationCmd;

    // Call allocator-specific commands.
    int scalable_allocation_command(int cmd, void *param);
}
```

These functions provide a C-level interface to the scalable allocator. With the exception of `scalable_allocation_mode` and `scalable_allocation_command`, each routine `scalable_x` behaves analogously to the library function `x`. The routines form the two families shown in the table below, "C Interface to Scalable Allocator". Storage allocated by a `scalable_x` function in one family must be freed or resized by the `scalable_x` function in the same family, not by a C standard library function. Likewise, storage allocated by a C standard library function should not be freed or resized by a `scalable_x` function.

Table 5: C Interface to Scalable Allocator

Allocation Routine	Deallocation Routine	Analogous Library
scalable_malloc	scalable_free	C standard library
scalable_calloc		
scalable_realloc		
scalable_posix_memalign		POSIX*
scalable_aligned_malloc	scalable_aligned_free	Microsoft* C run-time library
scalable_aligned_realloc		

The following functions do not allocate or free memory but allow obtaining useful information or influencing behavior of the memory allocator.

size_t **scalable_msize** (void *_ptr_)

Returns: The usable size of the memory block pointed to by *ptr* if it was allocated by the scalable allocator. Returns zero if *ptr* does not point to such a block.

int **scalable_allocation_mode** (int *mode*, intptr_t *value*)

Use this function to adjust behavior of the scalable memory allocator.

Returns: TBBMALLOC_OK if the operation succeeded, TBBMALLOC_INVALID_PARAM if `mode` is not one of the described below, or if `value` is not valid for the given mode. Other return values are possible, as described below.

---

scalable_allocation_mode Parameters: Parameter, Description

TBBMALLOC_USE_HUGE_PAGES
`scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 1)` tells the allocator to use huge pages if enabled by the operating system. `scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 0)` disables it. Setting `TBB_MALLOC_USE_HUGE_PAGES` environment variable to 1 has the same effect as `scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 1)`. The mode set with `scalable_allocation_mode()` takes priority over the environment variable.

May return: `TBBMALLOC_NO_EFFECT` if huge pages are not supported on the platform.

For now, this allocation mode is only supported for Linux* OS. It works with both explicitly configured and transparent huge pages. For information about enabling and configuring huge pages, refer to OS documentation or ask your system administrator.

TBBMALLOC_SET_SOFT_HEAP_LIMIT
`scalable_allocation_mode(TBBMALLOC_SET_SOFT_HEAP_LIMIT, size)` sets a threshold of `size` bytes on the amount of memory the allocator takes from OS. Exceeding the threshold urges the allocator to release memory from its internal buffers; however it does not prevent from requesting more memory if needed.

TBBMALLOC_SET_HUGE_SIZE_THRESHOLD
`scalable_allocation_mode(TBBMALLOC_SET_HUGE_SIZE_THRESHOLD, size)` sets a lower bound threshold (with no upper limit) of `size` bytes. Any object bigger than this threshold becomes huge and does not participate in internal periodic cleanup logic. However, it does not affect the logic of the `TBBMALLOC_SET_SOFT_HEAP_LIMIT` mode as well as the `TBBMALLOC_CLEAN_ALL_BUFFERS` operation.

Setting `TBB_MALLOC_SET_HUGE_SIZE_THRESHOLD` environment variable to the size value has the same effect, but is limited to the `LONG_MAX` value. The mode set with `scalable_allocation_mode` takes priority over the environment variable.

---

int **scalable_allocation_command**(int *cmd*, void **reserved*)
This function may be used to command the scalable memory allocator to perform an action specified by the first parameter. The second parameter is reserved and must be set to 0.

Returns: `TBBMALLOC_OK` if the operation succeeded, `TBBMALLOC_INVALID_PARAM` if `cmd` is not one of the described below, or if `reserved` is not equal to 0.

---

scalable_allocation_command Parameters: Parameter, Description

TBBMALLOC_CLEAN_ALL_BUFFERS
`scalable_allocation_command(TBBMALLOC_CLEAN_ALL_BUFFERS, 0)` cleans internal memory buffers of the allocator, and possibly reduces memory footprint. It may result in increased time for subsequent memory allocation requests. The command is not designed for frequent use, and careful evaluation of the performance impact is recommended.

May return: `TBBMALLOC_NO_EFFECT` if no buffers were released.

Note: It is not guaranteed that the call will release all unused memory.

TBBMALLOC_CLEAN_THREAD_BUFFERS
`scalable_allocation_command(TBBMALLOC_CLEAN_THREAD_BUFFERS, 0)` cleans internal memory buffers, but only for the calling thread.

May return: `TBBMALLOC_NO_EFFECT` if no buffers were released.

9.3.2 Mutual Exclusion

[mutex]

The library provides a set of mutual exclusion primitives to simplify writing race-free code. A mutex object facilitates protection against data races and provides safe synchronization of data between threads.

Mutex Classes

spin_mutex

[mutex.spin_mutex]

A `spin_mutext` is a class that models the *Mutex requirement* using a spin lock. The `spin_mutex` class satisfies all requirements of mutex type from the [thread.mutex.requirements] ISO C++ section. The `spin_mutex` class is not fair or recursive.

```
// Defined in header <tbb/spin_mutex.h>

namespace tbb {
    class spin_mutex {
    public:
        spin_mutex() noexcept;
        ~spin_mutex();

        spin_mutex(const spin_mutex&) = delete;
        spin_mutex& operator=(const spin_mutex&) = delete;

        class scoped_lock;

        void lock();
        bool try_lock();
        void unlock();

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = false;
    };
}
```

Member classes

class scoped_lock
Corresponding `scoped_lock` class. See the *Mutex requirement*.

Member functions

spin_mutex()
Constructs `spin_mutex` with unlocked state.

~spin_mutex()
Destroys an unlocked `spin_mutex`.

void **lock**()
Acquires a lock. Spins if the lock is taken.

bool **try_lock**()
Attempts to acquire a lock (non-blocking). Returns **true** if lock is acquired; **false**, otherwise.

void **unlock**()
Releases a lock held by a current thread.

spin_rw_mutex

[mutex.spin_rw_mutex]

A `spin_rw_mutex` is a class that models the *ReaderWriterMutex requirement* and satisfies all requirements of shared mutex type from the [thread.sharedmutex.requirements] ISO C++ section.

The `spin_rw_mutex` class is unfair spinning reader-writer lock with backoff and writer-preference.

```
// Defined in header <tbb/spin_rw_mutex.h>

namespace tbb {
    class spin_rw_mutex {
    public:
        spin_rw_mutex() noexcept;
        ~spin_rw_mutex();

        spin_rw_mutex(const spin_rw_mutex&) = delete;
        spin_rw_mutex& operator=(const spin_rw_mutex&) = delete;

        class scoped_lock;

        // exclusive ownership
        void lock();
        bool try_lock();
        void unlock();

        // shared ownership
        void lock_shared();
        bool try_lock_shared();
        void unlock_shared();

        static constexpr bool is_rw_mutex = true;
        static constexpr bool is_recursive_mutex = false;
```

(continues on next page)

(continued from previous page)

```
        static constexpr bool is_fair_mutex = false;
    };
}
```

Member classes

class scoped_lock
Corresponding scoped-lock class. See the *ReaderWriterMutex requirement*.

Member functions

spin_rw_mutex()
Constructs unlocked `spin_rw_mutex`.

~spin_rw_mutex()
Destroys unlocked `spin_rw_mutex`.

void **lock**()
Acquires a lock. Spins if the lock is taken.

bool **try_lock**()
Attempts to acquire a lock (non-blocking) on write. Returns true if the lock is acquired on write; false otherwise.

void **unlock**()
Releases a write lock, held by the current thread.

void **lock_shared**()
Acquires a lock on read. Spins if the lock is taken on write already.

bool **try_lock_shared**()
Attempts to acquire the lock (non-blocking) on read. Returns true if the lock is acquired on read; false, otherwise.

void **unlock_shared**()
Releases a read lock held by the current thread.

speculative_spin_mutex

[mutex.speculative_spin_mutex]

A `speculative_spin_mutex` is a class that models the *Mutex requirement* using a spin lock, and for processors that support hardware transactional memory (such as Intel® Transactional Synchronization Extensions (Intel® TSX)) may be implemented in a way that allows non-contending changes to the protected data to proceed in parallel.

The `speculative_spin_mutex` is not fair and not recursive. The `speculative_spin_mutex` is like a `spin_mutex`, but it may provide better throughput than non-speculative mutexes when the following conditions are met:

- Running on a processor that supports hardware transactional memory.
- Multiple threads can concurrently execute the critical section(s) protected by the mutex, mostly without conflicting.

Otherwise, it performs like a `spin_mutex`, possibly with worse throughput.

```
// Defined in header <tbb/spin_mutex.h>

namespace tbb {
    class speculative_spin_mutex {
    public:
        speculative_spin_mutex() noexcept;
        ~speculative_spin_mutex();

        speculative_spin_mutex(const speculative_spin_mutex&) = delete;
        speculative_spin_mutex& operator=(const speculative_spin_mutex&) = delete;

        class scoped_lock;

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = false;
    };
}
```

Member classes

class scoped_lock
Corresponding `scoped_lock` class. See the *Mutex requirement*.

Member functions

speculative_spin_mutex()
Constructs `speculative_spin_mutex` with unlocked state.

~speculative_spin_mutex()
Destroys an unlocked `speculative_spin_mutex`.

speculative_spin_rw_mutex

[mutex.speculative_spin_rw_mutex]

A `speculative_spin_rw_mutex` is a class that models the *ReaderWriterMutex requirement*, and for processors that support hardware transactional memory (such as Intel® Transactional Synchronization Extensions (Intel® TSX)) may be implemented in a way that allows non-contending changes to the protected data to proceed in parallel.

The `speculative_spin_rw_mutex` class is not fair and not recursive. The `speculative_spin_rw_mutex` class is like a `spin_rw_mutex`, but it may provide better throughput than non-speculative mutexes when the following conditions are met:

- Running on a processor that supports hardware transactional memory.
- Multiple threads can concurrently execute the critical section(s) protected by the mutex, mostly without conflicting.

Otherwise, it performs like a `spin_rw_mutex`, possibly with worse throughput.

For processors that support hardware transactional memory, `speculative_spin_rw_mutex` may be implemented in a way that

- speculative readers and writers do not block each other

- a non-speculative reader blocks writers but allows speculative readers
- a non-speculative writer blocks all readers and writers

```
// Defined in header <tbb/spin_rw_mutex.h>

namespace tbb {
    class speculative_spin_rw_mutex {
    public:
        speculative_spin_rw_mutex() noexcept;
        ~speculative_spin_rw_mutex();

        speculative_spin_rw_mutex(const speculative_spin_rw_mutex&) = delete;
        speculative_spin_rw_mutex& operator=(const speculative_spin_rw_mutex&) =␣
↪delete;

        class scoped_lock;

        static constexpr bool is_rw_mutex = true;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = false;
    };
}
```

Member classes

class scoped_lock
Corresponding `scoped_lock` class. See the *ReaderWriterMutex requirement*.

Member functions

speculative_spin_rw_mutex()
Constructs `speculative_spin_rw_mutex` with unlocked state.

~speculative_spin_rw_mutex()
Destroys an unlocked `speculative_spin_rw_mutex`.

queuing_mutex

[mutex.queuing_mutex]

A `queuing_mutex` is a class that models the *Mutex requirement*. The `queuing_mutex` is not recursive. The `queuing_mutex` is fair, threads acquire a lock on a mutex in the order that they request it.

```
// Defined in header <tbb/queuing_mutex.h>

namespace tbb {
    class queuing_mutex {
    public:
        queuing_mutex() noexcept;
        ~queuing_mutex();

        queuing_mutex(const queuing_mutex&) = delete;
        queuing_mutex& operator=(const queuing_mutex&) = delete;
```

(continues on next page)

(continued from previous page)

```
        class scoped_lock;

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = true;
    };
}
```

Member classes

class scoped_lock
Corresponding `scoped_lock` class. See the *Mutex requirement*.

Member functions

queuing_mutex()
Constructs unlocked `queuing_mutex`.

~queuing_mutex()
Destroys unlocked `queuing_mutex`.

queuing_rw_mutex

[mutex.queuing_rw_mutex]

A `queuing_rw_mutex` is a class that models the *ReaderWriterMutex requirement* concept. The `queuing_rw_mutex` is not recursive. The `queuing_rw_mutex` is fair, threads acquire a lock on a mutex in the order that they request it.

```
// Defined in header <tbb/queuing_rw_mutex.h>

namespace tbb {
    class queuing_rw_mutex {
    public:
        queuing_rw_mutex() noexcept;
        ~queuing_rw_mutex();

        queuing_rw_mutex(const queuing_rw_mutex&) = delete;
        queuing_rw_mutex& operator=(const queuing_rw_mutex&) = delete;

        class scoped_lock;

        static constexpr bool is_rw_mutex = true;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = true;
    };
}
```

Member classes

class scoped_lock
Corresponding `scoped_lock` class. See the *ReaderWriterMutex requirement*.

Member functions

queuing_rw_mutex()
Constructs unlocked `queuing_rw_mutex`.

~queuing_rw_mutex()
Destroys unlocked `queuing_rw_mutex`.

null_mutex

[mutex.null_mutex]

A `null_mutex` is a class that models the *Mutex requirement* concept syntactically, but does nothing. It is useful for instantiating a template that expects a Mutex, but no mutual exclusion is actually needed for that instance.

```
// Defined in header <tbb/null_mutex.h>

namespace tbb {
    class null_mutex {
    public:
        constexpr null_mutex() noexcept;
        ~null_mutex();

        null_mutex(const null_mutex&) = delete;
        null_mutex& operator=(const null_mutex&) = delete;

        class scoped_lock;

        void lock();
        bool try_lock();
        void unlock();

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = true;
        static constexpr bool is_fair_mutex = true;
    };
}
```

Member classes

class scoped_lock
Corresponding `scoped_lock` class. See the *Mutex requirement*.

Member functions

null_mutex()
: Constructs unlocked mutex.

~null_mutex()
: Destroys unlocked mutex.

void **lock**()
: Acquires lock.

bool **try_lock**()
: Tries acquiring lock (non-blocking).

void **unlock**()
: Releases the lock.

null_rw_mutex

[mutex.null_rw_mutex]

A `null_rw_mutex` is a class that models the *ReaderWriterMutex requirement* syntactically, but does nothing. The `null_rw_mutex` class also satisfies all syntactic requirements of shared mutex type from the [thread.sharedmutex.requirements] ISO C++ section, but does nothing. It is useful for instantiating a template that expects a ReaderWriterMutex, but no mutual exclusion is actually needed for that instance.

```
// Defined in header <tbb/null_rw_mutex.h>

namespace tbb {
    class null_rw_mutex {
    public:
        constexpr null_rw_mutex() noexcept;
        ~null_rw_mutex();

        null_rw_mutex(const null_rw_mutex&) = delete;
        null_rw_mutex& operator=(const null_rw_mutex&) = delete;

        class scoped_lock;

        void lock();
        bool try_lock();
        void unlock();

        void lock_shared();
        bool try_lock_shared();
        void unlock_shared();

        static constexpr bool is_rw_mutex = true;
        static constexpr bool is_recursive_mutex = true;
        static constexpr bool is_fair_mutex = true;
    };
}
```

Member classes

class scoped_lock
Corresponding `scoped_lock` class. See the *ReaderWriterMutex requirement*.

Member functions

null_rw_mutex()
Constructs unlocked mutex.

~null_rw_mutex()
Destroys unlocked mutex.

void **lock**()
Acquires a lock.

bool **try_lock**()
Attempts to acquire a lock (non-blocking) on write. Returns **true**.

void **unlock**()
Releases a write lock held by the current thread.

void **lock_shared**()
Acquires a lock on read.

bool **try_lock_shared**()
Attempts to acquire the lock (non-blocking) on read. Returns **true**.

void **unlock_shared**()
Releases a read lock held by the current thread.

9.3.3 Timing

[timing]

Parallel programming is about speeding up *wall clock* time, which is the real time that it takes a program or function to run. The library provides API to simplify timing within an application.

Syntax

```
// Declared in tick_count.h

class tick_count;

class tick_count::interval_t;
```

Classes

tick_count class

[timing.tick_count]

A `tick_count` is an absolute wall clock timestamp. Two `tick_count` objects can be subtracted to compute wall clock duration `tick_count::interval_t`, which can be converted to seconds.

```
namespace tbb {

    class tick_count {
    public:
        class interval_t;
        tick_count();
        tick_count( const tick_count& );
        ~tick_count();
        tick_count& operator=( const tick_count& );
        static tick_count now();
        static double resolution();
    };

} // namespace tbb
```

`tick_count()` Constructs `tick_count` with an unspecified wall clock timestamp.

`tick_count( const tick_count& )` Constructs `tick_count` with the timestamp of the given `tick_count`.

`~tick_count()` Destructor.

`tick_count& operator=( const tick_count& )` Assigns the timestamp of one `tick_count` to another.

`static tick_count now()` Returns a `tick_count` object that represents the current wall clock timestamp.

`static double resolution()` Returns the resolution of the clock used by `tick_count`, in seconds.

tick_count::interval_t class

[timing.tick_count.interval_t]

A `tick_count::interval_t` represents wall clock duration.

```
namespace tbb {

    class tick_count::interval_t {
    public:
        interval_t();
        explicit interval_t( double );
        ~interval_t();
        interval_t& operator=( const interval_t& );
        interval_t& operator+=( const interval_t& );
        interval_t& operator-=( const interval_t& );
        double seconds() const;
    };

} // namespace tbb
```

interval_t() Constructs interval_t representing zero time duration.

explicit interval_t(double) Constructs interval_t representing the specified number of seconds.

~interval_t() Destructor.

interval_t& operator=(const interval_t&) Assigns the wall clock duration of one interval_t to another.

interval_t& operator+=(const interval_t&) Increases the duration to the given interval_t, and returns *this.

interval_t& operator-=(const interval_t&) Decreases the duration to the given interval_t, and returns *this.

double seconds() const Returns the duration measured in seconds.

Non-member functions

[timing.tick_count.nonmember]

These functions provide arithmetic binary operations with wall clock timestamps and durations.

```
tbb::tick_count::interval_t operator-( const tbb::tick_count&, const tbb::tick_count&␣
↪);
tbb::tick_count::interval_t operator+( const tbb::tick_count::interval_t&, const␣
↪tbb::tick_count::interval_t& );
tbb::tick_count::interval_t operator-( const tbb::tick_count::interval_t&, const␣
↪tbb::tick_count::interval_t& );
```

The namespace where these functions are defined is unspecified as long as they may be used in respective binary operation expressions on tick_count and tick_count::interval_t objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define tbb::tick_count as a type alias for which the non-member functions are reachable only via argument-dependent lookup.

tbb::tick_count::interval_t operator-(const tbb::tick_count&, const tbb::tick_count&) Returns interval_t representing the duration between two given wall clock timestamps.

tbb::tick_count::interval_t operator+(const tbb::tick_count::interval_t&, const tbb::tick_ Returns interval_t representing the sum of two given intervals.

tbb::tick_count::interval_t operator-(const tbb::tick_count::interval_t&, const tbb::tick_ Returns interval_t representing the difference of two given intervals.

9.3.4 info Namespace

[info_namespace]

Interfaces to query information about execution environment.

```
// Declared in info.h

namespace tbb {
    using numa_node_id = /*implementation-defined*/;
    namespace info {
        std::vector<numa_node_id> numa_nodes();
        int default_concurrency(numa_node_id id = tbb::task_arena::automatic);
    }
}
```

Types

`numa_node_id` - Represents NUMA node identifier.

Functions

std::vector<numa_node_id> **`numa_nodes`**`()`
Returns the vector of integral indexes that indicate available NUMA nodes.

Note: If error occurs during system topology parsing, returns vector containing single element that equals to `task_arena::automatic`.

---

int **`default_concurrency`**`(`numa_node_id *id* = tbb::task_arena::automatic`)`
Returns concurrency level of the given NUMA node. If argument is not specified, returns default concurrency level for current library configuration.

CHAPTER

TEN

ONEVPL

The oneAPI Video Processing Library (oneVPL) is a programming interface for video decoding, encoding, and processing to build portable media pipelines on CPUs, GPUs, and other accelerators. It provides device discovery and selection in media centric and video analytics workloads and API primitives for zero-copy buffer sharing. oneVPL is backwards and cross-architecture compatible to ensure optimal execution on current and next generation hardware without source code changes.

CHAPTER

ELEVEN

ONEVPL SPECIFICATION VERSION

Latest oneVPL specification version is 2.1.0.

11.1 Architecture

oneVPL functions fall into the following categories:

DECODE Functions that decode compressed video streams into raw video frames

ENCODE Functions that encode raw video frames into compressed bitstreams

VPP Functions that perform video processing on raw video frames

DECODE_VPP Functions that perfom combined operations of decoding and video processing

CORE Auxiliary functions for synchronization

Misc Global auxiliary functions

With the exception of the global auxiliary functions, oneVPL functions are named after their functioning domain and category. oneVPL exposes video domain functions.

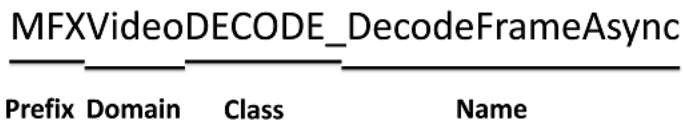


Fig. 1: oneVPL function name notation

Applications use oneVPL functions by linking with the oneVPL dispatcher library.

The dispatcher library identifies the hardware acceleration device on the running platform, determines the most suitable platform library for the identified hardware acceleration, and then redirects function calls accordingly.

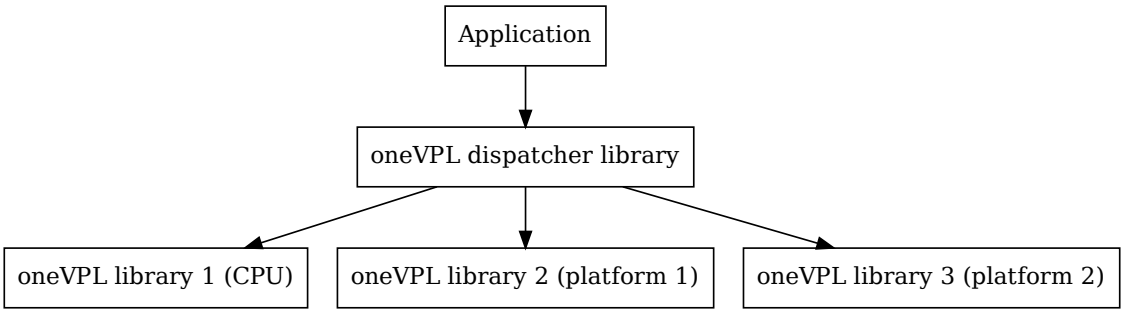


Fig. 2: oneVPL dispatching mechanism

11.1.1 Video Decoding

The *DECODE* class of functions take a compressed bitstream as input and converts it to raw frames as output.

DECODE processes only pure or elementary video streams with the exception of AV1/VP9/VP8 decoders, which accept the IVF format. The library can process bitstreams that reside in an IVF container but cannot process bitstreams that reside in any other container format, such as MP4 or MPEG.

The application must first demultiplex the bitstreams. Demultiplexing extracts pure video streams out of the container format. The application can provide the input bitstream as one complete frame of data, a partial frame (less than one complete frame), or as multiple frames. If only a partial frame is provided, DECODE internally constructs one frame of data before decoding it.

The time stamp of a bitstream buffer must be accurate to the first byte of the frame data. For H.264 the first byte of the frame data comes from the NAL unit in the video coding layer. For MPEG-2 or VC-1 the first byte of the frame data comes from the picture header. DECODE passes the time stamp to the output surface for audio and video multiplexing or synchronization.

Decoding the first frame is a special case because DECODE does not provide enough configuration parameters to correctly process the bitstream. DECODE searches for the sequence header (a sequence parameter set in H.264 or a sequence header in MPEG-2 and VC-1) that contains the video configuration parameters used to encode subsequent video frames. The decoder skips any bitstream prior to the sequence header. In the case of multiple sequence headers in the bitstream, DECODE adopts the new configuration parameters, ensuring proper decoding of subsequent frames.

DECODE supports repositioning of the bitstream at any time during decoding. Because there is no way to obtain the correct sequence header associated with the specified bitstream position after a position change, the application must supply DECODE with a sequence header before the decoder can process the next frame at the new position. If the sequence header required to correctly decode the bitstream at the new position is not provided by the application, DECODE treats the new location as a new “first frame” and follows the procedure for decoding first frames.

11.1.2 Video Encoding

The *ENCODE* class of functions take raw frames as input and compresses them into a bitstream.

Input frames usually come encoded in a repeated pattern called the Group of Picture (GOP) sequence. For example, a GOP sequence can start with an I-frame followed by a few B-frames, a P-frame, and so on. ENCODE uses an MPEG-2 style GOP sequence structure that can specify the length of the sequence and the distance between two keyframes: I- or P-frames. A GOP sequence ensures that the segments of a bitstream do not completely depend upon each other. It also enables decoding applications to reposition the bitstream.

ENCODE processes input frames in two ways;

- **Display order:** ENCODE receives input frames in the display order. GOP structure parameters specify the GOP sequence during ENCODE initialization. Scene changes resulting from the video processing stage of a pipeline can alter the GOP sequence.
- **Encoded order:** ENCODE receives input frames in their encoding order. The application must specify the exact input frame type for encoding. ENCODE references GOP parameters to determine when to insert information, such as an end-of-sequence, into the bitstream.

An ENCODE output consists of one frame of a bitstream with the time stamp passed from the input frame. The time stamp is used for multiplexing subsequent video with other associated data such as audio. oneVPL provides only pure video stream encoding. The application must provide its own multiplexing.

ENCODE supports the following bitrate control algorithms: constant bitrate, variable bitrate (VBR), and constant quantization parameter (QP). In the constant bitrate mode, ENCODE performs stuffing when the size of the least-compressed frame is smaller than what is required to meet the hypothetical reference decoder (HRD) buffer requirements (or VBR requirements). (Stuffing is a process that appends zeros to the end of encoded frames.)

11.1.3 Video Processing

Video processing functions (*VPP*) take raw frames as input and provide raw frames as output.

The actual conversion process is a chain operation with many single-function filters.

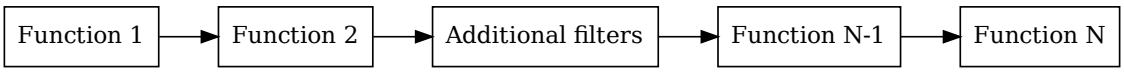


Fig. 3: Video processing operation pipeline

The application specifies the input and output format; oneVPL configures the pipeline according to the specified input and output formats. The application can also attach one or more hint structures to configure individual filters or turn them on and off. Unless specifically instructed, oneVPL builds the pipeline in a way that best utilizes hardware acceleration or generates the best video processing quality.

The *Video Processing Features table* shows oneVPL video processing features. The application can configure supported video processing features through the video processing I/O parameters. The application can also configure optional features through hints. See *Video Processing Procedures* for more details on how to configure optional filters.

Table 1: Video Processing Features

Video Processing Features	Configuration
Convert color format from input to output	I/O parameters
De-interlace to produce progressive frames at the output	I/O parameters
Crop and resize the input frames	I/O parameters
Convert input frame rate to match the output	I/O parameters
Perform inverse telecine operations	I/O parameters
Fields weaving	I/O parameters
Fields splitting	I/O parameters
Remove noise	Hint (optional feature)
Enhance picture details/edges	Hint (optional feature)
Adjust the brightness, contrast, saturation, and hue settings	Hint (optional feature)
Perform image stabilization	Hint (optional feature)
Convert input frame rate to match the output, based on frame interpolation	Hint (optional feature)
Perform detection of picture structure	Hint (optional feature)

11.1.4 Video Decoding with multiple video processing

The *DECODE_VPP* class of functions take a compressed bitstream as input, converts it to raw frames and applies video processing filters to raw frames. Users can set several output channels where each channel represents a list of video processing filters applied for decoded frames.

The *DECODE_VPP* supports only internal allocation.

11.2 Programming Guide

This chapter describes the concepts used in programming with oneVPL.

The application must use the include file `mfxvideo.h` for C/C++ programming and link the oneVPL dispatcher library `libmfx.so`.

Include these files:

```
#include "mfxvideo.h"    /* oneVPL include file */
```

Link this library:

```
libmfx.so                /* oneVPL dynamic dispatcher library (Linux\*) */
```

11.2.1 Status Codes

The oneVPL functions are organized into categories for easy reference. The categories include *ENCODE* (encoding functions), *DECODE* (decoding functions), and *VPP* (video processing functions).

Init, **Reset**, and **Close** are member functions within the ENCODE, DECODE, and VPP classes that initialize, restart, and deinitialize specific operations defined for the class. Call all member functions of a given class within the **Init - Reset - Close** sequence, except **Query** and **QueryIOSurf**. **Reset** functions are optional within the sequence.

The **Init** and **Reset** member functions set up necessary internal structures for media processing. **Init** functions allocate memory and **Reset** functions only reuse allocated internal memory. If oneVPL needs to allocate additional memory, **Reset** can fail. **Reset** functions can also fine-tune ENCODE and VPP parameters during those processes or reposition a bitstream during DECODE.

All oneVPL functions return status codes to indicate if an operation succeeded or failed. The *mfxStatus::MFX_ERR_NONE* status code indicates that the function successfully completed its operation. Error status codes are less than *mfxStatus::MFX_ERR_NONE* and warning status codes are greater than *mfxStatus::MFX_ERR_NONE*. See the *mfxStatus* enumerator for all defined status codes.

If a oneVPL function returns a warning, it has sufficiently completed its operation. Note that the output of the function might not be strictly reliable. The application must check the validity of the output generated by the function.

If a oneVPL function returns an error (except *mfxStatus::MFX_ERR_MORE_DATA*, *mfxStatus::MFX_ERR_MORE_SURFACE*, or *mfxStatus::MFX_ERR_MORE_BITSTREAM*), the function aborts the operation. The application must call either the **Reset** function to reset the class back to a clean state or the **Close** function to terminate the operation. The behavior is undefined if the application continues to call any class member functions without a **Reset** or **Close**. To avoid memory leaks, always call the **Close** function after **Init**.

11.2.2 oneVPL Session

Before calling any oneVPL functions, the application must initialize the library and create a oneVPL session. A oneVPL session maintains context for the use of any of *DECODE*, *ENCODE*, *VPP*, *DECODE_VPP* functions.

Intel® Media Software Development Kit Dispatcher (Legacy)

The *MFXInit()* or *MFXInitEx()* function starts (initializes) a session. The *MFXClose()* function closes (de-initializes) the session. To avoid memory leaks, always call *MFXClose()* after *MFXInit()*.

The application can initialize a session as a software-based session (*MFX_IMPL_SOFTWARE*) or a hardware-based session (*MFX_IMPL_HARDWARE*). In a software-based session, the SDK functions execute on a CPU. In a hardware-base session, the SDK functions use platform acceleration capabilities. For platforms that expose multiple graphic devices, the application can initialize a session on any alternative graphic device using the *MFX_IMPL_HARDWARE*, *MFX_IMPL_HARDWARE2*, *MFX_IMPL_HARDWARE3*, or *MFX_IMPL_HARDWARE4* values of *mfxIMPL*.

The application can also initialize a session to be automatic (*MFX_IMPL_AUTO* or *MFX_IMPL_AUTO_ANY*), instructing the dispatcher library to detect the platform capabilities and choose the best SDK library available. After initialization, the SDK returns the actual implementation through the *MFXQueryIMPL()* function.

Internally, the dispatcher works as follows:

1. Dispatcher searches for the shared library with the specific name:

OS	Name	Description
Linux*	libmfxsw64.so.1	64-bit software-based implementation
Linux	libmfxsw32.so.1	32-bit software-based implementation
Linux	libmfxhw64.so.1	64-bit hardware-based implementation
Linux	libmfxhw64.so.1	32-bit hardware-based implementation
Windows*	libmfxsw32.dll	64-bit software-based implementation
Windows	libmfxsw32.dll	32-bit software-based implementation
Windows	libmfxhw64.dll	64-bit hardware-based implementation
Windows	libmfxhw64.dll	32-bit hardware-based implementation

2. Once the library is loaded, the dispatcher obtains addresses for each SDK function. See the *Exported Functions/API Version table* for the list of functions to expose.

How the shared library is identified using the implementation search strategy will vary according to the OS.

- On Windows, the dispatcher searches the following locations, in the specified order, to find the correct implementation library:

1. The `Driver Store` directory for the current adapter. All types of graphics drivers can install libraries in this directory. Learn more about Driver Store.
2. The directory specified for the current hardware under the registry key `HKEY_CURRENT_USER\Software\Intel\MediaSDK\Dispatch`.
3. The directory specified for the current hardware under the registry key `HKEY_LOCAL_MACHINE\Software\Intel\MediaSDK\Dispatch`.
4. The directory that is stored in these registry keys: `C:Program FilesIntelMedia SDK`. This directory is where legacy graphics drivers install libraries.
5. The directory where the current module (the module that links the dispatcher) is located (only if the current module is a dll).

After the dispatcher completes the main search, it additionally checks:

1. The directory of the exe file of the current process, where it looks for software implementation only, regardless of which implementation the application requested.
2. Default dll search. This provides loading from the directory of the application's exe file and from the `System32` and `SysWOW64` directories. Learn more about default dll search order.
3. The `System32` and `SysWOW64` directories, which is where DCH graphics drivers install libraries.

- On Linux, the dispatcher searches the following locations, in the specified order, to find the correct implementation library:
  1. Directories provided by the environment variable `LD_LIBRARY_PATH`.
  2. Content of the `/etc/ld.so.cache` cache file.
  3. Default path `/lib`, then `/usr/lib` or `/lib64`, and then `/usr/lib64` on some 64 bit OSs. On Debian: `/usr/lib/x86_64-linux-gnu`.
  4. SDK installation folder.

oneVPL Dispatcher

The oneVPL dispatcher extends the legacy dispatcher by providing additional ability to select the appropriate implementation based on the implementation capabilities. Implementation capabilities include information about supported decoders, encoders, and VPP filters. For each supported encoder, decoder, and filter, capabilities include information about supported memory types, color formats, and image (frame) size in pixels.

The recommended approach to configure the dispatcher's capabilities search filters and to create a session based on a suitable implementation is as follows:

1. Create loader with *`MFXLoad()`*.
2. Create loader's configuration with *`MFXCreateConfig()`*.
3. Add configuration properties with *`MFXSetConfigFilterProperty()`*.
4. Explore available implementations with *`MFXEnumImplementations()`*.
5. Create a suitable session with *`MFXCreateSession()`*.

The procedure to terminate an application is as follows:

1. Destroy session with *`MFXClose()`*.
2. Destroy loader with *`MFXUnload()`*.

Note: Multiple loader instances can be created.

Note: Each loader may have multiple configuration objects associated with it. When a configuration object is modified through *MFXSetConfigFilterProperty()* it implicitly impacts the state and configuration of the associated loader.

Important: One configuration object can handle only one filter property.

Note: Multiple sessions can be created by using one loader object.

When the dispatcher searches for the implementation, it uses the following priority rules:

1. Hardware implementation has priority over software implementation.
2. General hardware implementation has priority over VSI hardware implementation.
3. Highest API version has higher priority over lower API version.

Note: Implementation has priority over the API version. In other words, the dispatcher must return the implementation with the highest API priority (greater than or equal to the implementation requested).

The dispatcher searches for the implementation in the following folders at runtime, in priority order:

1. User-defined search folders.
2. oneVPL package.
3. Path from `PATH` or `LD_LIBRARY_PATH` environmental variables, depending on OS.
4. Default system folders.
5. Standalone Intel® Media Software Development Kit package (or driver).

For more details, see the legacy dispatcher search order.

A user can develop their own implementation and direct the oneVPL dispatcher to load their implementation by providing a list of search folders. The specific steps depend on which OS is used.

- Linux: User can provide a colon separated list of folders in the `ONEVPL_SEARCH_PATH` environmental variable.
- Windows: User can provide a semicolon separated list of folders in the `ONEVPL_SEARCH_PATH` environmental variable. Alternatively, the user can use the Windows registry.

Table 2: Dispatcher Environmental Variables

Variable	Purpose
`ONEVPL_SEARCH_PATH`	List of user-defined search folders used by the dispatcher during implementation search.

The dispatcher supports different software implementations. The user can use the *mfxImplDescription::VendorID* field, the *mfxImplDescription::VendorImplID* field, or the *mfxImplDescription::ImplName* field to search for the specific implementation.

Internally, the dispatcher works as follows:

1. Dispatcher loads any shared library with *libvpl** prefix in the library name in the given search folders.
2. For each loaded library, the dispatcher tries to resolve address of the `MFXQueryImplsCapabilities()` function to collect the implementation's capabilities.
3. Once the user has requested to create the session based on this implementation, the dispatcher obtains addresses of each oneVPL function. See the *Exported Functions/API Version table* for the list of functions to export.

Note: For backward compatibility with Intel® Media Software Development Kit, the dispatcher will first try to load Intel® Media Software Development Kit if API version 1.x was requested. If loading fails, the dispatcher will search for the implementation with highest 2.x API version and load that version.

Multiple Sessions

Each oneVPL session can run exactly one instance of the DECODE, ENCODE, and VPP functions. This is adequate for a simple transcoding operation. If the application needs more than one instance of DECODE, ENCODE, or VPP in a complex transcoding setting or needs more simultaneous transcoding operations to balance CPU/GPU workloads, the application can initialize multiple oneVPL sessions. Each independent oneVPL session can be a software-based session or hardware-based session.

The application can use multiple oneVPL sessions independently or run a "joined" session. Independently operated oneVPL sessions cannot share data unless the application explicitly synchronizes session operations. This is to ensure that data is valid and complete before passing from the source to the destination session.

To join two sessions together, the application can use the function *`MFXJoinSession()`*. Alternatively, the application can use the *`MFXCloneSession()`* function to duplicate an existing session. Joined oneVPL sessions work together as a single session, sharing all session resources, threading control, and prioritization operations except hardware acceleration devices and external allocators. When joined, the first session (first join) serves as the parent session and will schedule execution resources with all other child sessions. Child sessions rely on the parent session for resource management.

With joined sessions, the application can set the priority of session operations through the *`MFXSetPriority()`* function. A lower priority session receives fewer CPU cycles. Session priority does not affect hardware accelerated processing.

After the completion of all session operations, the application can use the *`MFXDisjoinSession()`* function to remove the joined state of a session. Do not close the parent session until all child sessions are disjoined or closed.

11.2.3 Frame and Fields

In oneVPL terminology, a frame (also referred to as frame surface) contains either a progressive frame or a complementary field pair. If the frame is a complementary field pair, the odd lines of the surface buffer store the top fields and the even lines of the surface buffer store the bottom fields.

Frame Surface Management

During encoding, decoding, or video processing, cases arise that require reserving input or output frames for future use. For example, when decoding, a frame that is ready for output must remain as a reference frame until the current sequence pattern ends. The usual method to manage this is to cache the frames internally. This method requires a copy operation, which can significantly reduce performance.

oneVPL has two approaches to avoid the need for copy operations. The legacy approach uses a frame-locking mechanism that works as follows:

1. The application allocates a pool of frame surfaces large enough to include oneVPL function I/O frame surfaces and internal cache needs. Each frame surface maintains a `Locked` counter, which is part of the *mfxFrameData* structure. The `Locked` counter is initially set to zero.
2. The application calls a oneVPL function with frame surfaces from the pool whose `Locked` counter is set as appropriate for the desired operation. For decoding or video processing operations, where oneVPL uses the surfaces to write, the `Locked` counter should be equal to zero. If the oneVPL function needs to reserve any frame surface, the oneVPL function increases the `Locked` counter of the frame surface. A non-zero `Locked` counter indicates that the calling application must treat the frame surface as "in use." When the frame surface is in use, the application can read but cannot alter, move, delete, or free the frame surface.
3. In subsequent oneVPL executions, if the frame surface is no longer in use, oneVPL decreases the `Locked` counter. When the `Locked` counter reaches zero, the application is free to do as it wishes with the frame surface.

In general, the application should not increase or decrease the `Locked` counter since oneVPL manages this field. If, for some reason, the application needs to modify the `Locked` counter, the operation must be atomic to avoid a race condition.

oneVPL API version 2.0 introduces the *mfxFrameSurfaceInterface* structure which provides a set of callback functions for the *mfxFrameSurface1* structure to work with frame surfaces. This interface defines *mfxFrameSurface1* as a reference counted object which can be allocated by oneVPL or the application. The application must follow the general rules of operation with reference counted objects. For example, when surfaces are allocated by oneVPL during *MFXVideoDECODE_DecodeFrameAsync()* or with the help of *MFXMemory_GetSurfaceForVPP()* or *MFXMemory_GetSurfaceForVPPOut()* or *MFXMemory_GetSurfaceForEncode()*, the application must call the corresponding *mfxFrameSurfaceInterface::Release* function for the surfaces that are no longer in use.

> **Attention:** Note that the `Locked` counter defines read/write access policies and the reference counter is responsible for managing a frame's lifetime.

The second approach to avoid the need for copy operations is based on the *mfxFrameSurfaceInterface* and works as follows:

1. oneVPL or the application allocates a frame surface and the application stores a value of reference counter obtained through *mfxFrameSurfaceInterface::GetRefCounter*.
2. The application calls a oneVPL function with the frame surface. If oneVPL needs to reserve the frame surface it increments the reference counter through the *mfxFrameSurfaceInterface::AddRef* call. When the frame surface is no longer in use by the oneVPL it decrements reference counter through the *mfxFrameSurfaceInterface::Release* call which returns the reference counter to the original value.
3. The application checks the reference counter of the frame surface and when it is equal to the original value after allocation, it can reuse the reference counter for subsequent operations.

Note: All *mfxFrameSurface1* structures starting from mfxFrameSurface1::mfxStructVersion = {1,1} support the *mfxFrameSurfaceInterface*.

11.2.4 Decoding Procedures

There are several approaches to decode video frames. The first one is based on the internal allocation mechanism presented here:

```
MFXVideoDECODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
      append_more_bitstream(bitstream);
   bits=(end_of_stream())?NULL:bitstream;
   sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,NULL,&disp,&syncp);
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // skipped other error handling
   if (sts==MFX_ERR_NONE) {
      disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_
↪SyncOperation(session,syncp,INFINITE)
      do_something_with_decoded_frame(disp);
      disp->FrameInterface->Release(disp);
   }
}
MFXVideoDECODE_Close(session);
```

Note the following key points about the example:

- The application calls the *MFXVideoDECODE_DecodeFrameAsync()* function for a decoding operation with the bitstream buffer (bits), frame surface is allocated internally by the library.

  Attention: As shown in the example above starting with API version 2.0, the application can provide NULL as the working frame surface that leads to internal memory allocation.

- If decoding output is not available, the function returns a status code requesting additional bitstream input as follows:
  - *mfxStatus::MFX_ERR_MORE_DATA*: The function needs additional bitstream input. The existing buffer contains less than a frame's worth of bitstream data.
- Upon successful decoding, the *MFXVideoDECODE_DecodeFrameAsync()* function returns *mfxStatus::MFX_ERR_NONE*. However, the decoded frame data (identified by the surface_out pointer) is not yet available because the *MFXVideoDECODE_DecodeFrameAsync()* function is asynchronous. The application must use the *MFXVideoCORE_SyncOperation()* or *mfxFrameSurfaceInterface::Synchronize* to synchronize the decoding operation before retrieving the decoded frame data.
- At the end of the bitstream, the application continuously calls the *MFXVideoDECODE_DecodeFrameAsync()* function with a NULL bitstream pointer to drain any remaining frames cached within the decoder until the function returns *mfxStatus::MFX_ERR_MORE_DATA*.
- When application completes the work with frame surface, it must call release to avoid memory leaks.

The next example demonstrates how applications can use internally pre-allocated chunk of video surfaces:

```
MFXVideoDECODE_QueryIOSurf(session, &init_param, &request);
MFXVideoDECODE_Init(session, &init_param);
for (int i = 0; i < request.NumFrameSuggested; i++) {
    MFXMemory_GetSurfaceForDecode(session, &work);
    add_surface_to_pool(work);
}
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
      append_more_bitstream(bitstream);
   bits=(end_of_stream())?NULL:bitstream;
   find_free_surface_from_the_pool(&work);
   sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,work,&disp,&syncp);
   work->FrameInterface->Release(work);
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // skipped other error handling
   if (sts==MFX_ERR_NONE) {
      disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_
↪SyncOperation(session,syncp,INFINITE)
      do_something_with_decoded_frame(disp);
      disp->FrameInterface->Release(disp);
   }
}
MFXVideoDECODE_Close(session);
```

Here the application should use the *MFXVideoDECODE_QueryIOSurf()* function to obtain the number of working frame surfaces required to reorder output frames. It is also required that *MFXMemory_GetSurfaceForDecode()* call is done after decoder initialization. In the *MFXVideoDECODE_DecodeFrameAsync()* the oneVPL library increments reference counter of incoming surface frame so it is required that the application releases frame surface after the call.

The following pseudo code shows the decoding procedure according to the legacy mode (API version < 2.0) with external video frames allocation:

```
MFXVideoDECODE_DecodeHeader(session, bitstream, &init_param);
MFXVideoDECODE_QueryIOSurf(session, &init_param, &request);
allocate_pool_of_frame_surfaces(request.NumFrameSuggested);
MFXVideoDECODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
      append_more_bitstream(bitstream);
   find_free_surface_from_the_pool(&work);
   bits=(end_of_stream())?NULL:bitstream;
   sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,work,&disp,&syncp);
   if (sts==MFX_ERR_MORE_SURFACE) continue;
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   if (sts==MFX_ERR_REALLOC_SURFACE) {
      MFXVideoDECODE_GetVideoParam(session, &param);
      realloc_surface(work, param.mfx.FrameInfo);
      continue;
   }
   // skipped other error handling
   if (sts==MFX_ERR_NONE) {
      disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_
↪SyncOperation(session,syncp,INFINITE)
      do_something_with_decoded_frame(disp);
```

(continues on next page)

(continued from previous page)

```
    }
}
MFXVideoDECODE_Close(session);
free_pool_of_frame_surfaces();
```

Note the following key points about the example:

- The application can use the *MFXVideoDECODE_DecodeHeader()* function to retrieve decoding initialization parameters from the bitstream. This step is optional if the data is retrievable from other sources such as an audio/video splitter.
- The *MFXVideoDECODE_DecodeFrameAsync()* function can return following status codes in addition to the described above:
  - *mfxStatus::MFX_ERR_MORE_SURFACE*: The function needs one more frame surface to produce any output.
  - *mfxStatus::MFX_ERR_REALLOC_SURFACE*: Dynamic resolution change case - the function needs a bigger working frame surface (work).

The following pseudo code shows the simplified decoding procedure:

```
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
      append_more_bitstream(bitstream);
   bits=(end_of_stream())?NULL:bitstream;
   sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,NULL,&disp,&syncp);
   if (sts==MFX_ERR_MORE_SURFACE) continue;
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // skipped other error handling
   if (sts==MFX_ERR_NONE) {
      disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_
↪SyncOperation(session,syncp,INFINITE)
      do_something_with_decoded_frame(disp);
      disp->FrameInterface->Release(disp);
   }
}
```

oneVPL API version 2.0 introduces a new decoding approach. For simple use cases, when the user wants to decode a stream and does not want to set additional parameters, a simplified procedure for the decoder's initialization has been proposed. In this scenario it is possible to skip explicit stages of a stream's header decoding and the decoder's initialization and instead to perform these steps implicitly during decoding of the first frame. This change also requires setting the additional field *mfxBitstream::CodecId* to indicate codec type. In this mode the decoder allocates *mfxFrameSurface1* internally, so users should set the input surface to zero.

Bitstream Repositioning

The application can use the following procedure for bitstream reposition during decoding:

1. Use the *MFXVideoDECODE_Reset()* function to reset the oneVPL decoder.
2. Optional: If the application maintains a sequence header that correctly decodes the bitstream at the new position, the application may insert the sequence header to the bitstream buffer.
3. Append the bitstream from the new location to the bitstream buffer.

4. Resume the decoding procedure. If the sequence header is not inserted in the previous steps, the oneVPL decoder searches for a new sequence header before starting decoding.

Broken Streams Handling

Robustness and the capability to handle a broken input stream is an important part of the decoder.

First, the start code prefix (ITU-T* H.264 3.148 and ITU-T H.265 3.142) is used to separate NAL units. Then all syntax elements in the bitstream are parsed and verified. If any of the elements violate the specification, the input bitstream is considered invalid and the decoder tries to re-sync (find the next start code). Subsequent decoder behavior is dependent on which syntax element is broken:

- SPS header is broken: return *mfxStatus::MFX_ERR_INCOMPATIBLE_VIDEO_PARAM* (HEVC decoder only, AVC decoder uses last valid).
- PPS header is broken: re-sync, use last valid PPS for decoding.
- Slice header is broken: skip this slice, re-sync.
- Slice data is broken: corruption flags are set on output surface.

Many streams have IDR frames with `frame_num != 0` while the specification says that "If the current picture is an IDR picture, frame_num shall be equal to 0" (ITU-T H.265 7.4.3).

VUI is also validated, but errors do not invalidate the whole SPS. The decoder either does not use the corrupted VUI (AVC) or resets incorrect values to default (HEVC).

Note: Some requirements are relaxed because there are many streams which violate the strict standard but can be decoded without errors.

Corruption at the reference frame is spread over all inter-coded pictures that use the reference frame for prediction. To cope with this problem you must either periodically insert I-frames (intra-coded) or use the intra-refresh technique. The intra-refresh technique allows recovery from corruptions within a predefined time interval. The main point of intra-refresh is to insert a cyclic intra-coded pattern (usually a row) of macroblocks into the inter-coded pictures, restricting motion vectors accordingly. Intra-refresh is often used in combination with recovery point SEI, where the `recovery_frame_cnt` is derived from the intra-refresh interval. The recovery point SEI message is well described at ITU-T H.264 D.2.7 and ITU-T H.265 D.2.8. If decoding starts from AU associated with this SEI message, then the message can be used by the decoder to determine from which picture all subsequent pictures have no errors. In comparison to IDR, the recovery point message does not mark reference pictures as "unused for reference".

Besides validation of syntax elements and their constraints, the decoder also uses various hints to handle broken streams:

- If there are no valid slices for the current frame, then the whole frame is skipped.
- The slices which violate slice segment header semantics (ITU-T H.265 7.4.7.1) are skipped. Only the `slice_temporal_mvp_enabled_flag` is checked for now.
- Since LTR (Long Term Reference) stays at DPB until it is explicitly cleared by IDR or MMCO, the incorrect LTR could cause long standing visual artifacts. AVC decoder uses the following approaches to handle this:
  - When there is a DPB overflow in the case of an incorrect MMCO command that marks the reference picture as LT, the operation is rolled back.
  - An IDR frame with `frame_num != 0` can't be LTR.
- If the decoder detects frame gapping, it inserts "fake"" (marked as non-existing) frames, updates FrameNumWrap (ITU-T H.264 8.2.4.1) for reference frames, and applies the Sliding Window (ITU-T H.264

8.2.5.3) marking process. Fake frames are marked as reference, but since they are marked as non-existing, they are not used for inter-prediction.

VP8 Specific Details

Unlike other oneVPL supported decoders, VP8 can accept only a complete frame as input. The application should provide the complete frame accompanied by the *MFX_BITSTREAM_COMPLETE_FRAME* flag. This is the single specific difference.

JPEG

The application can use the same decoding procedures for JPEG/motion JPEG decoding, as shown in the following pseudo code:

```
// optional; retrieve initialization parameters
MFXVideoDECODE_DecodeHeader(...);
// decoder initialization
MFXVideoDECODE_Init(...);
// single frame/picture decoding
MFXVideoDECODE_DecodeFrameAsync(...);
MFXVideoCORE_SyncOperation(...);
// optional; retrieve meta-data
MFXVideoDECODE_GetUserData(...);
// close
MFXVideoDECODE_Close(...);
```

The *MFXVideoDECODE_Query()* function will return *mfxStatus::MFX_ERR_UNSUPPORTED* if the input bitstream contains unsupported features.

For still picture JPEG decoding, the input can be any JPEG bitstreams that conform to the ITU-T Recommendation T.81 with an EXIF or JFIF header. For motion JPEG decoding, the input can be any JPEG bitstreams that conform to the ITU-T Recommendation T.81.

Unlike other oneVPL decoders, JPEG decoding supports three different output color formats: *NV12*, *YUY2*, and *RGB32*. This support sometimes requires internal color conversion and more complicated initialization. The color format of the input bitstream is described by the *mfxInfoMFX::JPEGChromaFormat* and *mfxInfoMFX::JPEGColorFormat* fields.. The *MFXVideoDECODE_DecodeHeader()* function usually fills them in. If the JPEG bitstream does not contains color format information, the application should provide it. Output color format is described by general oneVPL parameters: the *mfxFrameInfo::FourCC* and *mfxFrameInfo::ChromaFormat* fields.

Motion JPEG supports interlaced content by compressing each field (a half-height frame) individually. This behavior is incompatible with the rest of the oneVPL transcoding pipeline, where oneVPL requires fields to be in odd and even lines of the same frame surface. The decoding procedure is modified as follows:

- The application calls the *MFXVideoDECODE_DecodeHeader()* function with the first field JPEG bitstream to retrieve initialization parameters.
- The application initializes the oneVPL JPEG decoder with the following settings:
  - The `PicStruct` field of the *mfxVideoParam* structure set to the correct interlaced type, *MFX_PICSTRUCT_FIELD_TFF* or *MFX_PICSTRUCT_FIELD_BFF*, from the motion JPEG header.
  - Double the `Height` field in the *mfxVideoParam* structure as the value returned by the *MFXVideoDECODE_DecodeHeader()* function describes only the first field. The actual frame surface should contain both fields.

- During decoding, the application sends both fields for decoding in the same *mfxBitstream*. The application should also set *mfxBitstream::DataFlag* to *MFX_BITSTREAM_COMPLETE_FRAME*. oneVPL decodes both fields and combines them into odd and even lines according to oneVPL convention.

By default, the *MFXVideoDECODE_DecodeHeader()* function returns the `Rotation` parameter so that after rotation, the pixel at the first row and first column is at the top left. The application can overwrite the default rotation before calling *MFXVideoDECODE_Init()*.

The application may specify Huffman and quantization tables during decoder initialization by attaching *mfxExtJPEGQuantTables* and *mfxExtJPEGHuffmanTables* buffers to the *mfxVideoParam* structure. In this case, the decoder ignores tables from bitstream and uses the tables specified by the application. The application can also retrieve these tables by attaching the same buffers to *mfxVideoParam* and calling *MFXVideoDECODE_GetVideoParam()* or *MFXVideoDECODE_DecodeHeader()* functions.

Multi-view Video Decoding

The oneVPL MVC decoder operates on complete MVC streams that contain all view and temporal configurations. The application can configure the oneVPL decoder to generate a subset at the decoding output. To do this, the application must understand the stream structure and use the stream information to configure the decoder for target views.

The decoder initialization procedure is as follows:

1. The application calls the *MFXVideoDECODE_DecodeHeader()* function to obtain the stream structural information. This is done in two steps:
    1. The application calls the *MFXVideoDECODE_DecodeHeader()* function with the *mfxExtMVCSeqDesc* structure attached to the *mfxVideoParam* structure. At this point, do not allocate memory for the arrays in the *mfxExtMVCSeqDesc* structure. Set the `View`, `ViewId`, and `OP` pointers to NULL and set `NumViewAlloc`, `NumViewIdAlloc`, and `NumOPAlloc` to zero. The function parses the bitstream and returns *mfxStatus::MFX_ERR_NOT_ENOUGH_BUFFER* with the correct values for `NumView`, `NumViewId`, and `NumOP`. This step can be skipped if the application is able to obtain the `NumView`, `NumViewId`, and `NumOP` values from other sources.
    2. The application allocates memory for the `View`, `ViewId`, and `OP` arrays and calls the *MFXVideoDECODE_DecodeHeader()* function again. The function returns the MVC structural information in the allocated arrays.
2. The application fills the *mfxExtMVCTargetViews* structure to choose the target views, based on information described in the *mfxExtMVCSeqDesc* structure.
3. The application initializes the oneVPL decoder using the *MFXVideoDECODE_Init()* function. The application must attach both the *mfxExtMVCSeqDesc* structure and the *mfxExtMVCTargetViews* structure to the *mfxVideoParam* structure.

In the above steps, do not modify the values of the *mfxExtMVCSeqDesc* structure after the *MFXVideoDECODE_DecodeHeader()* function, as the oneVPL decoder uses the values in the structure for internal memory allocation. Once the application configures the oneVPL decoder, the rest of the decoding procedure remains unchanged. As shown in the pseudo code below, the application calls the *MFXVideoDECODE_DecodeFrameAsync()* function multiple times to obtain all target views of the current frame picture, one target view at a time. The target view is identified by the `FrameID` field of the *mfxFrameInfo* structure.

```
mfxExtBuffer *eb[2];
mfxExtMVCSeqDesc  seq_desc;
mfxVideoParam init_param;

init_param.ExtParam=(mfxExtBuffer **)&eb;
init_param.NumExtParam=1;
```

(continues on next page)

(continued from previous page)

```
eb[0]=(mfxExtBuffer *)&seq_desc;
MFXVideoDECODE_DecodeHeader(session, bitstream, &init_param);

/* select views to decode */
mfxExtMVCTargetViews tv;
init_param.NumExtParam=2;
eb[1]=(mfxExtBuffer *)&tv;

/* initialize decoder */
MFXVideoDECODE_Init(session, &init_param);

/* perform decoding */
for (;;) {
    MFXVideoDECODE_DecodeFrameAsync(session, bits, work, &disp, &syncp);
    disp->FrameInterface->Synchronize(disp, INFINITE); // or MFXVideoCORE_
↪SyncOperation(session,syncp,INFINITE)
}

/* close decoder */
MFXVideoDECODE_Close(session);
```

Combined Decode with Multi-channel Video Processing

The oneVPL exposes interface for making decode and video processing operations in one call. Users can specify a number of output processing channels and multiple video filters per each channel. This interface supports only internal memory allocation model and returns array of processed frames through *mfxSurfaceArray* reference object as shown by the example:

```
num_channel_par = 2;
// first video processing channel with resize
vpp_par_array[0]->VPP.Width = 400;
vpp_par_array[0]->VPP.Height = 400;

// second video channel with color conversion filter
vpp_par_array[1]->VPP.FourCC = MFX_FOURCC_UYVY;

sts = MFXVideoDECODE_VPP_Init(session, decode_par, vpp_par_array, num_channel_par);

sts = MFXVideoDECODE_VPP_DecodeFrameAsync(session, bitstream, NULL, 0, &surf_array_
↪out);

//surf_array_out layout is
surf_array_out->Surfaces[0]; //The first channel which contains original decoded␣
↪frames
surf_array_out->Surfaces[1]; //The second channel contains resized processed frames␣
↪after decode.
surf_array_out->Surfaces[2]; //The third  channel contains color converted frames␣
↪after decode.
```

It's possible that different video processing channels may have diffrent latency:

```
//1st call
sts = MFXVideoDECODE_VPP_DecodeFrameAsync(session, bitstream, NULL, 0, &surf_array_
↪out);
//surf_array_out layout is
```

(continues on next page)

(continued from previous page)

```
surf_array_out->Surfaces[0]; //decoded frame
surf_array_out->Surfaces[1]; //resized frame (ChannelId = 1). The first frame from channel wit resize availble
// no output from channel with ADI output since it has one frame delay

//2nd call
sts = MFXVideoDECODE_VPP_DecodeFrameAsync(session, bitstream, NULL, 0, &surf_array_out);
//surf_array_out layout is
surf_array_out->Surfaces[0]; //decoded frame
surf_array_out->Surfaces[1]; //resized frame (ChannelId = 1)
surf_array_out->Surfaces[2]; //ADI output (ChannelId = 2). The first frame from ADI channel
```

Application can match decoded frame w/ specific VPP channels using *mfxFrameData::TimeStamp*, :cpp:member:mfxFrameData::FrameOrder` and *mfxFrameInfo::ChannelId*.

Application can skip some or all channels including decoding output with help of *skip_channels* and *num_skip_channels* parameters as follows: application fills *skip_channels* array with *ChannelId`s to disable output of correspondent channels. In that case :cpp:member: `surf_array_out* would contain only surfaces for the remaining channels. If the decoder's channel and/or impacted VPP channels don't have output frame(s) for the current call (for instance, input bitstream doesn't contain complete frame or deinterlacing/FRC filter have delay) *skip_channels* parameter is ignored for this channel. If application disables all channels the SDK returns NULL as *mfxSurfaceArray*. If application doesn't need to disable any channels it sets *num_skip_channels* to zero, *skip_channels* is ignored when *num_skip_channels* is zero.

Note: Even if more than one input compressed frame is consumed, the `MFXVideoDECODE_VPP_DecodeFrameAsync()` produces only one decoded frame and correspondent frames from VPP channels.

11.2.5 Encoding Procedures

There are two methods for shared memory allocation and handling in oneVPL: external and internal.

External Memory

The following pseudo code shows the encoding procedure with external memory (legacy mode):

```
MFXVideoENCODE_QueryIOSurf(session, &init_param, &request);
allocate_pool_of_frame_surfaces(request.NumFrameSuggested);
MFXVideoENCODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream()) {
      find_unlocked_surface_from_the_pool(&surface);
      fill_content_for_encoding(surface);
   }
   surface2=end_of_stream()?NULL:surface;
   sts=MFXVideoENCODE_EncodeFrameAsync(session,NULL,surface2,bits,&syncp);
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // Skipped other error handling
   if (sts==MFX_ERR_NONE) {
```

(continues on next page)

(continued from previous page)

```
      MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
      do_something_with_encoded_bits(bits);
   }
}
MFXVideoENCODE_Close(session);
free_pool_of_frame_surfaces();
```

Note the following key points about the example:

- The application uses the *MFXVideoENCODE_QueryIOSurf()* function to obtain the number of working frame surfaces required for reordering input frames.
- The application calls the *MFXVideoENCODE_EncodeFrameAsync()* function for the encoding operation. The input frame must be in an unlocked frame surface from the frame surface pool. If the encoding output is not available, the function returns the *mfxStatus::MFX_ERR_MORE_DATA* status code to request additional input frames.
- Upon successful encoding, the *MFXVideoENCODE_EncodeFrameAsync()* function returns *mfxStatus::MFX_ERR_NONE*. At this point, the encoded bitstream is not yet available because the *MFXVideoENCODE_EncodeFrameAsync()* function is asynchronous. The application must use the *MFXVideoCORE_SyncOperation()* function to synchronize the encoding operation before retrieving the encoded bitstream.
- At the end of the stream, the application continuously calls the *MFXVideoENCODE_EncodeFrameAsync()* function with a NULL surface pointer to drain any remaining bitstreams cached within the oneVPL encoder, until the function returns *mfxStatus::MFX_ERR_MORE_DATA*.

Note: It is the application's responsibility to fill pixels outside of the crop window when it is smaller than the frame to be encoded, especially in cases when crops are not aligned to minimum coding block size (16 for AVC and 8 for HEVC and VP9).

Internal Memory

The following pseudo code shows the encoding procedure with internal memory:

```
MFXVideoENCODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
   if (sts==MFX_ERR_MORE_DATA && !end_of_stream()) {
      MFXMemory_GetSurfaceForEncode(session,&surface);
      fill_content_for_encoding(surface);
   }
   surface2=end_of_stream()?NULL:surface;
   sts=MFXVideoENCODE_EncodeFrameAsync(session,NULL,surface2,bits,&syncp);
   if (surface2) surface->FrameInterface->Release(surface2);
   if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
   // Skipped other error handling
   if (sts==MFX_ERR_NONE) {
      MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
      do_something_with_encoded_bits(bits);
   }
}
MFXVideoENCODE_Close(session);
```

There are several key differences in this example, compared to external memory (legacy mode):

- The application does not need to call the *MFXVideoENCODE_QueryIOSurf()* function to obtain the number of working frame surfaces since allocation is done by oneVPL.
- The application calls the *MFXMemory_GetSurfaceForEncode()* function to get a free surface for the subsequent encode operation.
- The application must call the *mfxFrameSurfaceInterface::Release* function to decrement the reference counter of the obtained surface after the call to the *MFXVideoENCODE_EncodeFrameAsync()* function.

Configuration Change

The application changes configuration during encoding by calling the *MFXVideoENCODE_Reset()* function. Depending on the difference in configuration parameters before and after the change, the oneVPL encoder will either continue the current sequence or start a new one. If the encoder starts a new sequence, it completely resets internal state and begins a new sequence with the IDR frame.

The application controls encoder behavior during parameter change by attaching the *mfxExtEncoderResetOption* structure to the *mfxVideoParam* structure during reset. By using this structure, the application instructs the encoder to start or not start a new sequence after reset. In some cases, the request to continue the current sequence cannot be satisfied and the encoder will fail during reset. To avoid this scenario, the application may query the reset outcome before the actual reset by calling the *MFXVideoENCODE_Query()* function with the *mfxExtEncoderResetOption* attached to the *mfxVideoParam* structure.

The application uses the following procedure to change encoding configurations:

1. The application retrieves any cached frames in the oneVPL encoder by calling the *MFXVideoENCODE_EncodeFrameAsync()* function with a NULL input frame pointer until the function returns *mfxStatus::MFX_ERR_MORE_DATA*.
2. The application calls the *MFXVideoENCODE_Reset()* function with the new configuration:
   - If the function successfully sets the configuration, the application can continue encoding as usual.
   - If the new configuration requires a new memory allocation, the function returns *mfxStatus::MFX_ERR_INCOMPATIBLE_VIDEO_PARAM*. The application must close the oneVPL encoder and reinitialize the encoding procedure with the new configuration.

External Bitrate Control

The application can make the encoder use the external Bitrate Control (BRC) instead of the native bitrate control. To make the encoder use the external BRC, the application should attach the *mfxExtCodingOption2* structure with `ExtBRC = MFX_CODINGOPTION_ON` and the *mfxExtBRC* callback structure to the *mfxVideoParam* structure during encoder initialization. The **Init**, **Reset**, and **Close** callbacks will be invoked inside their corresponding functions: *MFXVideoENCODE_Init()*, *MFXVideoENCODE_Reset()*, and *MFXVideoENCODE_Close()*. The following figure shows asynchronous encoding flow with external BRC (using `GetFrameCtrl` and `Update`):

Note: `IntAsyncDepth` is the oneVPL max internal asynchronous encoding queue size. It is always less than or equal to *mfxVideoParam::AsyncDepth*.

The following pseudo code shows use of the external BRC:

```
1  #include "mfxvideo.h"
2  #include "mfxbrc.h"
3
```

(continues on next page)

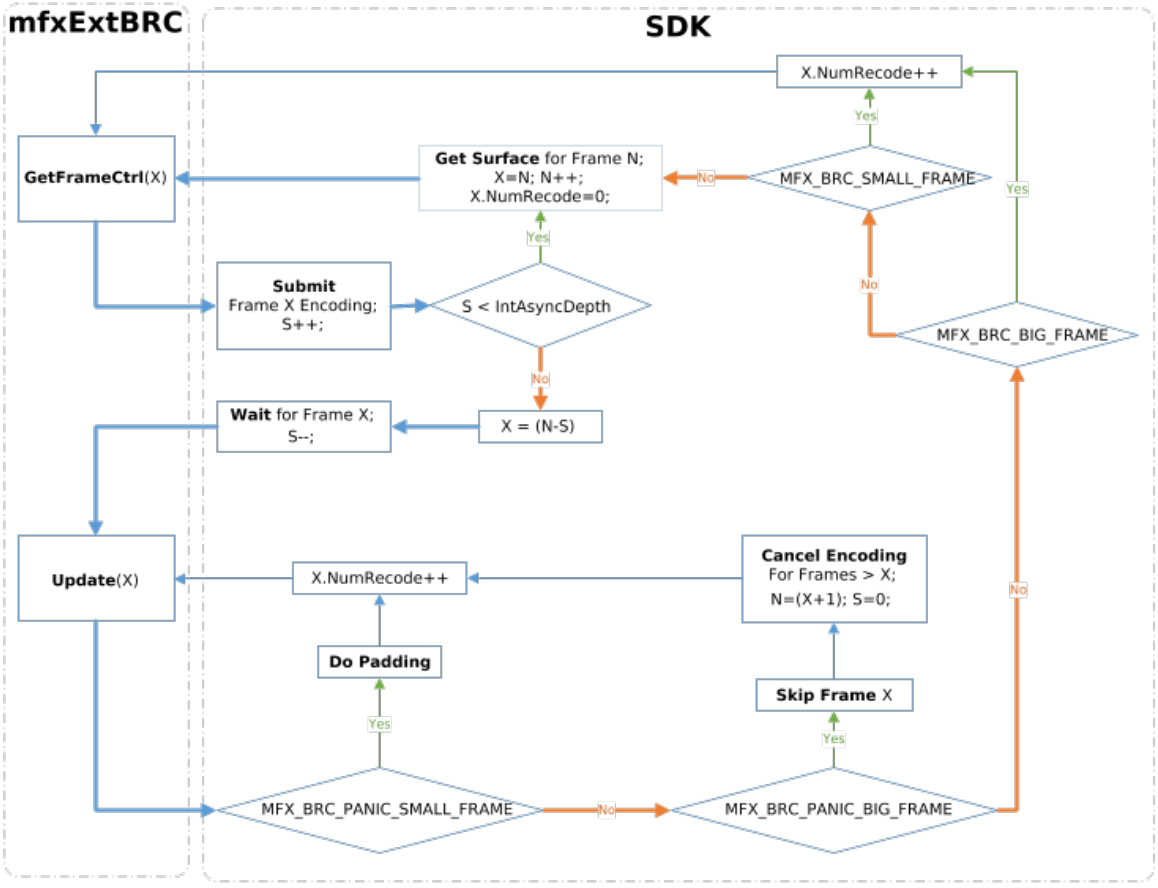


Fig. 4: Asynchronous encoding flow with external BRC

(continued from previous page)

```
   typedef struct {
      mfxU32 EncodedOrder;
      mfxI32 QP;
      mfxU32 MaxSize;
      mfxU32 MinSize;
      mfxU16 Status;
      mfxU64 StartTime;
      // ... skipped
   } MyBrcFrame;

   typedef struct {
      MyBrcFrame* frame_queue;
      mfxU32 frame_queue_size;
      mfxU32 frame_queue_max_size;
      mfxI32 max_qp[3]; //I,P,B
      mfxI32 min_qp[3]; //I,P,B
      // ... skipped
   } MyBrcContext;

   void* GetExtBuffer(mfxExtBuffer** ExtParam, mfxU16 NumExtParam, mfxU32 bufferID)
   {
       int i=0;
       for(i = 0; i < NumExtParam; i++) {
           if(ExtParam[i]->BufferId == bufferID) return ExtParam[i];
       }
       return NULL;
   }

   static int IsParametersSupported(mfxVideoParam *par)
   {
       // do some checks
       return 1;
   }

   static int IsResetPossible(MyBrcContext* ctx, mfxVideoParam *par)
   {
       // do some checks
       return 1;
   }

   static MyBrcFrame* GetFrame(MyBrcFrame *frame_queue, mfxU32 frame_queue_size,␣
↪mfxU32 EncodedOrder)
   {
       //do some logic
       if(frame_queue_size) return &frame_queue[0];
       return NULL;
   }

   static mfxU32 GetFrameCost(mfxU16 FrameType, mfxU16 PyramidLayer)
   {
       // calculate cost
       return 1;
   }

   static mfxU32 GetMinSize(MyBrcContext *ctx, mfxU32 cost)
   {
       // do some logic
```

(continues on next page)

(continued from previous page)

```
        return 1;
    }

    static mfxU32 GetMaxSize(MyBrcContext *ctx, mfxU32 cost)
    {
        // do some logic
        return 1;
    }

    static mfxI32 GetInitQP(MyBrcContext *ctx, mfxU32 MinSize, mfxU32 MaxSize, mfxU32 cost)
    {
        // do some logic
        return 1;
    }

    static mfxU64 GetTime()
    {
        mfxU64 wallClock;
        return wallClock;
    }

    static void UpdateBRCState(mfxU32 CodedFrameSize, MyBrcContext *ctx)
    {
        return;
    }

    static void RemoveFromQueue(MyBrcFrame* frame_queue, mfxU32 frame_queue_size, MyBrcFrame* frame)
    {
        return;
    }

    static mfxU64 GetMaxFrameEncodingTime(MyBrcContext *ctx)
    {
        return 2;
    }

    mfxStatus MyBrcInit(mfxHDL pthis, mfxVideoParam* par) {
       MyBrcContext* ctx = (MyBrcContext*)pthis;
       mfxI32 QpBdOffset;
       mfxExtCodingOption2* co2;
       mfxI32 defaultQP;

       if (!pthis || !par)
          return MFX_ERR_NULL_PTR;

       if (!IsParametersSupported(par))
          return MFX_ERR_UNSUPPORTED;

       ctx->frame_queue_max_size = par->AsyncDepth;
       ctx->frame_queue = (MyBrcFrame*)malloc(sizeof(MyBrcFrame) * ctx->frame_queue_max_size);

       if (!ctx->frame_queue)
          return MFX_ERR_MEMORY_ALLOC;

```

(continues on next page)

(continued from previous page)

```
114      co2 = (mfxExtCodingOption2*)GetExtBuffer(par->ExtParam, par->NumExtParam, MFX_
↪EXTBUFF_CODING_OPTION2);
115      QpBdOffset = (par->mfx.FrameInfo.BitDepthLuma > 8) ? (6 * (par->mfx.FrameInfo.
↪BitDepthLuma - 8)) : 0;
116
117      ctx->max_qp[0] = (co2 && co2->MaxQPI) ? (co2->MaxQPI - QpBdOffset) : defaultQP;
118      ctx->min_qp[0] = (co2 && co2->MinQPI) ? (co2->MinQPI - QpBdOffset) : defaultQP;
119
120      ctx->max_qp[1] = (co2 && co2->MaxQPP) ? (co2->MaxQPP - QpBdOffset) : defaultQP;
121      ctx->min_qp[1] = (co2 && co2->MinQPP) ? (co2->MinQPP - QpBdOffset) : defaultQP;
122
123      ctx->max_qp[2] = (co2 && co2->MaxQPB) ? (co2->MaxQPB - QpBdOffset) : defaultQP;
124      ctx->min_qp[2] = (co2 && co2->MinQPB) ? (co2->MinQPB - QpBdOffset) : defaultQP;
125
126      // skipped initialization of other other BRC parameters
127
128      ctx->frame_queue_size = 0;
129
130      return MFX_ERR_NONE;
131   }
132
133   mfxStatus MyBrcReset(mfxHDL pthis, mfxVideoParam* par) {
134      MyBrcContext* ctx = (MyBrcContext*)pthis;
135
136      if (!pthis || !par)
137         return MFX_ERR_NULL_PTR;
138
139      if (!IsParametersSupported(par))
140         return MFX_ERR_UNSUPPORTED;
141
142      if (!IsResetPossible(ctx, par))
143         return MFX_ERR_INCOMPATIBLE_VIDEO_PARAM;
144
145      // reset here BRC parameters if required
146
147      return MFX_ERR_NONE;
148   }
149
150   mfxStatus MyBrcClose(mfxHDL pthis) {
151      MyBrcContext* ctx = (MyBrcContext*)pthis;
152
153      if (!pthis)
154         return MFX_ERR_NULL_PTR;
155
156      if (ctx->frame_queue) {
157         free(ctx->frame_queue);
158         ctx->frame_queue = NULL;
159         ctx->frame_queue_max_size = 0;
160         ctx->frame_queue_size = 0;
161      }
162
163      return MFX_ERR_NONE;
164   }
165
166   mfxStatus MyBrcGetFrameCtrl(mfxHDL pthis, mfxBRCFrameParam* par, mfxBRCFrameCtrl*␣
↪ctrl) {
167      MyBrcContext* ctx = (MyBrcContext*)pthis;
```

(continues on next page)

(continued from previous page)

```
      MyBrcFrame* frame = NULL;
      mfxU32 cost;

      if (!pthis || !par || !ctrl)
         return MFX_ERR_NULL_PTR;

      if (par->NumRecode > 0)
         frame = GetFrame(ctx->frame_queue, ctx->frame_queue_size, par->EncodedOrder);
      else if (ctx->frame_queue_size < ctx->frame_queue_max_size)
         frame = &ctx->frame_queue[ctx->frame_queue_size++];

      if (!frame)
         return MFX_ERR_UNDEFINED_BEHAVIOR;

      if (par->NumRecode == 0) {
         frame->EncodedOrder = par->EncodedOrder;
         cost = GetFrameCost(par->FrameType, par->PyramidLayer);
         frame->MinSize = GetMinSize(ctx, cost);
         frame->MaxSize = GetMaxSize(ctx, cost);
         frame->QP = GetInitQP(ctx, frame->MinSize, frame->MaxSize, cost); // from QP/
↪size stat
         frame->StartTime = GetTime();
      }

      ctrl->QpY = frame->QP;

      return MFX_ERR_NONE;
   }

   #define DEFAULT_QP_INC 4
   #define DEFAULT_QP_DEC 4

   mfxStatus MyBrcUpdate(mfxHDL pthis, mfxBRCFrameParam* par, mfxBRCFrameCtrl* ctrl,␣
↪mfxBRCFrameStatus* status) {
      MyBrcContext* ctx = (MyBrcContext*)pthis;
      MyBrcFrame* frame = NULL;
      mfxU32 panic = 0;

      if (!pthis || !par || !ctrl || !status)
         return MFX_ERR_NULL_PTR;

      frame = GetFrame(ctx->frame_queue, ctx->frame_queue_size, par->EncodedOrder);
      if (!frame)
         return MFX_ERR_UNDEFINED_BEHAVIOR;

      // update QP/size stat here

      if (   frame->Status == MFX_BRC_PANIC_BIG_FRAME
        || frame->Status == MFX_BRC_PANIC_SMALL_FRAME)
         panic = 1;

      if (panic || (par->CodedFrameSize >= frame->MinSize && par->CodedFrameSize <=␣
↪frame->MaxSize)) {
         UpdateBRCState(par->CodedFrameSize, ctx);
         RemoveFromQueue(ctx->frame_queue, ctx->frame_queue_size, frame);
         ctx->frame_queue_size--;
         status->BRCStatus = MFX_BRC_OK;
```

(continues on next page)

(continued from previous page)

```
        // Here update Min/MaxSize for all queued frames

        return MFX_ERR_NONE;
    }

    panic = ((GetTime() - frame->StartTime) >= GetMaxFrameEncodingTime(ctx));

    if (par->CodedFrameSize > frame->MaxSize) {
        if (panic || (frame->QP >= ctx->max_qp[0])) {
            frame->Status = MFX_BRC_PANIC_BIG_FRAME;
        } else {
            frame->Status = MFX_BRC_BIG_FRAME;
            frame->QP = DEFAULT_QP_INC;
        }
    }

    if (par->CodedFrameSize < frame->MinSize) {
        if (panic || (frame->QP <= ctx->min_qp[0])) {
            frame->Status = MFX_BRC_PANIC_SMALL_FRAME;
            status->MinFrameSize = frame->MinSize;
        } else {
            frame->Status = MFX_BRC_SMALL_FRAME;
            frame->QP = DEFAULT_QP_DEC;
        }
    }

    status->BRCStatus = frame->Status;

    return MFX_ERR_NONE;
}

void EncoderInit()
{
      //initialize encoder
      MyBrcContext brc_ctx;
      mfxExtBRC ext_brc;
      mfxExtCodingOption2 co2;
      mfxExtBuffer* ext_buf[2] = {&co2.Header, &ext_brc.Header};
      mfxVideoParam vpar;

      memset(&brc_ctx, 0, sizeof(MyBrcContext));
      memset(&ext_brc, 0, sizeof(mfxExtBRC));
      memset(&co2, 0, sizeof(mfxExtCodingOption2));

      vpar.ExtParam = ext_buf;
      vpar.NumExtParam = sizeof(ext_buf) / sizeof(ext_buf[0]);

      co2.Header.BufferId = MFX_EXTBUFF_CODING_OPTION2;
      co2.Header.BufferSz = sizeof(mfxExtCodingOption2);
      co2.ExtBRC = MFX_CODINGOPTION_ON;

      ext_brc.Header.BufferId = MFX_EXTBUFF_BRC;
      ext_brc.Header.BufferSz = sizeof(mfxExtBRC);
      ext_brc.pthis           = &brc_ctx;
      ext_brc.Init            = MyBrcInit;
      ext_brc.Reset           = MyBrcReset;
```

(continues on next page)

(continued from previous page)

```
        ext_brc.Close           = MyBrcClose;
        ext_brc.GetFrameCtrl    = MyBrcGetFrameCtrl;
        ext_brc.Update          = MyBrcUpdate;

        sts = MFXVideoENCODE_Query(session, &vpar, &vpar);
        if (sts == MFX_ERR_UNSUPPORTED || co2.ExtBRC != MFX_CODINGOPTION_ON)
            // unsupported case
            sts = sts;
        else
            sts = MFXVideoENCODE_Init(session, &vpar);
    }
```

JPEG

The application can use the same encoding procedures for JPEG/motion JPEG encoding, as shown in the following pseudo code:

```
// encoder initialization
MFXVideoENCODE_Init (...);
// single frame/picture encoding
MFXVideoENCODE_EncodeFrameAsync (...);
MFXVideoCORE_SyncOperation(...);
// close down
MFXVideoENCODE_Close(...);
```

The application may specify Huffman and quantization tables during encoder initialization by attaching *mfxExtJPEGQuantTables* and *mfxExtJPEGHuffmanTables* buffers to the *mfxVideoParam* structure. If the application does not define tables, then the oneVPL encoder uses tables recommended in ITU-T* Recommendation T.81. If the application does not define a quantization table it must specify the *mfxInfoMFX::Quality* parameter. In this case, the oneVPL encoder scales the default quantization table according to the specified *mfxInfoMFX::Quality* parameter value.

The application should properly configure chroma sampling format and color format using the *mfxFrameInfo::FourCC* and *mfxFrameInfo::ChromaFormat* fields. For example, to encode a 4:2:2 vertically sampled YCbCr picture, the application should set *mfxFrameInfo::FourCC* to *MFX_FOURCC_YUY2* and *mfxFrameInfo::ChromaFormat* to *MFX_CHROMAFORMAT_YUV422V*. To encode a 4:4:4 sampled RGB picture, the application should set *mfxFrameInfo::FourCC* to *MFX_FOURCC_RGB4* and *mfxFrameInfo::ChromaFormat* to *MFX_CHROMAFORMAT_YUV444*.

The oneVPL encoder supports different sets of chroma sampling and color formats on different platforms. The application must call the *MFXVideoENCODE_Query()* function to check if the required color format is supported on a given platform and then initialize the encoder with proper values of *mfxFrameInfo::FourCC* and *mfxFrameInfo::ChromaFormat*.

The application should not define the number of scans and number of components. These numbers are derived by the oneVPL encoder from the `mfxInfoMFx::Interleaved` flag and from chroma type. If interleaved coding is specified, then one scan is encoded that contains all image components. Otherwise, the number of scans is equal to number of components. The encoder uses the following component IDs: “1” for luma (Y), “2” for chroma Cb (U), and “3” for chroma Cr (V).

The application should allocate a buffer that is big enough to hold the encoded picture. A rough upper limit may be calculated using the following equation where **Width** and **Height** are width and height of the picture in pixel and **BytesPerPx** is the number of bytes for one pixel:

```
BufferSizeInKB = 4 + (Width * Height * BytesPerPx + 1023) / 1024;
```

The equation equals 1 for a monochrome picture, 1.5 for NV12 and YV12 color formats, 2 for YUY2 color format, and 3 for RGB32 color format (alpha channel is not encoded).

Multi-view Video Encoding

Similar to the decoding and video processing initialization procedures, the application attaches the *mfxExtMVCSeqDesc* structure to the *mfxVideoParam* structure for encoding initialization. The *mfxExtMVCSeqDesc* structure configures the oneVPL MVC encoder to work in three modes:

- **Default dependency mode:** The application specifies *mfxExtMVCSeqDesc::NumView* and all other fields to zero. The oneVPL encoder creates a single operation point with all views (view identifier 0 : NumView-1) as target views. The first view (view identifier 0) is the base view. Other views depend on the base view.
- **Explicit dependency mode:** The application specifies *mfxExtMVCSeqDesc::NumView* and the view dependency array, and sets all other fields to zero. The oneVPL encoder creates a single operation point with all views (view identifier View[0 : NumView-1].ViewId) as target views. The first view (view identifier View[0].ViewId) is the base view. View dependencies are defined as *mfxMVCViewDependency* structures.
- **Complete mode:** The application fully specifies the views and their dependencies. The oneVPL encoder generates a bitstream with corresponding stream structures.

During encoding, the oneVPL encoding function *MFXVideoENCODE_EncodeFrameAsync()* accumulates input frames until encoding of a picture is possible. The function returns *mfxStatus::MFX_ERR_MORE_DATA* for more data at input or *mfxStatus::MFX_ERR_NONE* if it successfully accumulated enough data for encoding a picture. The generated bitstream contains the complete picture (multiple views). The application can change this behavior and instruct the encoder to output each view in a separate bitstream buffer. To do so, the application must turn on the *mfxExtCodingOption::ViewOutput* flag. In this case, the encoder returns *mfxStatus::MFX_ERR_MORE_BITSTREAM* if it needs more bitstream buffers at output and *mfxStatus::MFX_ERR_NONE* when processing of the picture (multiple views) has been finished. It is recommended that the application provide a new input frame each time the oneVPL encoder requests a new bitstream buffer. The application must submit view data for encoding in the order they are described in the *mfxExtMVCSeqDesc* structure. Particular view data can be submitted for encoding only when all views that it depends upon have already been submitted.

The following pseudo code shows the encoding procedure:

```
mfxExtBuffer *eb;
mfxExtMVCSeqDesc  seq_desc;
mfxVideoParam init_param;

init_param.ExtParam=(mfxExtBuffer **)&eb;
init_param.NumExtParam=1;
eb=(mfxExtBuffer *)&seq_desc;

/* init encoder */
MFXVideoENCODE_Init(session, &init_param);

/* perform encoding */
for (;;) {
    MFXVideoENCODE_EncodeFrameAsync(session, NULL, surface2, bits,
                                    &syncp);
    MFXVideoCORE_SyncOperation(session,syncp,INFINITE);
}

```

(continues on next page)

(continued from previous page)

```
/* close encoder */
MFXVideoENCODE_Close(session);
```

11.2.6 Video Processing Procedures

The following pseudo code shows the video processing procedure:

```
MFXVideoVPP_QueryIOSurf(session, &init_param, response);
allocate_pool_of_surfaces(in_pool, response[0].NumFrameSuggested);
allocate_pool_of_surfaces(out_pool, response[1].NumFrameSuggested);
MFXVideoVPP_Init(session, &init_param);
mfxFrameSurface1 *in=find_unlocked_surface_and_fill_content(in_pool);
mfxFrameSurface1 *out=find_unlocked_surface_from_the_pool(out_pool);
for (;;) {
   sts=MFXVideoVPP_RunFrameVPPAsync(session,in,out,NULL,&syncp);
   if (sts==MFX_ERR_MORE_SURFACE || sts==MFX_ERR_NONE) {
      MFXVideoCORE_SyncOperation(session,syncp,INFINITE);
      process_output_frame(out);
      out=find_unlocked_surface_from_the_pool(out_pool);
   }
   if (sts==MFX_ERR_MORE_DATA && in==NULL)
      break;
   if (sts==MFX_ERR_NONE || sts==MFX_ERR_MORE_DATA) {
      in=find_unlocked_surface_from_the_pool(in_pool);
      fill_content_for_video_processing(in);
      if (end_of_stream())
         in=NULL;
   }
}
MFXVideoVPP_Close(session);
free_pool_of_surfaces(in_pool);
free_pool_of_surfaces(out_pool);
```

Note the following key points about the example:

- The application uses the *MFXVideoVPP_QueryIOSurf()* function to obtain the number of frame surfaces needed for input and output. The application must allocate two frame surface pools: one for the input and one for the output.
- The video processing function *MFXVideoVPP_RunFrameVPPAsync()* is asynchronous. The application must use the *MFXVideoCORE_SyncOperation()* function to synchronize in order to make the output result ready.
- The body of the video processing procedure covers the following three scenarios:
  - If the number of frames consumed at input is equal to the number of frames generated at output, *VPP* returns *mfxStatus::MFX_ERR_NONE* when an output is ready. The application must process the output frame after synchronization, as the *MFXVideoVPP_RunFrameVPPAsync()* function is asynchronous. The application must provide a NULL input at the end of the sequence to drain any remaining frames.
  - If the number of frames consumed at input is more than the number of frames generated at output, VPP returns *mfxStatus::MFX_ERR_MORE_DATA* for additional input until an output is ready. When the output is ready, VPP returns *mfxStatus::MFX_ERR_NONE*. The application must process the output frame after synchronization and provide a NULL input at the end of the sequence to drain any remaining frames.

  - If the number of frames consumed at input is less than the number of frames generated at output, VPP returns either *mfxStatus::MFX_ERR_MORE_SURFACE* (when more than one output is ready), or *mfxStatus::MFX_ERR_NONE* (when one output is ready and VPP expects new input). In both cases, the application must process the output frame after synchronization and provide a NULL input at the end of the sequence to drain any remaining frames.

Configuration

oneVPL configures the video processing pipeline operation based on the difference between the input and output formats, specified in the *mfxVideoParam* structure. The following list shows several examples:

- When the input color format is *YUY2* and the output color format is *NV12*, oneVPL enables color conversion from YUY2 to NV12.
- When the input is interleaved and the output is progressive, oneVPL enables deinterlacing.
- When the input is single field and the output is interlaced or progressive, oneVPL enables field weaving, optionally with deinterlacing.
- When the input is interlaced and the output is single field, oneVPL enables field splitting.

In addition to specifying the input and output formats, the application can provide hints to fine-tune the video processing pipeline operation. The application can disable filters in the pipeline by using the *mfxExtVPPDoNotUse* structure, enable filters by using the *mfxExtVPPDoUse* structure, and configure filters by using dedicated configuration structures. See the *Configurable VPP Filters table* for a complete list of configurable video processing filters, their IDs, and configuration structures. See the *ExtendedBufferID enumerator* for more details.

oneVPL ensures that all filters necessary to convert the input format to the output format are included in the pipeline. oneVPL may skip some optional filters even if they are explicitly requested by the application, for example due to limitations of the underlying hardware. To notify the application about skipped optional filters, oneVPL returns the *mfxStatus::MFX_WRN_FILTER_SKIPPED* warning. The application can retrieve the list of active filters by attaching the *mfxExtVPPDoUse* structure to the *mfxVideoParam* structure and calling the *MFXVideoVPP_GetVideoParam()* function. The application must allocate enough memory for the filter list.

See the *Configurable VPP Filters table* for a full list of configurable filters.

Table 3: Configurable VPP Filters

Filter ID	Configuration Structure
MFX_EXTBUFF_VPP_DENOISE	*mfxExtVPPDenoise*
MFX_EXTBUFF_VPP_MCTF	*mfxExtVppMctf*
MFX_EXTBUFF_VPP_DETAIL	*mfxExtVPPDetail*
MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION	*mfxExtVPPFrameRateConversion*
MFX_EXTBUFF_VPP_IMAGE_STABILIZATION	*mfxExtVPPImageStab*
MFX_EXTBUFF_VPP_PROCAMP	*mfxExtVPPProcAmp*
MFX_EXTBUFF_VPP_FIELD_PROCESSING	*mfxExtVPPFieldProcessing*

The following example shows video processing configuration:

```
/* enable image stabilization filter with default settings */
mfxExtVPPDoUse du;
mfxU32 al=MFX_EXTBUFF_VPP_IMAGE_STABILIZATION;

du.Header.BufferId=MFX_EXTBUFF_VPP_DOUSE;
du.Header.BufferSz=sizeof(mfxExtVPPDoUse);
du.NumAlg=1;
du.AlgList=&al;
```

(continues on next page)

(continued from previous page)

```
9
10 /* configure the mfxVideoParam structure */
11 mfxVideoParam conf;
12 mfxExtBuffer *eb=(mfxExtBuffer *)&du;
13
14 memset(&conf,0,sizeof(conf));
15 conf.IOPattern=MFX_IOPATTERN_IN_SYSTEM_MEMORY | MFX_IOPATTERN_OUT_SYSTEM_MEMORY;
16 conf.NumExtParam=1;
17 conf.ExtParam=&eb;
18
19 conf.vpp.In.FourCC=MFX_FOURCC_YV12;
20 conf.vpp.Out.FourCC=MFX_FOURCC_NV12;
21 conf.vpp.In.Width=conf.vpp.Out.Width=1920;
22 conf.vpp.In.Height=conf.vpp.Out.Height=1088;
23
24 /* video processing initialization */
25 MFXVideoVPP_Init(session, &conf);
```

Region of Interest

During video processing operations, the application can specify a region of interest for each frame as shown in the following figure:

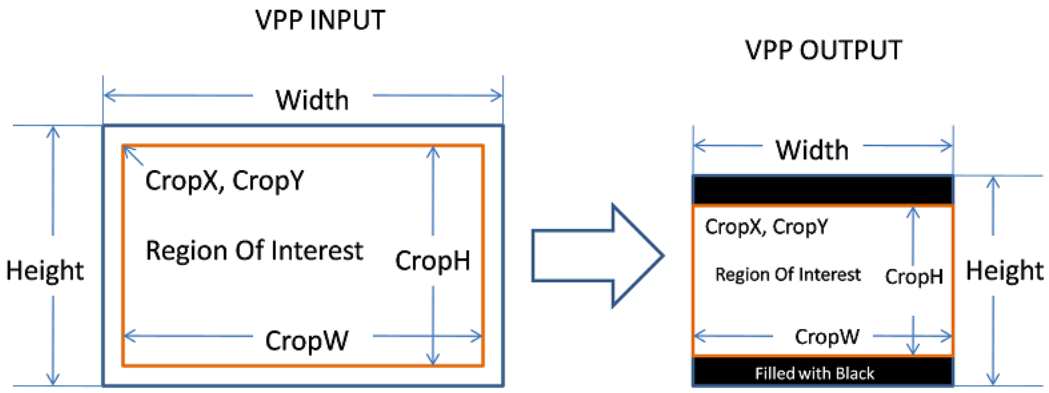


Fig. 5: VPP region of interest operation

Specifying a region of interest guides the resizing function to achieve special effects, such as resizing from 16:9 to 4:3, while keeping the aspect ratio intact. Use the `CropX`, `CropY`, `CropW`, and `CropH` parameters in the *`mfxVideoParam`* structure to specify a region of interest.

The *VPP Region of Interest Operations table* shows examples of VPP operations applied to a region of interest.

Table 4: VPP Region of Interest Operations

Operation	**VPP Input** *Width X Height*	**VPP Input** *CropX, CropY, CropW, CropH*	**VPP Output** *Width X Height*	**VPP Output** *CropX, CropY, CropW, CropH*
Cropping	720 x 480	16, 16, 688, 448	720 x 480	16, 16, 688, 448
Resizing	720 x 480	0, 0, 720, 480	1440 x 960	0, 0, 1440, 960
Horizontal stretching	720 x 480	0, 0, 720, 480	640 x 480	0, 0, 640, 480
16:9 4:3 with letter boxing at the top and bottom	1920 x 1088	0, 0, 1920, 1088	720 x 480	0, 36, 720, 408
4:3 16:9 with pillar boxing at the left and right	720 x 480	0, 0, 720, 480	1920 x 1088	144, 0, 1632, 1088

Multi-view Video Processing

oneVPL video processing supports processing multiple views. For video processing initialization, the application needs to attach the *mfxExtMVCSeqDesc* structure to the *mfxVideoParam* structure and call the *MFXVideoVPP_Init()* function. The function saves the view identifiers. During video processing, oneVPL processes each view individually. oneVPL refers to the `FrameID` field of the *mfxFrameInfo* structure to configure each view according to its processing pipeline. If the video processing source frame is not the output from the oneVPL MVC decoder, then the application needs to fill the the `FrameID` field before calling the *MFXVideoVPP_RunFrameVPPAsync()* function. This is shown in the following pseudo code:

```
mfxExtBuffer *eb;
mfxExtMVCSeqDesc  seq_desc;
mfxVideoParam init_param;

init_param.ExtParam = &eb;
init_param.NumExtParam=1;
eb=(mfxExtBuffer *)&seq_desc;

/* init VPP */
MFXVideoVPP_Init(session, &init_param);

/* perform processing */
for (;;) {
    MFXVideoVPP_RunFrameVPPAsync(session,in,out,NULL,&syncp);
    MFXVideoCORE_SyncOperation(session,syncp,INFINITE);
}

/* close VPP */
MFXVideoVPP_Close(session);
```

11.2.7 Transcoding Procedures

The application can use oneVPL encoding, decoding, and video processing functions together for transcoding operations. This section describes the key aspects of connecting two or more oneVPL functions together.

Asynchronous Pipeline

The application passes the output of an upstream oneVPL function to the input of the downstream oneVPL function to construct an asynchronous pipeline. Pipeline construction is done at runtime and can be dynamically changed, as shown in the following example:

```
mfxSyncPoint sp_d, sp_e;
MFXVideoDECODE_DecodeFrameAsync(session,bs,work,&vin, &sp_d);
if (going_through_vpp) {
   MFXVideoVPP_RunFrameVPPAsync(session,vin,vout, NULL, &sp_d);
   MFXVideoENCODE_EncodeFrameAsync(session,NULL,vout,bits2,&sp_e);
} else {
   MFXVideoENCODE_EncodeFrameAsync(session,NULL,vin,bits2,&sp_e);
}
MFXVideoCORE_SyncOperation(session,sp_e,INFINITE);
```

oneVPL simplifies the requirements for asynchronous pipeline synchronization. The application only needs to synchronize after the last oneVPL function. Explicit synchronization of intermediate results is not required and may slow performance.

oneVPL tracks dynamic pipeline construction and verifies dependency on input and output parameters to ensure the execution order of the pipeline function. In the previous example, oneVPL will ensure *MFXVideoENCODE_EncodeFrameAsync()* does not begin its operation until *MFXVideoDECODE_DecodeFrameAsync()* or *MFXVideoVPP_RunFrameVPPAsync()* has finished.

During the execution of an asynchronous pipeline, the application must consider the input data as "in use" and must not change it until the execution has completed. The application must also consider output data unavailable until the execution has finished. In addition, for encoders, the application must consider extended and payload buffers as "in use" while the input surface is locked.

oneVPL checks dependencies by comparing the input and output parameters of each oneVPL function in the pipeline. Do not modify the contents of input and output parameters before the previous asynchronous operation finishes. Doing so will break the dependency check and can result in undefined behavior. An exception occurs when the input and output parameters are structures, in which case overwriting fields in the structures is allowed.

Note: The dependency check works on the pointers to the structures only.

There are two exceptions with respect to intermediate synchronization:

- If the input is from any asynchronous operation, the application must synchronize any input before calling the oneVPL *MFXVideoDECODE_DecodeFrameAsync()* function.
- When the application calls an asynchronous function to generate an output surface in video memory and passes that surface to a non-oneVPL component, it must explicitly synchronize the operation before passing the surface to the non-oneVPL component.

Surface Pool Allocation

When connecting API function **A** to API function **B**, the application must take into account the requirements of both functions to calculate the number of frame surfaces in the surface pool. Typically, the application can use the formula **Na+Nb**, where **Na** is the frame surface requirements for oneVPL function **A** output, and **Nb** is the frame surface requirements for oneVPL function **B** input.

For performance considerations, the application must submit multiple operations and delay synchronization as much as possible, which gives oneVPL flexibility to organize internal pipelining. For example, compare the following two operation sequences, where the first sequence is the recommended order:

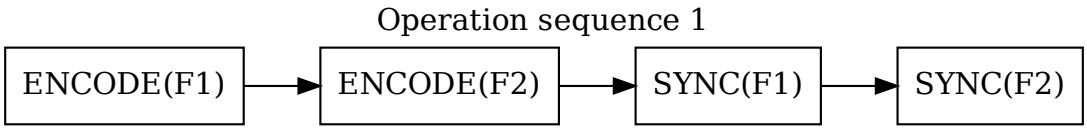


Fig. 6: Recommended operation sequence

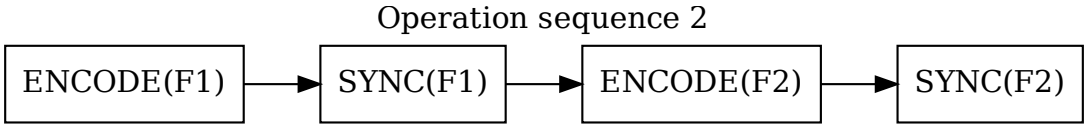


Fig. 7: Operation sequence - not recommended

In this example, the surface pool needs additional surfaces to take into account multiple asynchronous operations before synchronization. The application can use the *mfxVideoParam::AsyncDepth* field to inform a oneVPL function of the number of asynchronous operations the application plans to perform before synchronization. The corresponding oneVPL **QueryIOSurf** function will reflect this number in the *mfxFrameAllocRequest::NumFrameSuggested* value. The following example shows a way of calculating the surface needs based on *mfxFrameAllocRequest::NumFrameSuggested* values:

```
mfxVideoParam init_param_v, init_param_e;
mfxFrameAllocRequest response_v[2], response_e;

// Desired depth
mfxU16 async_depth=4;

init_param_v.AsyncDepth=async_depth;
MFXVideoVPP_QueryIOSurf(session, &init_param_v, response_v);
init_param_e.AsyncDepth=async_depth;
MFXVideoENCODE_QueryIOSurf(session, &init_param_e, &response_e);
mfxU32 num_surfaces=    response_v[1].NumFrameSuggested
```

(continues on next page)

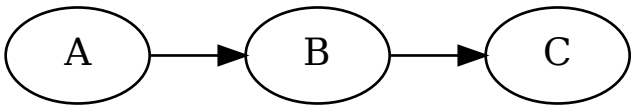
(continued from previous page)

```
        +response_e.NumFrameSuggested
        -async_depth; /* double counted in ENCODE & VPP */
```

Pipeline Error Reporting

During asynchronous pipeline construction, each pipeline stage function will return a synchronization point (sync point). These synchronization points are useful in tracking errors during the asynchronous pipeline operation.

For example, assume the following pipeline:



The application synchronizes on sync point **C**. If the error occurs in function **C**, then the synchronization returns the exact error code. If the error occurs before function **C**, then the synchronization returns *mfxStatus::MFX_ERR_ABORTED*. The application can then try to synchronize on sync point **B**. Similarly, if the error occurs in function **B**, the synchronization returns the exact error code, or else *mfxStatus::MFX_ERR_ABORTED*. The same logic applies if the error occurs in function **A**.

11.2.8 Hardware Acceleration

oneVPL provides a new model for working with hardware acceleration while continuing to support hardware acceleration in legacy mode.

New Model to Work with Hardware Acceleration

oneVPL API version 2.0 introduces a new memory model: internal allocation where oneVPL is responsible for video memory allocation. In this mode, an application is not dependent on a low-level video framework API, such as DirectX* or the VA API, and does not need to create and set corresponding low-level oneVPL primitives such as *ID3D11Device* or *VADisplay*. Instead, oneVPL creates all required objects to work with hardware acceleration and video surfaces internally. An application can get access to these objects using *MFXVideoCORE_GetHandle()* or with help of the *mfxFrameSurfaceInterface* interface.

This approach simplifies the oneVPL initialization, making calls to the *MFXVideoENCODE_QueryIOSurf()*, *MFXVideoDECODE_QueryIOSurf()*, or *MFXVideoVPP_QueryIOSurf()* functions optional. See *Internal Memory Management*.

Work with Hardware Acceleration in Legacy Mode

Work with Multiple Media Devices

If your system has multiple graphics adapters, you may need hints on which adapter is better suited to process a particular workload. The legacy mode of oneVPL provides a helper API to select the most suitable adapter for your workload based on the provided workload description. The following example shows workload initialization on a discrete adapter:

```
mfxU32 num_adapters_available;
mfxIMPL impl;

// Query number of graphics adapters available on system
mfxStatus sts = MFXQueryAdaptersNumber(&num_adapters_available);
MSDK_CHECK_STATUS(sts, "MFXQueryAdaptersNumber failed");

// Allocate memory for response
std::vector<mfxAdapterInfo> displays_data(num_adapters_available);
mfxAdaptersInfo adapters = { displays_data.data(), mfxU32(displays_data.size()), 0u };

// Query information about all adapters (mind that first parameter is NULL)
sts = MFXQueryAdapters(nullptr, &adapters);
MSDK_CHECK_STATUS(sts, "MFXQueryAdapters failed");

// Find dGfx adapter in list of adapters
auto idx_d = std::find_if(adapters.Adapters, adapters.Adapters + adapters.NumActual,
    [](const mfxAdapterInfo info)
{
   return info.Platform.MediaAdapterType == mfxMediaAdapterType::MFX_MEDIA_DISCRETE;
});

// No dGfx in list
if (idx_d == adapters.Adapters + adapters.NumActual)
{
   printf("Warning: No dGfx detected on machine\n");
   return -1;
}

mfxU32 idx = static_cast<mfxU32>(std::distance(adapters.Adapters, idx_d));

// Choose correct implementation for discrete adapter
switch (adapters.Adapters[idx].Number)
{
case 0:
   impl = MFX_IMPL_HARDWARE;
   break;
case 1:
   impl = MFX_IMPL_HARDWARE2;
   break;
case 2:
   impl = MFX_IMPL_HARDWARE3;
   break;
case 3:
   impl = MFX_IMPL_HARDWARE4;
   break;

default:
```

(continues on next page)

(continued from previous page)

```
    // Try searching on all display adapters
    impl = MFX_IMPL_HARDWARE_ANY;
    break;
}

// Initialize mfxSession in regular way with obtained implementation
```

The example shows that after obtaining the adapter list with *MFXQueryAdapters()*, further initialization of *mfxSession* is performed in the regular way. The specific adapter is selected using the *MFX_IMPL_HARDWARE*, *MFX_IMPL_HARDWARE2*, *MFX_IMPL_HARDWARE3*, or *MFX_IMPL_HARDWARE4* values of *mfxIMPL*.

The following example shows the use of *MFXQueryAdapters()* for querying the most suitable adapter for a particular encode workload:

```
mfxU32 num_adapters_available;
mfxIMPL impl;
mfxVideoParam Encode_mfxVideoParam;

// Query number of graphics adapters available on system
mfxStatus sts = MFXQueryAdaptersNumber(&num_adapters_available);
MSDK_CHECK_STATUS(sts, "MFXQueryAdaptersNumber failed");

// Allocate memory for response
std::vector<mfxAdapterInfo> displays_data(num_adapters_available);
mfxAdaptersInfo adapters = { displays_data.data(), mfxU32(displays_data.size()), 0u };

// Fill description of Encode workload
mfxComponentInfo interface_request = { MFX_COMPONENT_ENCODE, Encode_mfxVideoParam };

// Query information about suitable adapters for Encode workload described by Encode_
↪mfxVideoParam
sts = MFXQueryAdapters(&interface_request, &adapters);

if (sts == MFX_ERR_NOT_FOUND)
{
    printf("Error: No adapters on machine capable to process desired workload\n");
    return -1;
}

MSDK_CHECK_STATUS(sts, "MFXQueryAdapters failed");

// Choose correct implementation for discrete adapter. Mind usage of index 0, this is␣
↪best suitable adapter from MSDK perspective
switch (adapters.Adapters[0].Number)
{
case 0:
    impl = MFX_IMPL_HARDWARE;
    break;
case 1:
    impl = MFX_IMPL_HARDWARE2;
    break;
case 2:
    impl = MFX_IMPL_HARDWARE3;
    break;
case 3:
    impl = MFX_IMPL_HARDWARE4;
    break;
```

(continues on next page)

(continued from previous page)

```
42
43 default:
44    // Try searching on all display adapters
45    impl = MFX_IMPL_HARDWARE_ANY;
46    break;
47 }
48
49 // Initialize mfxSession in regular way with obtained implementation
```

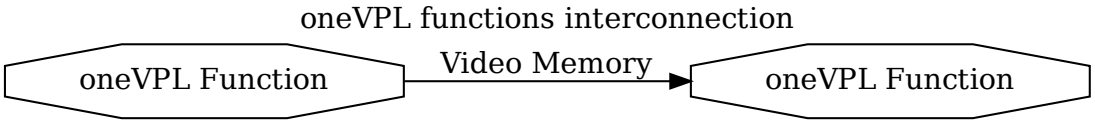
See the *MFXQueryAdapters()* description for adapter priority rules.

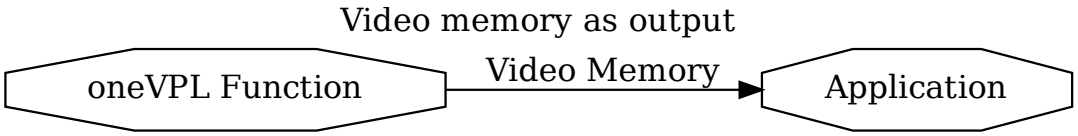
Work with Video Memory

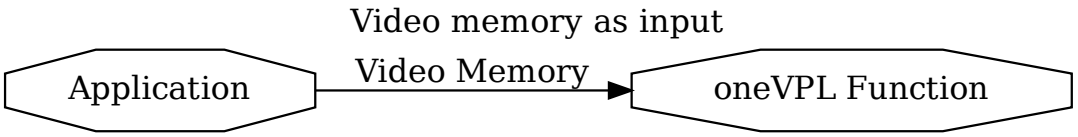
To fully utilize the oneVPL acceleration capability, the application should support OS specific infrastructures. If using Microsoft* Windows*, the application should support Microsoft DirectX*. If using Linux*, the application should support the VA API for Linux.

The hardware acceleration support in an application consists of video memory support and acceleration device support.

Depending on the usage model, the application can use video memory at different stages in the pipeline. Three major scenarios are shown in the following diagrams:







The application must use the *mfxVideoParam::IOPattern* field to indicate the I/O access pattern during initialization. Subsequent function calls must follow this access pattern. For example, if a function operates on video memory surfaces at both input and output, the application must specify the access pattern *IOPattern* at initialization in *MFX_IOPATTERN_IN_VIDEO_MEMORY* for input and *MFX_IOPATTERN_OUT_VIDEO_MEMORY* for output. This particular I/O access pattern must not change inside the **Init - Close** sequence.

Initialization of any hardware accelerated oneVPL component requires the acceleration device handle. This handle is also used by the oneVPL component to query hardware capabilities. The application can share its device with oneVPL by passing the device handle through the *MFXVideoCORE_SetHandle()* function. It is recommended to share the handle before any actual usage of oneVPL.

Work with Microsoft DirectX* Applications

oneVPL supports two different infrastructures for hardware acceleration on the Microsoft Windows OS: the Direct3D* 9 DXVA2 and Direct3D 11 Video API. If Direct3D 9 DXVA2 is used for hardware acceleration, the application should use the *IDirect3DDeviceManager9* interface as the acceleration device handle. If the Direct3D 11 Video API is used for hardware acceleration, the application should use the *ID3D11Device* interface as the acceleration device handle.

The application should share one of these interfaces with oneVPL through the *MFXVideoCORE_SetHandle()* function. If the application does not provide the interface, then oneVPL creates its own internal acceleration device. As a result, oneVPL input and output will be limited to system memory only for the external allocation mode, which will reduce oneVPL performance. If oneVPL fails to create a valid acceleration device, then oneVPL cannot proceed with hardware acceleration and returns an error status to the application.

Note: It is recommended to work in the internal allocation mode if the application does not provide the *IDirect3DDeviceManager9* or *ID3D11Device* interface.

The application must create the Direct3D 9 device with the flag `D3DCREATE_MULTITHREADED`. The flag `D3DCREATE_FPU_PRESERVE` is also recommended. This influences floating-point calculations, including PTS values.

The application must also set multi-threading mode for the Direct3D 11 device. The following example shows how to set multi-threading mode for a Direct3D 11 device:

```
1 ID3D11Device            *pD11Device;
2 ID3D11DeviceContext     *pD11Context;
3 ID3D10Multithread       *pD10Multithread;
4
```

(continues on next page)

(continued from previous page)

```
pD11Device->GetImmediateContext(&pD11Context);
pD11Context->QueryInterface(IID_ID3D10Multithread, &pD10Multithread);
pD10Multithread->SetMultithreadProtected(true);
```

During hardware acceleration, if a Direct3D "device lost" event occurs, the oneVPL operation terminates with the *mfxStatus::MFX_ERR_DEVICE_LOST* return status. If the application provided the Direct3D device handle, the application must reset the Direct3D device.

When the oneVPL decoder creates auxiliary devices for hardware acceleration, it must allocate the list of Direct3D surfaces for I/O access, also known as the surface chain, and pass the surface chain as part of the device creation command. In most cases, the surface chain is the frame surface pool mentioned in the *Frame Surface Locking* section.

The application passes the surface chain to the oneVPL component **Init** function through a oneVPL external allocator callback. See the *Memory Allocation and External Allocators* section for details.

Only the decoder **Init** function requests the external surface chain from the application and uses it for auxiliary device creation. Encoder and VPP **Init** functions may only request internal surfaces. See the *ExtMemFrameType enumerator* for more details about different memory types.

Depending on configuration parameters, oneVPL requires different surface types. It is strongly recommended to call the *MFXVideoENCODE_QueryIOSurf()* function, the *MFXVideoDECODE_QueryIOSurf()* function, or the *MFXVideoVPP_QueryIOSurf()* function to determine the appropriate type in the external allocation mode.

Work with VA API Applications

oneVPL supports the VA API infrastructure for hardware acceleration on Linux. The application should use the *VADisplay* interface as the acceleration device handle for this infrastructure and share it with oneVPL through the *MFXVideoCORE_SetHandle()* function.

The following example shows how to obtain the VA display from the X Window System:

```
Display   *x11_display;
VADisplay va_display;

x11_display = XOpenDisplay(current_display);
va_display  = vaGetDisplay(x11_display);

MFXVideoCORE_SetHandle(session, MFX_HANDLE_VA_DISPLAY, (mfxHDL) va_display);
```

The following example shows how to obtain the VA display from the Direct Rendering Manager:

```
int card;
VADisplay va_display;

card = open("/dev/dri/card0", O_RDWR); /* primary card */
va_display = vaGetDisplayDRM(card);
vaInitialize(va_display, &major_version, &minor_version);

MFXVideoCORE_SetHandle(session, MFX_HANDLE_VA_DISPLAY, (mfxHDL) va_display);
```

When the oneVPL decoder creates a hardware acceleration device, it must allocate the list of video memory surfaces for I/O access, also known as the surface chain, and pass the surface chain as part of the device creation command. The application passes the surface chain to the oneVPL component **Init** function through a oneVPL external allocator callback. See the *Memory Allocation and External Allocators* section for details. Starting from oneVPL API version 2.0, oneVPL creates its own surface chain if an external allocator is not set. See the :ref`New Model to work with Hardware Acceleration <hw-acceleration>` section for details.

Note: The VA API does not define any surface types and the application can use either *MFX_MEMTYPE_VIDEO_MEMORY_DECODER_TARGET* or *MFX_MEMTYPE_VIDEO_MEMORY_PROCESSOR_TARGET* to indicate data in video memory.

11.2.9 Memory Allocation and External Allocators

There are two models of memory management in oneVPL: internal and external.

External Memory Management

In the external memory model, the application must allocate sufficient memory for input and output parameters and buffers and deallocate it when oneVPL functions complete their operations. During execution, the oneVPL functions use callback functions to the application to manage memory for video frames through the external allocator interface *mfxFrameAllocator*.

If an application needs to control the allocation of video frames, it can use callback functions through the *mfxFrameAllocator* interface. If an application does not specify an allocator, an internal allocator is used. However, if an application uses video memory surfaces for input and output, it must specify the hardware acceleration device and an external frame allocator using *mfxFrameAllocator*.

The external frame allocator can allocate different frame types:

- In-system memory.
- In-video memory, as 'Decoder Render Targets' or 'Processor Render Targets.' See *Working with Hardware Acceleration* for additional details.

The external frame allocator responds only to frame allocation requests for the requested memory type and returns *mfxStatus::MFX_ERR_UNSUPPORTED* for all other types. The allocation request uses flags (part of the memory type field) to indicate which oneVPL class initiated the request so that the external frame allocator can respond accordingly.

The following example shows a simple external frame allocator:

```
#define ALIGN32(X) (((mfxU32)((X)+31)) & (~ (mfxU32)31))

typedef struct {
   mfxU16 width, height;
   mfxU8 *base;
} mid_struct;

mfxStatus fa_alloc(mfxHDL pthis, mfxFrameAllocRequest *request, mfxFrameAllocResponse␣
↪*response) {
   if (!(request->Type&MFX_MEMTYPE_SYSTEM_MEMORY))
      return MFX_ERR_UNSUPPORTED;
   if (request->Info.FourCC!=MFX_FOURCC_NV12)
      return MFX_ERR_UNSUPPORTED;
   response->NumFrameActual=request->NumFrameMin;
   for (int i=0;i<request->NumFrameMin;i++) {
      mid_struct *mmid=(mid_struct *)malloc(sizeof(mid_struct));
      mmid->width=ALIGN32(request->Info.Width);
      mmid->height=ALIGN32(request->Info.Height);
      mmid->base=(mfxU8*)malloc(mmid->width*mmid->height*3/2);
      response->mids[i]=mmid;
```

(continues on next page)

(continued from previous page)

```
    }
    return MFX_ERR_NONE;
}

mfxStatus fa_lock(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr) {
    mid_struct *mmid=(mid_struct *)mid;
    ptr->Pitch=mmid->width;
    ptr->Y=mmid->base;
    ptr->U=ptr->Y+mmid->width*mmid->height;
    ptr->V=ptr->U+1;
    return MFX_ERR_NONE;
}

mfxStatus fa_unlock(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr) {
    if (ptr) ptr->Y=ptr->U=ptr->V=ptr->A=0;
    return MFX_ERR_NONE;
}

mfxStatus fa_gethdl(mfxHDL pthis, mfxMemId mid, mfxHDL *handle) {
    return MFX_ERR_UNSUPPORTED;
}

mfxStatus fa_free(mfxHDL pthis, mfxFrameAllocResponse *response) {
    for (int i=0;i<response->NumFrameActual;i++) {
        mid_struct *mmid=(mid_struct *)response->mids[i];
        free(mmid->base); free(mmid);
    }
    return MFX_ERR_NONE;
}
```

For system memory, it is highly recommended to allocate memory for all planes of the same frame as a single buffer (using one single malloc call).

Internal Memory Management

In the internal memory management model, oneVPL provides interface functions for frames allocation:

- *MFXMemory_GetSurfaceForVPP()*
- *MFXMemory_GetSurfaceForVPPOut()*
- *MFXMemory_GetSurfaceForEncode()*
- *MFXMemory_GetSurfaceForDecode()*

These functions are used together with *mfxFrameSurfaceInterface* for surface management. The surface returned by these functions is a reference counted object and the application must call *mfxFrameSurfaceInterface::Release* after finishing all operations with the surface. In this model the application does not need to create and set the external allocator to oneVPL.

Another method to obtain an internally allocated surface is to call *MFXVideoDECODE_DecodeFrameAsync()* with a working surface equal to NULL (see *Simplified decoding procedure*). In this scenario, the decoder will allocate a new refcountable *mfxFrameSurface1* and return it to the user. All assumed contracts with the user are similar to the **MFXMemory_GetSurfaceForXXX** functions.

mfxFrameSurfaceInterface

oneVPL API version 2.0 introduces *mfxFrameSurfaceInterface*. This interface is a set of callback functions to manage the lifetime of allocated surfaces, get access to pixel data, and obtain native handles and device abstractions (if suitable). Instead of directly accessing *mfxFrameSurface1* structure members, it's recommended to use the *mfxFrameSurfaceInterface* if present or call external allocator callback functions if set.

The following pseudo code shows the usage of *mfxFrameSurfaceInterface* for memory sharing:

```
// lets decode frame and try to access output in a an optimal way.
sts = MFXVideoDECODE_DecodeFrameAsync(session, NULL, NULL, &outsurface, &syncp);
if (MFX_ERR_NONE == sts)
{
    outsurface->FrameInterface->GetDeviceHandle(outsurface, &device_handle, &device_type);
    // if application or component is familar with mfxHandleType and it's possible to share memory created by device_handle.
    if (isDeviceTypeCompatible(device_type) && isPossibleForMemorySharing(device_handle)) {
        // get native handle and type
        outsurface->FrameInterface->GetNativeHandle(outsurface, &resource, &resource_type);
        if (isResourceTypeCompatible(resource_type)) {
            //use memory directly
            ProcessNativeMemory(resource);
            outsurface->FrameInterface->Release(outsurface);
        }
    }
    // Application or component is not aware about such DeviceHandle or Resource type need to map to system memory.
    outsurface->FrameInterface->Map(outsurface, MFX_MAP_READ);
    ProcessSystemMemory(outsurface);
    outsurface->FrameInterface->Unmap(outsurface);
    outsurface->FrameInterface->Release(outsurface);
}
}
```

11.2.10 Hardware Device Error Handling

For implementations that accelerate decoding, encoding, and video processing through a hardware device, API functions may return errors or warnings if the hardware device encounters errors. See the *Hardware Device Errors and Warnings table* for detailed information about the errors and warnings.

Table 5: Hardware Device Errors and Warnings

Status	Description
mfxStatus::MFX_ERR_DEVICE_FAILED	Hardware device returned unexpected errors. oneVPL was unable to restore operation.
mfxStatus::MFX_ERR_DEVICE_LOST	Hardware device was lost due to system lock or shutdown.
mfxStatus::MFX_WRN_PARTIAL_ACCELERATION	The hardware does not fully support the specified configuration. The encoding, decoding, or video processing operation may be partially accelerated.
mfxStatus::MFX_WRN_DEVICE_BUSY	Hardware device is currently busy.

oneVPL **Query**, **QueryIOSurf**, and **Init** functions return *mfxStatus::MFX_WRN_PARTIAL_ACCELERATION* to indicate that the encoding, decoding, or video processing operation can be partially hardware accelerated or not hardware accelerated at all. The application can ignore this warning and proceed with the operation. (Note that oneVPL functions may return errors or other warnings overwriting *mfxStatus::MFX_WRN_PARTIAL_ACCELERATION*, as it is a lower priority warning.)

oneVPL functions return *mfxStatus::MFX_WRN_DEVICE_BUSY* to indicate that the hardware device is busy and unable to take commands at this time. Resume the operation by waiting for a few milliseconds and resubmitting the request. The following example shows the decoding pseudo-code:

```
mfxStatus sts=MFX_ERR_NONE;
for (;;) {
   // do something
   sts=MFXVideoDECODE_DecodeFrameAsync(session, bitstream,  surface_work, &surface_
↪disp, &syncp);
   if (sts == MFX_WRN_DEVICE_BUSY) sleep(5);
}
```

The same procedure applies to encoding and video processing.

oneVPL functions return *mfxStatus::MFX_ERR_DEVICE_LOST* or *mfxStatus::MFX_ERR_DEVICE_FAILED* to indicate that there is a complete failure in hardware acceleration. The application must close and reinitialize the oneVPL function class. If the application has provided a hardware acceleration device handle to oneVPL, the application must reset the device.

11.3 Mandatory APIs and Functions

11.3.1 Disclaimer

Developers can implement any subset of the oneVPL API. The specification makes no claim about what encoder, decoder, VPP filter, or any other underlying features are mandatory for the implementation. The oneVPL API is designed such that users have several options to discover capabilities exposed by the implementation:

1. Before session creation: Users can get a list of supported encoders, decoders, VPP filters, correspondent color formats, and memory types with the help of the *MFXEnumImplementations()* function.
2. After session is created: Users can call **Query** functions to obtain low level implementation capabilities.

Attention: The legacy Intel® Media Software Development Kit implementation does not support the first approach to obtain capabilities.

11.3.2 Exported Functions

The *Exported Functions table* lists all functions that must be exposed by any oneAPI Video Processing Library implementation. The realization of all listed functions is mandatory; most functions may return *mfxStatus::MFX_ERR_NOT_IMPLEMENTED*.

Note: Functions *MFXInit()* and *MFXInitEx()* are not required to be exported.

See *Mandatory APIs* for details about which functions, in which conditions, must not return *mfxStatus::MFX_ERR_NOT_IMPLEMENTED*.

Table 6: Exported Functions

Function	API Version
MFXClose()	1.0
MFXQueryIMPL()	1.0
MFXQueryVersion()	1.0
MFXJoinSession()	1.0
MFXDisjoinSession()	1.0
MFXCloneSession()	1.0
MFXSetPriority()	1.0
MFXGetPriority()	1.0
MFXVideoCORE_SetFrameAllocator()	1.0
MFXVideoCORE_SetHandle()	1.0
MFXVideoCORE_GetHandle()	1.0
MFXVideoCORE_SyncOperation()	1.0
MFXVideoENCODE_Query()	1.0
MFXVideoENCODE_QueryIOSurf()	1.0
MFXVideoENCODE_Init()	1.0
MFXVideoENCODE_Reset()	1.0
MFXVideoENCODE_Close()	1.0
MFXVideoENCODE_GetVideoParam()	1.0
MFXVideoENCODE_GetEncodeStat()	1.0
MFXVideoENCODE_EncodeFrameAsync()	1.0
MFXVideoDECODE_Query()	1.0
MFXVideoDECODE_DecodeHeader()	1.0
MFXVideoDECODE_QueryIOSurf()	1.0
MFXVideoDECODE_Init()	1.0
MFXVideoDECODE_Reset()	1.0
MFXVideoDECODE_Close()	1.0
MFXVideoDECODE_GetVideoParam()	1.0
MFXVideoDECODE_GetDecodeStat()	1.0
MFXVideoDECODE_SetSkipMode()	1.0
MFXVideoDECODE_GetPayload()	1.0
MFXVideoDECODE_DecodeFrameAsync()	1.0
MFXVideoVPP_Query()	1.0
MFXVideoVPP_QueryIOSurf()	1.0
MFXVideoVPP_Init()	1.0
MFXVideoVPP_Reset()	1.0
MFXVideoVPP_Close()	1.0
MFXVideoVPP_GetVideoParam()	1.0
MFXVideoVPP_GetVPPStat()	1.0
MFXVideoVPP_RunFrameVPPAsync()	1.0
MFXVideoCORE_QueryPlatform()	1.19
MFXMemory_GetSurfaceForVPP()	2.0
MFXMemory_GetSurfaceForEncode()	2.0
MFXMemory_GetSurfaceForDecode()	2.0
MFXQueryImplsDescription()	2.0
MFXReleaseImplDescription()	2.0
MFXInitialize()	2.0
MFXMemory_GetSurfaceForVPPOut()	2.1
MFXVideoVPP_ProcessFrameAsync()	2.1

11.3.3 Mandatory APIs

All implementations must implement the APIs listed in the *Mandatory APIs table*:

Table 7: Mandatory APIs

Functions	Description
MFXInitialize() *MFXClose()*	Required functions for the dispatcher to create a session.
MFXQueryImplsDescription() *MFXReleaseImplDescription()*	Required functions for the dispatcher to return implementation capabilities.
MFXVideoCORE_SyncOperation()	Required function for synchronization of asynchronous operations.

If the implementation implements any encoder, decoder, or VPP filter, it must implement the corresponding mandatory APIs, as described in the *Mandatory Encode*, *Decode* and *VPP* APIs tables:

Table 8: Mandatory Encode APIs

Functions	Description
MFXVideoENCODE_Init() *MFXVideoENCODE_Close()* *MFXVideoENCODE_Query()* *MFXVideoENCODE_EncodeFrameAsync()*	Required functions if the implementation implements any encoder.

Table 9: Mandatory Decode APIs

Functions	Description
MFXVideoDECODE_Init() *MFXVideoDECODE_Close()* *MFXVideoDECODE_Query()* *MFXVideoDECODE_DecodeFrameAsync()*	Required functions if the implementation implements any decoder.

Table 10: Mandatory VPP APIs

Functions	Description
MFXVideoVPP_Init() *MFXVideoVPP_Close()* *MFXVideoVPP_Query()* *MFXVideoVPP_RunFrameVPPAsync()* or *MFXVideoVPP_ProcessFrameAsync()*	Required functions if the implementation implements any VPP filter.

Note: Mandatory functions must not return the *MFX_ERR_NOT_IMPLEMENTED* status.

If at least one of the encoder, decoder, or VPP filter functions is implemented, the *MFXQueryImplsDescription()* function must return a valid *mfxImplDescription* structure instance with mandatory capabilities of the implementation, including decoder, encoder, or VPP capabilities information.

Any other functions or extension buffers are optional for the implementation.

11.4 oneVPL API Reference

11.4.1 Function Reference

VideoDECODE Functions that implement a complete decoder that decompresses input bitstreams directly to output frame surfaces.

VideoENCODE Functions that perform the entire encoding pipeline from the input video frames to the output bitstream.

VideoVPP Functions that perform video processing before encoding, rendering, or standalone.

VideoCORE Functions to perform external device and memory management and synchronization.

Session Management Functions to manage sessions.

Memory Functions for internal memory allocation and management.

Implementation Capabilities Functions to report capabilities of available implementations and create user-requested library implementations.

Adapters Functions that identify graphics adapters for Microsoft* DirectX* video processing, encoding, and decoding.

VideoDECODE_VPP Functions that implement combined operation of decoding and video processing with multiple output frame surfaces.

VideoDECODE

Functions that implement a complete decoder that decompresses input bitstreams directly to output frame surfaces.

API

- *MFXVideoDECODE_Query*
- *MFXVideoDECODE_DecodeHeader*
- *MFXVideoDECODE_QueryIOSurf*
- *MFXVideoDECODE_Init*
- *MFXVideoDECODE_Reset*
- *MFXVideoDECODE_Close*
- *MFXVideoDECODE_GetVideoParam*
- *MFXVideoDECODE_GetDecodeStat*
- *MFXVideoDECODE_SetSkipMode*
- *MFXVideoDECODE_GetPayload*
- *MFXVideoDECODE_DecodeFrameAsync*

MFXVideoDECODE_Query

mfxStatus **MFXVideoDECODE_Query** (*mfxSession session*, *mfxVideoParam *in*, *mfxVideoParam *out*)

Works in one of two modes:

- If the `in` parameter is zero, the function returns the class configurability in the output structure. A non-zero value in each field of the output structure indicates that the field is configurable by the implementation with the MFXVideoDECODE_Init function.
- If the `in` parameter is non-zero, the function checks the validity of the fields in the input structure. Then the function returns the corrected values to the output structure. If there is insufficient information to determine the validity or correction is impossible, the function zeros the fields. This feature can verify whether the implementation supports certain profiles, levels, or bitrates.

The application can call this function before or after it initializes the decoder. The `CodecId` field of the output structure is a mandated field (to be filled by the application) to identify the coding standard.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_UNSUPPORTED The function failed to identify a specific implementation for the required features.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The decoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

Parameters

- `[in] session`: Session handle.
- `[in] in`: Pointer to the *mfxVideoParam* structure as input.
- `[out] out`: Pointer to the *mfxVideoParam* structure as output.

Important: The *`MFXVideoDECODE_Query()`* is mandatory when implementing a decoder.

MFXVideoDECODE_DecodeHeader

mfxStatus **`MFXVideoDECODE_DecodeHeader`**(*mfxSession session*, *mfxBitstream *bs*, *mfxVideoParam *par*)

Parses the input bitstream and fills the *mfxVideoParam* structure with appropriate values, such as resolution and frame rate, for the Init API function.

The application can then pass the resulting structure to the MFXVideoDECODE_Init function for decoder initialization.

An application can call this API function at any time before or after decoder initialization. If the library finds a sequence header in the bitstream, the function moves the bitstream pointer to the first bit of the sequence header. Otherwise, the function moves the bitstream pointer close to the end of the bitstream buffer but leaves enough data in the buffer to avoid possible loss of start code.

The `CodecId` field of the *mfxVideoParam* structure is a mandated field (to be filled by the application) to identify the coding standard.

The application can retrieve a copy of the bitstream header, by attaching the *mfxExtCodingOptionSPSPPS* structure to the *mfxVideoParam* structure.

Return

- MFX_ERR_NONE The function successfully filled the structure. It does not mean that the stream can be decoded by the library. The application should call MFXVideoDECODE_Query function to check if decoding of the stream is supported.
- MFX_ERR_MORE_DATA The function requires more bitstream data.
- MFX_ERR_UNSUPPORTED `CodecId` field of the *mfxVideoParam* structure indicates some unsupported codec.
- MFX_ERR_INVALID_HANDLE Session is not initialized.
- MFX_ERR_NULL_PTR `bs` or `par` pointer is NULL.

Parameters

- `[in] session`: Session handle.
- `[in] bs`: Pointer to the bitstream.
- `[in] par`: Pointer to the *mfxVideoParam* structure.

MFXVideoDECODE_QueryIOSurf

mfxStatus **MFXVideoDECODE_QueryIOSurf**(*mfxSession session*, *mfxVideoParam *par*, *mfxFrameAllocRequest *request*)

Returns minimum and suggested numbers of the output frame surfaces required for decoding initialization and their type.

Init will call the external allocator for the required frames with the same set of numbers. The use of this function is recommended.

For more information, see the *Working with Hardware Acceleration section*.

The `CodecId` field of the *mfxVideoParam* structure is a mandated field (to be filled by the application) to identify the coding standard. This function does not validate I/O parameters except those used in calculating the number of output surfaces.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoParam* structure as input.
- `[in] request`: Pointer to the *mfxFrameAllocRequest* structure as output.

MFXVideoDECODE_Init

mfxStatus **MFXVideoDECODE_Init**(*mfxSession session*, *mfxVideoParam *par*)

Allocates memory and prepares tables and necessary structures for encoding.

This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX_ERR_UNDEFINED_BEHAVIOR The function is called twice without a close.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoParam* structure.

Important: The *MFXVideoDECODE_Init()* is mandatory when implementing a decoder.

MFXVideoDECODE_Reset

mfxStatus **MFXVideoDECODE_Reset** (*mfxSession session*, *mfxVideoParam* *par)

Stops the current decoding operation and restores internal structures or parameters for a new decoding operation.

Reset serves two purposes:

- It recovers the decoder from errors.
- It restarts decoding from a new position

The function resets the old sequence header (sequence parameter set in H.264, or sequence header in MPEG-2 and VC-1). The decoder will expect a new sequence header before it decodes the next frame and will skip any bitstream before encountering the new sequence header.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected that video parameters are wrong or they conflict with initialization parameters. Reset is impossible.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoParam* structure.

MFXVideoDECODE_Close

mfxStatus **MFXVideoDECODE_Close** (*mfxSession session*)

Terminates the current decoding operation and de-allocates any internal tables or structures.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.

Important: The *MFXVideoDECODE_Close()* is mandatory when implementing a decoder.

MFXVideoDECODE_GetVideoParam

mfxStatus **MFXVideoDECODE_GetVideoParam** (*mfxSession session*, *mfxVideoParam *par*)

Retrieves current working parameters to the specified output structure.

If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure. The application can retrieve a copy of the bitstream header, by attaching the *mfxExtCodingOptionSPSPPS* structure to the *mfxVideoParam* structure.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the corresponding parameter structure.

MFXVideoDECODE_GetDecodeStat

mfxStatus **MFXVideoDECODE_GetDecodeStat** (*mfxSession session*, *mfxDecodeStat *stat*)

Obtains statistics collected during decoding.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.
- `[in] stat`: Pointer to the *mfxDecodeStat* structure.

MFXVideoDECODE_SetSkipMode

mfxStatus **MFXVideoDECODE_SetSkipMode** (*mfxSession session*, *mfxSkipMode mode*)

Sets the decoder skip mode.

The application may use this API function to increase decoding performance by sacrificing output quality. Increasing the skip level first results in skipping of some decoding operations like deblocking and then leads to frame skipping; first B, then P. Particular details are platform dependent.

Return MFX_ERR_NONE The function completed successfully and the output surface is ready for decoding MFX_WRN_VALUE_NOT_CHANGED The skip mode is not affected as the maximum or minimum skip range is reached.

Parameters

- `[in] session`: Session handle.
- `[in] mode`: Decoder skip mode. See the mfxSkipMode enumerator for details.

MFXVideoDECODE_GetPayload

mfxStatus **MFXVideoDECODE_GetPayload**(*mfxSession session*, *mfxU64 *ts*, *mfxPayload *payload*)

Extracts user data (MPEG-2) or SEI (H.264) messages from the bitstream.

Internally, the decoder implementation stores encountered user data or SEI messages. The application may call this API function multiple times to retrieve the user data or SEI messages, one at a time.

If there is no payload available, the function returns with payload->NumBit=0.

Return MFX_ERR_NONE The function completed successfully and the output buffer is ready for decoding. MFX_ERR_NOT_ENOUGH_BUFFER The payload buffer size is insufficient.

Parameters

- `[in] session`: Session handle.
- `[in] ts`: Pointer to the user data time stamp in units of 90 KHz; divide ts by 90,000 (90 KHz) to obtain the time in seconds; the time stamp matches the payload with a specific decoded frame.
- `[in] payload`: Pointer to the *mfxPayload* structure; the payload contains user data in MPEG-2 or SEI messages in H.264.

MFXVideoDECODE_DecodeFrameAsync

mfxStatus **MFXVideoDECODE_DecodeFrameAsync**(*mfxSession session*, *mfxBitstream *bs*, *mfxFrameSurface1 *surface_work*, *mfxFrameSurface1 **surface_out*, *mfxSyncPoint *syncp*)

Decodes the input bitstream to a single output frame.

The `surface_work` parameter provides a working frame buffer for the decoder. The application should allocate the working frame buffer, which stores decoded frames. If the function requires caching frames after decoding, it locks the frames and the application must provide a new frame buffer in the next call.

If, and only if, the function returns MFX_ERR_NONE, the pointer `surface_out` points to the output frame in the display order. If there are no further frames, the function will reset the pointer to zero and return the appropriate status code.

Before decoding the first frame, a sequence header (sequence parameter set in H.264 or sequence header in MPEG-2 and VC-1) must be present. The function skips any bitstreams before it encounters the new sequence header.

The input bitstream `bs` can be of any size. If there are not enough bits to decode a frame, the function returns MFX_ERR_MORE_DATA, and consumes all input bits except if a partial start code or sequence header is at the end of the buffer. In this case, the function leaves the last few bytes in the bitstream buffer. If there is more incoming bitstream, the application should append the incoming bitstream to the bitstream buffer. Otherwise, the application should ignore the remaining bytes in the bitstream buffer and apply the end of stream procedure described below.

The application must set `bs` to NULL to signal end of stream. The application may need to call this API function several times to drain any internally cached frames until the function returns MFX_ERR_MORE_DATA.

If more than one frame is in the bitstream buffer, the function decodes until the buffer is consumed. The decoding process can be interrupted for events such as if the decoder needs additional working buffers, is readying a frame for retrieval, or encountering a new header. In these cases, the function returns appropriate status code and moves the bitstream pointer to the remaining data.

The decoder may return MFX_ERR_NONE without taking any data from the input bitstream buffer. If the application appends additional data to the bitstream buffer, it is possible that the bitstream buffer may contain

more than one frame. It is recommended that the application invoke the function repeatedly until the function returns MFX_ERR_MORE_DATA, before appending any more data to the bitstream buffer.

This function is asynchronous.

Return MFX_ERR_NONE The function completed successfully and the output surface is ready for decoding.
MFX_ERR_MORE_DATA The function requires more bitstream at input before decoding can proceed.

MFX_ERR_MORE_SURFACE The function requires more frame surface at output before decoding can proceed.

MFX_ERR_DEVICE_LOST Hardware device was lost.

See the *Working with Microsoft* DirectX* Applications section* for further information.

MFX_WRN_DEVICE_BUSY Hardware device is currently busy. Call this function again in a few milliseconds.

MFX_WRN_VIDEO_PARAM_CHANGED The decoder detected a new sequence header in the bitstream. Video parameters may have changed.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The decoder detected incompatible video parameters in the bitstream and failed to follow them.

MFX_ERR_REALLOC_SURFACE Bigger surface_work required. May be returned only if *mfxInfoMFX::EnableReallocRequest* was set to ON during initialization.

Parameters

- `[in] session`: Session handle.
- `[in] bs`: Pointer to the input bitstream.
- `[in] surface_work`: Pointer to the working frame buffer for the decoder.
- `[out] surface_out`: Pointer to the output frame in the display order.
- `[out] syncp`: Pointer to the sync point associated with this operation.

Important: The *`MFXVideoDECODE_DecodeFrameAsync()`* is mandatory when implementing a decoder.

VideoENCODE

Functions that perform the entire encoding pipeline from the input video frames to the output bitstream.

API

- *MFXVideoENCODE_Query*
- *MFXVideoENCODE_QueryIOSurf*
- *MFXVideoENCODE_Init*
- *MFXVideoENCODE_Reset*
- *MFXVideoENCODE_Close*

- *MFXVideoENCODE_GetVideoParam*
- *MFXVideoENCODE_GetEncodeStat*
- *MFXVideoENCODE_EncodeFrameAsync*

MFXVideoENCODE_Query

mfxStatus **MFXVideoENCODE_Query** (*mfxSession session*, *mfxVideoParam *in*, *mfxVideoParam *out*)

Works in either of four modes:

- If the `in` parameter is zero, the function returns the class configurability in the output structure. The output structure has a non-zero value in each field that the implementation can configure using Init.
- If the `in` parameter is non-zero, the function checks the validity of the fields in the input structure. Then the function returns the corrected values in the output structure. If there is insufficient information to determine the validity or correction is impossible, the function zeroes the fields. This feature can verify whether the implementation supports certain profiles, levels or bitrates.
- If the `in` parameter is non-zero and *mfxExtEncoderResetOption* structure is attached to it, the function queries for the outcome of the MFXVideoENCODE_Reset function and returns it in the *mfxExtEncoderResetOption* structure attached to out. The query function succeeds if a reset is possible and returns an error otherwise. Unlike other modes that are independent of the encoder state, this one checks if reset is possible in the present encoder state. This mode also requires a completely defined *mfxVideoParam* structure, unlike other modes that support partially defined configurations. See *mfxExtEncoderResetOption* description for more details.
- If the `in` parameter is non-zero and *mfxExtEncoderCapability* structure is attached to it, the function returns encoder capability in the *mfxExtEncoderCapability* structure attached to out. It is recommended to fill in the *mfxVideoParam* structure and set the hardware acceleration device handle before calling the function in this mode.

The application can call this function before or after it initializes the encoder. The `CodecId` field of the output structure is a mandated field (to be filled by the application) to identify the coding standard.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_UNSUPPORTED The function failed to identify a specific implementation for the required features.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

Parameters

- `[in] session`: Session handle.
- `[in] in`: Pointer to the *mfxVideoParam* structure as input.
- `[out] out`: Pointer to the *mfxVideoParam* structure as output.

Important: The *`MFXVideoENCODE_Query()`* function is mandatory when implementing an encoder.

MFXVideoENCODE_QueryIOSurf

mfxStatus **MFXVideoENCODE_QueryIOSurf**(*mfxSession session*, *mfxVideoParam *par*, *mfxFrameAllocRequest *request*)

Returns minimum and suggested numbers of the input frame surfaces required for encoding initialization and their type.

Init will call the external allocator for the required frames with the same set of numbers. This function does not validate I/O parameters except those used in calculating the number of input surfaces.

The use of this function is recommended.

For more information, see the *Working with Hardware Acceleration section*.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoParam* structure as input.
- `[in] request`: Pointer to the *mfxFrameAllocRequest* structure as output.

MFXVideoENCODE_Init

mfxStatus **MFXVideoENCODE_Init**(*mfxSession session*, *mfxVideoParam *par*)

Allocates memory and prepares tables and necessary structures for encoding.

This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX_ERR_UNDEFINED_BEHAVIOR The function is called twice without a close;

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoParam* structure.

Important: The *MFXVideoENCODE_Init()* function is mandatory when implementing an encoder.

MFXVideoENCODE_Reset

mfxStatus **MFXVideoENCODE_Reset** (*mfxSession session*, *mfxVideoParam* *par)

Stops the current encoding operation and restores internal structures or parameters for a new encoding operation, possibly with new parameters.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoParam* structure.

MFXVideoENCODE_Close

mfxStatus **MFXVideoENCODE_Close** (*mfxSession session*)

Terminates the current encoding operation and de-allocates any internal tables or structures.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.

Important: The *MFXVideoENCODE_Close()* function is mandatory when implementing an encoder.

MFXVideoENCODE_GetVideoParam

mfxStatus **MFXVideoENCODE_GetVideoParam** (*mfxSession session*, *mfxVideoParam* *par)

Retrieves current working parameters to the specified output structure.

If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure. The application can retrieve a copy of the bitstream header by attaching the *mfxExtCodingOptionSPSPPS* structure to the *mfxVideoParam* structure.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.

- `[in] par`: Pointer to the corresponding parameter structure.

MFXVideoENCODE_GetEncodeStat

mfxStatus **MFXVideoENCODE_GetEncodeStat** (*mfxSession session*, *mfxEncodeStat *stat*)

Obtains statistics collected during encoding.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.
- `[in] stat`: Pointer to the *mfxEncodeStat* structure.

MFXVideoENCODE_EncodeFrameAsync

mfxStatus **MFXVideoENCODE_EncodeFrameAsync** (*mfxSession session*, *mfxEncodeCtrl *ctrl*, *mfxFrameSurface1 *surface*, *mfxBitstream *bs*, *mfxSyncPoint *syncp*)

Takes a single input frame in either encoded or display order and generates its output bitstream.

In the case of encoded ordering, the *mfxEncodeCtrl* structure must specify the explicit frame type. In the case of display ordering, this function handles frame order shuffling according to the GOP structure parameters specified during initialization.

Since encoding may process frames differently from the input order, not every call of the function generates output and the function returns MFX_ERR_MORE_DATA. If the encoder needs to cache the frame, the function locks the frame. The application should not alter the frame until the encoder unlocks the frame. If there is output (with return status MFX_ERR_NONE), the return is a frame's worth of bitstream.

It is the calling application's responsibility to ensure that there is sufficient space in the output buffer. The value `BufferSizeInKB` in the *mfxVideoParam* structure at encoding initialization specifies the maximum possible size for any compressed frames. This value can also be obtained from the MFXVideoENCODE_GetVideoParam function after encoding initialization.

To mark the end of the encoding sequence, call this function with a NULL surface pointer. Repeat the call to drain any remaining internally cached bitstreams (one frame at a time) until MFX_ERR_MORE_DATA is returned.

This function is asynchronous.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NOT_ENOUGH_BUFFER The bitstream buffer size is insufficient.

MFX_ERR_MORE_DATA The function requires more data to generate any output.

MFX_ERR_DEVICE_LOST Hardware device was lost.

See the *Working with Microsoft* DirectX* Applications section* for further information.

MFX_WRN_DEVICE_BUSY Hardware device is currently busy. Call this function again in a few milliseconds.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM Inconsistent parameters detected not conforming to Configuration Parameter Constraints.

Parameters

- `[in] session`: Session handle.

- `[in] ctrl`: Pointer to the *mfxEncodeCtrl* structure for per-frame encoding control; this parameter is optional (it can be NULL) if the encoder works in the display order mode.
- `[in] surface`: Pointer to the frame surface structure.
- `[out] bs`: Pointer to the output bitstream.
- `[out] syncp`: Pointer to the returned sync point associated with this operation.

Important: The *MFXVideoENCODE_EncodeFrameAsync()* function is mandatory when implementing an encoder.

VideoVPP

Functions that perform video processing before encoding, rendering, or standalone.

API

- *MFXVideoVPP_Query*
- *MFXVideoVPP_QueryIOSurf*
- *MFXVideoVPP_Init*
- *MFXVideoVPP_Reset*
- *MFXVideoVPP_Close*
- *MFXVideoVPP_GetVideoParam*
- *MFXVideoVPP_GetVPPStat*
- *MFXVideoVPP_RunFrameVPPAsync*
- *MFXVideoVPP_ProcessFrameAsync*

MFXVideoVPP_Query

mfxStatus **MFXVideoVPP_Query** (*mfxSession session*, *mfxVideoParam *in*, *mfxVideoParam *out*)

Works in one of two modes:

- If the `in` pointer is zero, the function returns the class configurability in the output structure. A non-zero value in a field indicates that the implementation can configure it with Init.
- If the `in` parameter is non-zero, the function checks the validity of the fields in the input structure. Then the function returns the corrected values to the output structure. If there is insufficient information to determine the validity or correction is impossible, the function zeroes the fields.

The application can call this function before or after it initializes the preprocessor.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_UNSUPPORTED The implementation does not support the specified configuration.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The video processing may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

Parameters

- `[in] session`: Session handle.
- `[in] in`: Pointer to the *mfxVideoParam* structure as input.
- `[out] out`: Pointer to the *mfxVideoParam* structure as output.

Important: The *`MFXVideoVPP_Query()`* function is mandatory when implementing a VPP filter.

MFXVideoVPP_QueryIOSurf

mfxStatus **`MFXVideoVPP_QueryIOSurf`**(*mfxSession session*, *mfxVideoParam *par*, *mfxFrameAllocRequest request*[2])

Returns minimum and suggested numbers of the input frame surfaces required for video processing initialization and their type.

The parameter `request[0]` refers to the input requirements; `request[1]` refers to output requirements. Init will call the external allocator for the required frames with the same set of numbers. This function does not validate I/O parameters except those used in calculating the number of input surfaces.

The use of this function is recommended.

For more information, see the *Working with Hardware Acceleration section*.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The video processing may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoParam* structure as input.
- `[in] request`: Pointer to the *mfxFrameAllocRequest* structure; use `request[0]` for input requirements and `request[1]` for output requirements for video processing.

MFXVideoVPP_Init

mfxStatus **MFXVideoVPP_Init** (*mfxSession session*, *mfxVideoParam *par*)

Allocates memory and prepares tables and necessary structures for video processing.

This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_PARTIAL_ACCELERATION The underlying hardware does not fully support the specified video parameters. The video processing may be partially accelerated. Only hardware implementations may return this status code.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX_ERR_UNDEFINED_BEHAVIOR The function is called twice without a close.

MFX_WRN_FILTER_SKIPPED The VPP skipped one or more filters requested by the application.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoParam* structure.

Important: The *`MFXVideoVPP_Init()`* function is mandatory when implementing a VPP filter.

MFXVideoVPP_Reset

mfxStatus **MFXVideoVPP_Reset** (*mfxSession session*, *mfxVideoParam *par*)

Stops the current video processing operation and restores internal structures or parameters for a new operation.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected that video parameters are wrong or they conflict with initialization parameters. Reset is impossible.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoParam* structure.

MFXVideoVPP_Close

mfxStatus **MFXVideoVPP_Close**(*mfxSession session*)

Terminates the current video processing operation and de-allocates any internal tables or structures.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.

Important: The *MFXVideoVPP_Close()* function is mandatory when implementing a VPP filter.

MFXVideoVPP_GetVideoParam

mfxStatus **MFXVideoVPP_GetVideoParam**(*mfxSession session*, *mfxVideoParam *par*)

Retrieves current working parameters to the specified output structure.

If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the corresponding parameter structure.

MFXVideoVPP_GetVPPStat

mfxStatus **MFXVideoVPP_GetVPPStat**(*mfxSession session*, *mfxVPPStat *stat*)

Obtains statistics collected during video processing.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.
- `[in] stat`: Pointer to the *mfxVPPStat* structure.

MFXVideoVPP_RunFrameVPPAsync

mfxStatus **MFXVideoVPP_RunFrameVPPAsync**(*mfxSession session*, *mfxFrameSurface1 *in*, *mfxFrameSurface1 *out*, *mfxExtVppAuxData *aux*, *mfxSyncPoint *syncp*)

Processes a single input frame to a single output frame.

Retrieval of the auxiliary data is optional; the encoding process may use it. The video processing process may not generate an instant output given an input.

See the *Video Processing Procedures section* for details on how to correctly send input and retrieve output.

At the end of the stream, call this function with the input argument `in=NULL` to retrieve any remaining frames, until the function returns MFX_ERR_MORE_DATA. This function is asynchronous.

Return MFX_ERR_NONE The output frame is ready after synchronization. MFX_ERR_MORE_DATA Need more input frames before VPP can produce an output.

MFX_ERR_MORE_SURFACE The output frame is ready after synchronization. Need more surfaces at output for additional output frames available.

MFX_ERR_DEVICE_LOST Hardware device was lost.

See the *Working with Microsoft* DirectX* Applications section* for further information.

MFX_WRN_DEVICE_BUSY Hardware device is currently busy. Call this function again in a few milliseconds.

Parameters

- `[in] session`: Session handle.
- `[in] in`: Pointer to the input video surface structure.
- `[out] out`: Pointer to the output video surface structure.
- `[in] aux`: Optional pointer to the auxiliary data structure.
- `[out] syncp`: Pointer to the output sync point.

MFXVideoVPP_ProcessFrameAsync

mfxStatus **MFXVideoVPP_ProcessFrameAsync**(*mfxSession session*, *mfxFrameSurface1 *in*, *mfxFrameSurface1 **out*)

The function processes a single input frame to a single output frame with internal allocation of output frame.

At the end of the stream, call this function with the input argument `in=NULL` to retrieve any remaining frames, until the function returns MFX_ERR_MORE_DATA. This function is asynchronous.

Return MFX_ERR_NONE The output frame is ready after synchronization. MFX_ERR_MORE_DATA Need more input frames before VPP can produce an output.

MFX_ERR_MEMORY_ALLOC The function failed to allocate output videoframe.

MFX_ERR_DEVICE_LOST Hardware device was lost.

See the *Working with Microsoft* DirectX* Applications section* for further information.

MFX_WRN_DEVICE_BUSY Hardware device is currently busy. Call this function again in a few milliseconds.

Parameters

- `[in] session`: Session handle.
- `[in] in`: Pointer to the input video surface structure.
- `[out] out`: Pointer to the output video surface structure which is reference counted object allocated by the library.

Important: Either *`MFXVideoVPP_RunFrameVPPAsync()`* or *`MFXVideoVPP_ProcessFrameAsync()`* function is mandatory when implementing a VPP filter.

VideoCORE

Functions to perform external device and memory management and synchronization.

API

MFXVideoCORE_SetFrameAllocator

mfxStatus **MFXVideoCORE_SetFrameAllocator** (*mfxSession session*, *mfxFrameAllocator *allocator*)

Sets the external allocator callback structure for frame allocation.

If the allocator argument is NULL, the library uses the default allocator, which allocates frames from system memory or hardware devices. The behavior of the API is undefined if it uses this function while the previous allocator is in use. A general guideline is to set the allocator immediately after initializing the session.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.
- `[in] allocator`: Pointer to the *mfxFrameAllocator* structure

MFXVideoCORE_SetHandle

mfxStatus **MFXVideoCORE_SetHandle** (*mfxSession session*, *mfxHandleType type*, *mfxHDL hdl*)

Sets any essential system handle that the library might use.

If the specified system handle is a COM interface, the reference counter of the COM interface will increase. The counter will decrease when the session closes.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_UNDEFINED_BEHAVIOR The same handle is redefined. For example, the function has been called twice with the same handle type or an internal handle has been created before this function call.

Parameters

- `[in] session`: Session handle.
- `[in] type`: Handle type
- `[in] hdl`: Handle to be set

MFXVideoCORE_GetHandle

mfxStatus **MFXVideoCORE_GetHandle** (*mfxSession session*, *mfxHandleType type*, *mfxHDL *hdl*)

Obtains system handles previously set by the MFXVideoCORE_SetHandle function.

If the handler is a COM interface, the reference counter of the interface increases. The calling application must release the COM interface.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_UNDEFINED_BEHAVIOR Specified handle type not found.

Parameters

- `[in] session`: Session handle.
- `[in] type`: Handle type
- `[in] hdl`: Pointer to the handle to be set

MFXVideoCORE_QueryPlatform

mfxStatus **MFXVideoCORE_QueryPlatform** (*mfxSession session*, *mfxPlatform *platform*)

Returns information about current hardware platform in the Legacy mode.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.
- `[out] platform`: Pointer to the *mfxPlatform* structure

MFXVideoCORE_SyncOperation

mfxStatus **MFXVideoCORE_SyncOperation** (*mfxSession session*, *mfxSyncPoint syncp*, *mfxU32 wait*)

Initiates execution of an asynchronous function not already started and returns the status code after the specified asynchronous operation completes. If wait is zero, the function returns immediately.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NONE_PARTIAL_OUTPUT The function completed successfully, bitstream contains a portion of the encoded frame according to required granularity.

MFX_WRN_IN_EXECUTION The specified asynchronous function is in execution.

MFX_ERR_ABORTED The specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

Parameters

- `[in] session`: Session handle.
- `[in] syncp`: Sync point
- `[in] wait`: wait time in milliseconds

Important: The *`MFXVideoCORE_SyncOperation()`* function is mandatory for any implementation.

Session Management

Functions to manage sessions.

API

MFXInit

mfxStatus **MFXInit** (*mfxIMPL impl*, *mfxVersion *ver*, *mfxSession *session*)

Creates and initializes a session in legacy mode.

Call this function before calling any other API function. If the desired implementation specified by `impl` is MFX_IMPL_AUTO, the function will search for the platform-specific implementation. If the function cannot find the platform-specific implementation, it will use the software implementation instead.

The `ver` argument indicates the desired version of the library implementation. The loaded implementation will have an API version compatible to the specified version (equal in the major version number, and no less in the minor version number.) If the desired version is not specified, the default is to use the API version from the library release with which an application is built.

Production applications should always specify the minimum API version that meets the functional requirements. For example, if an application uses only H.264 decoding as described in API v1.0, the application should initialize the library with API v1.0. This ensures backward compatibility.

Return MFX_ERR_NONE The function completed successfully. The output parameter contains the handle of the session. MFX_ERR_UNSUPPORTED The function cannot find the desired legacy Intel(r) Media SDK implementation or version.

Parameters

- `[in] impl`: mfxIMPL enumerator that indicates the desired legacy Intel(r) Media SDK implementation.
- `[in] ver`: Pointer to the minimum library version or zero, if not specified.
- `[out] session`: Pointer to the legacy Intel(r) Media SDK session handle.

MFXInitEx

mfxStatus **MFXInitEx** (*mfxInitParam par*, *mfxSession *session*)

Creates and initializes a session in legacy mode.

Call this function before calling any other API functions. If the desired implementation specified by `par` is MFX_IMPL_AUTO, the function will search for the platform-specific implementation. If the function cannot find the platform-specific implementation, it will use the software implementation instead.

The argument `par.Version` indicates the desired version of the implementation. The loaded implementation will have an API version compatible to the specified version (equal in the major version number, and no less in the minor version number.) If the desired version is not specified, the default is to use the API version from the library release with which an application is built.

Production applications should always specify the minimum API version that meets the functional requirements. For example, if an application uses only H.264 decoding as described in API v1.0, the application should initialize the library with API v1.0. This ensures backward compatibility.

The argument `par.ExternalThreads` specifies threading mode. Value 0 means that the implementation should create and handle work threads internally (this is essentially the equivalent of the regular MFXInit). I

Return MFX_ERR_NONE The function completed successfully. The output parameter contains the handle of the session. MFX_ERR_UNSUPPORTED The function cannot find the desired implementation or version.

Parameters

- `[in] par`: *mfxInitParam* structure that indicates the desired implementation, minimum library version and desired threading mode.
- `[out] session`: Pointer to the session handle.

MFXInitialize

mfxStatus **MFXInitialize** (*mfxInitializationParam par*, *mfxSession *session*)

Creates and initializes a session starting from API version 2.0. This function is used by the dispatcher. The dispatcher creates and fills the *mfxInitializationParam* structure according to mfxConfig values set by an application. Calling this function directly is not recommended. Instead, applications must call the mfxCreateSession function.

Return MFX_ERR_NONE The function completed successfully. The output parameter contains the handle of the session. MFX_ERR_UNSUPPORTED The function cannot find the desired implementation or version.

Parameters

- `[in] par`: *mfxInitializationParam* structure that indicates the minimum library version and acceleration type.
- `[out] session`: Pointer to the session handle.

Important: The *`MFXInitialize()`* function is mandatory for any implementation.

MFXClose

mfxStatus **MFXClose** (*mfxSession session*)

Completes and deinitializes a session. Any active tasks in execution or in queue are aborted. The application cannot call any API function after calling this function.

All child sessions must be disjoined before closing a parent session.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: session handle.

Important: The *`MFXClose()`* function is mandatory for any implementation.

MFXQueryIMPL

mfxStatus **MFXQueryIMPL** (*mfxSession session*, *mfxIMPL *impl*)

Returns the implementation type of a given session.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.
- `[out] impl`: Pointer to the implementation type

MFXQueryVersion

mfxStatus **MFXQueryVersion** (*mfxSession session*, *mfxVersion *version*)

Returns the implementation version.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: Session handle.
- `[out] version`: Pointer to the returned implementation version.

MFXJoinSession

mfxStatus **MFXJoinSession** (*mfxSession session*, *mfxSession child*)

Joins the child session to the current session.

After joining, the two sessions share thread and resource scheduling for asynchronous operations. However, each session still maintains its own device manager and buffer/frame allocator. Therefore, the application must use a compatible device manager and buffer/frame allocator to share data between two joined sessions.

The application can join multiple sessions by calling this function multiple times. When joining the first two sessions, the current session becomes the parent responsible for thread and resource scheduling of any later joined sessions.

Joining of two parent sessions is not supported.

Return MFX_ERR_NONE The function completed successfully. MFX_WRN_IN_EXECUTION Active tasks are executing or in queue in one of the sessions. Call this function again after all tasks are completed.

MFX_ERR_UNSUPPORTED The child session cannot be joined with the current session.

Parameters

- `[inout] session`: The current session handle.
- `[in] child`: The child session handle to be joined

MFXDisjoinSession

mfxStatus **MFXDisjoinSession** (*mfxSession session*)

Removes the joined state of the current session.

```
   After disjoining, the current session becomes independent. The application␣
↪must ensure there is no active task running in the session before calling this␣
↪API function.
```

Return MFX_ERR_NONE The function completed successfully. MFX_WRN_IN_EXECUTION Active tasks are executing or in queue in one of the sessions. Call this function again after all tasks are completed.

MFX_ERR_UNDEFINED_BEHAVIOR The session is independent, or this session is the parent of all joined sessions.

Parameters

- `[inout] session`: The current session handle.

MFXCloneSession

mfxStatus **MFXCloneSession** (*mfxSession session*, *mfxSession *clone*)

Creates a clean copy of the current session.

```
   The cloned session is an independent session and does not inherit any user-
↪defined buffer, frame allocator, or device manager handles from the current␣
↪session.
   This function is a light-weight equivalent of MFXJoinSession after MFXInit.
```

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: The current session handle.
- `[out] clone`: Pointer to the cloned session handle.

MFXSetPriority

mfxStatus **MFXSetPriority** (*mfxSession session*, *mfxPriority priority*)
Sets the current session priority.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: The current session handle.
- `[in] priority`: Priority value.

MFXGetPriority

mfxStatus **MFXGetPriority** (*mfxSession session*, *mfxPriority *priority*)
Returns the current session priority.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] session`: The current session handle.
- `[out] priority`: Pointer to the priority value.

Memory

Functions for internal memory allocation and management.

API

- *MFXMemory_GetSurfaceForVPP*
- *MFXMemory_GetSurfaceForVPPOut*
- *MFXMemory_GetSurfaceForEncode*
- *MFXMemory_GetSurfaceForDecode*

MFXMemory_GetSurfaceForVPP

mfxStatus **MFXMemory_GetSurfaceForVPP** (*mfxSession session*, *mfxFrameSurface1 **surface*)
Returns surface which can be used as input for VPP.

VPP should be initialized before this call. Surface should be released with mfxFrameSurface1::FrameInterface.Release(...) after usage. The value of mfxFrameSurface1::Data.Locked for the returned surface is 0.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR If double-pointer to the `surface` is NULL. MFX_ERR_INVALID_HANDLE If `session` was not initialized.

MFX_ERR_NOT_INITIALIZED If VPP was not initialized (allocator needs to know surface size from somewhere).

MFX_ERR_MEMORY_ALLOC In case of any other internal allocation error.

Parameters

- `[in] session`: Session handle.
- `[out] surface`: Pointer is set to valid *mfxFrameSurface1* object.

Alias below, can be used as well:

MFXMemory_GetSurfaceForVPPIn
Alias for MFXMemory_GetSurfaceForVPP function.

MFXMemory_GetSurfaceForVPPOut

mfxStatus **MFXMemory_GetSurfaceForVPPOut** (*mfxSession session*, *mfxFrameSurface1 **surface*)
Returns surface which can be used as output of VPP.

VPP should be initialized before this call. Surface should be released with mfxFrameSurface1::FrameInterface.Release(...) after usage. The value of mfxFrameSurface1::Data.Locked for the returned surface is 0.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR If double-pointer to the `surface` is NULL. MFX_ERR_INVALID_HANDLE If `session` was not initialized. MFX_ERR_NOT_INITIALIZED If VPP was not initialized (allocator needs to know surface size from somewhere).

MFX_ERR_MEMORY_ALLOC In case of any other internal allocation error.

Parameters

- `[in] session`: Session handle.
- `[out] surface`: Pointer is set to valid *mfxFrameSurface1* object.

MFXMemory_GetSurfaceForEncode

mfxStatus **MFXMemory_GetSurfaceForEncode** (*mfxSession session*, *mfxFrameSurface1 **surface*)
Returns a surface which can be used as input for the encoder.

Encoder should be initialized before this call. Surface should be released with mfxFrameSurface1::FrameInterface.Release(...) after usage. The value of mfxFrameSurface1::Data.Locked for the returned surface is 0.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If session was not initialized.

MFX_ERR_NOT_INITIALIZED If the encoder was not initialized (allocator needs to know surface size from somewhere).

MFX_ERR_MEMORY_ALLOC In case of any other internal allocation error.

Parameters

- `[in] session`: Session handle.

- `[out] surface`: Pointer is set to valid *mfxFrameSurface1* object.

MFXMemory_GetSurfaceForDecode

mfxStatus **MFXMemory_GetSurfaceForDecode** (*mfxSession session*, *mfxFrameSurface1 **surface*)

Returns a surface which can be used as output of the decoder.

Decoder should be initialized before this call. Surface should be released with mfxFrameSurface1::FrameInterface.Release(...) after usage. The value of mfxFrameSurface1::Data.Locked for the returned surface is 0.'

Note This function was added to simplify transition from legacy surface management to the proposed internal allocation approach. Previously, the user allocated surfaces for the working pool and fed them to the decoder using DecodeFrameAsync calls. With MFXMemory_GetSurfaceForDecode it is possible to change the existing pipeline by just changing the source of work surfaces. Newly developed applications should prefer direct usage of DecodeFrameAsync with internal allocation.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If session was not initialized.

MFX_ERR_NOT_INITIALIZED If the decoder was not initialized (allocator needs to know surface size from somewhere).

MFX_ERR_MEMORY_ALLOC Other internal allocation error.

Parameters

- `[in] session`: Session handle.
- `[out] surface`: Pointer is set to valid *mfxFrameSurface1* object.

Implementation Capabilities

Functions to report capabilities of available implementations and create user-requested library implementations.

API

- *MFXQueryImplsDescription*
- *MFXReleaseImplDescription*

MFXQueryImplsDescription

mfxHDL ***MFXQueryImplsDescription*** (*mfxImplCapsDeliveryFormat format*, *mfxU32 *num_impls*)

Delivers implementation capabilities in the requested format according to the format value.

Return Array of handles to the capability report or NULL in case of unsupported format or NULL num_impls pointer. Length of array is equal to num_impls.

Parameters

- `[in] format`: Format in which capabilities must be delivered. See mfxImplCapsDeliveryFormat for more details.
- `[out] num_impls`: Number of the implementations.

Important: The *`MFXQueryImplsDescription()`* function is mandatory for any implementation.

MFXReleaseImplDescription

mfxStatus ***MFXReleaseImplDescription*** (*mfxHDL hdl*)

Destroys the handle allocated by the MFXQueryImplsCapabilities function. Implementation must remember which handles are released. Once the last handle is released, this function must release memory allocated for the array of handles.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] hdl`: Handle to destroy. Can be equal to NULL.

Important: The *`MFXReleaseImplDescription()`* function is mandatory for any implementation.

Adapters

Functions that identify graphics adapters for Microsoft* DirectX* video processing, encoding, and decoding.

API

- *MFXQueryAdapters*
- *MFXQueryAdaptersDecode*
- *MFXQueryAdaptersNumber*

MFXQueryAdapters

mfxStatus **MFXQueryAdapters**(*mfxComponentInfo* *input_info, *mfxAdaptersInfo* *adapters)

Returns a list of adapters that are suitable to handle workload `input_info`. The list is sorted in priority order, with iGPU given the highest precedence. This rule may change in the future. If the `input_info` pointer is NULL, the list of all available adapters will be returned.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR `input_info` or adapters pointer is NULL. MFX_ERR_NOT_FOUND No suitable adapters found.

MFX_WRN_OUT_OF_RANGE Not enough memory to report back entire list of adapters. In this case as many adapters as possible will be returned.

Parameters

- `[in] input_info`: Pointer to workload description. See *mfxComponentInfo* description for details.
- `[out] adapters`: Pointer to output description of all suitable adapters for input workload. See *mfxAdaptersInfo* description for details.

MFXQueryAdaptersDecode

mfxStatus **MFXQueryAdaptersDecode**(*mfxBitstream* *bitstream, *mfxU32* codec_id, *mfxAdaptersInfo* *adapters)

Returns list of adapters that are suitable to decode the input bitstream. The list is sorted in priority order, with iGPU given the highest precedence. This rule may change in the future. This function is a simplification of MFXQueryAdapters, because bitstream is a description of the workload itself.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR bitstream or `adapters` pointer is NULL. MFX_ERR_NOT_FOUND No suitable adapters found.

MFX_WRN_OUT_OF_RANGE Not enough memory to report back entire list of adapters. In this case as many adapters as possible will be returned.

Parameters

- `[in] bitstream`: Pointer to bitstream with input data.
- `[in] codec_id`: Codec ID to determine the type of codec for the input bitstream.
- `[out] adapters`: Pointer to the output list of adapters. Memory should be allocated by user. See *mfxAdaptersInfo* description for details.

MFXQueryAdaptersNumber

mfxStatus **MFXQueryAdaptersNumber**(*mfxU32* *num_adapters)

Returns the number of detected graphics adapters. It can be used before calling MFXQueryAdapters to determine the size of input data that the user will need to allocate.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR num_adapters pointer is NULL.

Parameters

- `[out] num_adapters`: Pointer for the output number of detected graphics adapters.

VideoDECODE_VPP

Functions that implement combined operation of decoding and video processing with multiple output frame surfaces.

API

- *MFXVideoDECODE_VPP_Init*
- *MFXVideoDECODE_VPP_Reset*
- *MFXVideoDECODE_VPP_GetChannelParam*

MFXVideoDECODE_VPP_Init

mfxStatus **MFXVideoDECODE_VPP_Init**(*mfxSession session*, *mfxVideoParam *decode_par*, *mfxVideoChannelParam **vpp_par_array*, *mfxU32 num_vpp_par*)

Initialize the SDK in (decode + vpp) mode. The logic of this function is similar to MFXVideoDECODE_Init, but application has to provide array of pointers to *mfxVideoChannelParam* and num_channel_param - number of channels. Application is responsible for memory allocation for *mfxVideoChannelParam* parameters and for each channel it should specify channel IDs: mfxVideoChannelParam::mfxFrameInfo::ChannelId. ChannelId should be unique value within one session. The application can attach *mfxExtInCrops* to *mfxVideoChannelParam::ExtParam* to annotate input video frame if it wants to enable letterboxing operation.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX_ERR_UNDEFINED_BEHAVIOR The component is already initialized.

MFX_WRN_FILTER_SKIPPED The VPP skipped one or more filters requested by the application.

Parameters

- `[in] session`: SDK session handle.
- `[in] decode_par`: Pointer to the *mfxVideoParam* structure which contains initialization parameters for decoder.
- `[in] vpp_par_array`: Array of pointers to *mfxVideoChannelParam*structures. Each *mfxVideoChannelParam* contains initialization parameters for each VPP channel.
- `[in] num_vpp_par`: Size of array of pointers to *mfxVideoChannelParam* structures.

Important: The *MFXVideoDECODE_VPP_Init()* is mandatory when implementing a combined decode plus vpp.

MFXVideoDECODE_VPP_Reset

mfxStatus **MFXVideoDECODE_VPP_Reset**(*mfxSession session*, *mfxVideoParam *decode_par*, *mfxVideoChannelParam **vpp_par_array*, *mfxU32 num_vpp_par*)

This function is similar to MFXVideoDECODE_Reset and stops the current decoding and vpp operation, and restores internal structures or parameters for a new decoding plus vpp operation. It resets the state of the decoder and/or all initialized vpp channels. Applications have to care about draining of buffered frames for decode and all vpp channels before call this function. The application can attach *mfxExtInCrops* to *mfxVideoChannelParam::ExtParam* to annotate input video frame if it wants to enable letterboxing operation.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_INVALID_VIDEO_PARAM The function detected that video parameters are wrong or they conflict with initialization parameters. Reset is impossible.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the component and then reinitialize it.

MFX_WRN_INCOMPATIBLE_VIDEO_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved. MFX_ERR_NULL_PTR Both pointers decode_par and vpp_par_array` equal to zero.

Parameters

- `[in] session`: Session handle.
- `[in] decode_par`: Pointer to the *mfxVideoParam* structure which contains new initialization parameters for decoder. Might be NULL if application wants to Reset only VPP channels.
- `[in] vpp_par_array`: Array of pointers to *mfxVideoChannelParam* structures. Each *mfxVideoChannelParam* contains new initialization parameters for each VPP channel.
- `[in] num_vpp_par`: Size of array of pointers to *mfxVideoChannelParam* structures.

MFXVideoDECODE_VPP_GetChannelParam

mfxStatus **MFXVideoDECODE_VPP_GetChannelParam**(*mfxSession session*, *mfxVideoChannelParam *par*, *mfxU32 channel_id*)

Returns actual VPP parameters for selected channel which should be specified by application through mfxVideoChannelParam::mfxFrameInfo::ChannelId.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR par pointer is NULL.

MFX_ERR_NOT_FOUND the library is not able to find VPP channel with such channel_id.

Parameters

- `[in] session`: Session handle.
- `[in] par`: Pointer to the *mfxVideoChannelParam* structure which allocated by application
- `[in] channel_id`: specifies the requested channel's info

11.4.2 Structure Reference

Type Definitions Structures used for type definitions.

Memory Structures Structures used for memory.

Implementation Management Structures used for implementation management.

Cross-component Structures Structures used across library components.

Decode Structures Structures used by Decode only.

Encode Structures Structures used by Encode only.

VPP Structures Structures used by VPP only.

Protected Structures Protected structures.

DECODDE_VPP Structures Structures used by *DECODE_VPP* only.

Type Definitions

Structures used for type definitions.

API

- *mfxExtBuffer*
- *mfxHDLPair*
- *mfxI16Pair*
- *mfxRange32U*
- *mfxStructVersion*

mfxExtBuffer

struct mfxExtBuffer
The common header definition for external buffers and video processing hints.

Public Members

mfxU32 **BufferId**
Identifier of the buffer content. See the ExtendedBufferID enumerator for a complete list of extended buffers.

mfxU32 **BufferSz**
Size of the buffer.

mfxHDLPair

struct mfxHDLPair
Represents pair of handles of type mfxHDL.

Public Members

mfxHDL **first**
First handle.

mfxHDL **second**
Second handle.

mfxI16Pair

struct mfxI16Pair
Represents a pair of numbers of type mfxI16.

Public Members

mfxI16 **x**
First number.

mfxI16 **y**
Second number.

mfxRange32U

struct mfxRange32U
Represents a range of unsigned values.

Public Members

mfxU32 **Min**
Minimal value of the range.

mfxU32 **Max**
Maximal value of the range.

mfxU32 **Step**
Value increment.

mfxStructVersion

union mfxStructVersion

#include <mfxdefs.h> Introduce the field Version for any structure. Assumed that any structure changes are backward binary compatible. *mfxStructVersion* starts from {1,0} for any new API structures. If *mfxStructVersion* is added to the existent legacy structure (replacing reserved fields) it starts from {1, 1}.

Major and Minor fields

Anonymous structure with Major and Minor fields. Minor number is incremented when reserved fields are used. Major number is incremented when the size of structure is increased.

mfxU8 **Minor**
Minor number of the correspondent structure.

mfxU8 **Major**
Major number of the correspondent structure.

Public Members

struct *mfxStructVersion*::**[anonymous] [anonymous]**

mfxU16 **Version**
Structure version number.

Memory Structures

Structures used for memory.

API

- *mfxBitstream*
- *mfxFrameAllocator*
- *mfxFrameAllocRequest*
- *mfxFrameAllocResponse*
- *mfxFrameData*
- *mfxFrameInfo*
- *mfxFrameSurface1*
- *mfxFrameSurfaceInterface*

mfxBitstream

struct mfxBitstream

Defines the buffer that holds compressed video data.

Public Members

mfxEncryptedData ***EncryptedData**

Reserved and must be zero.

mfxExtBuffer ****ExtParam**

Array of extended buffers for additional bitstream configuration. See the ExtendedBufferID enumerator for a complete list of extended buffers.

mfxU16 **NumExtParam**

The number of extended buffers attached to this structure.

mfxU32 **CodecId**

Specifies the codec format identifier in the FourCC code. See the CodecFormatFourCC enumerator for details. This optional parameter is required for the simplified decode initialization.

mfxI64 **DecodeTimeStamp**

Decode time stamp of the compressed bitstream in units of 90KHz. A value of MFX_TIMESTAMP_UNKNOWN indicates that there is no time stamp.

This value is calculated by the encoder from the presentation time stamp provided by the application in the *mfxFrameSurface1* structure and from the frame rate provided by the application during the encoder initialization.

mfxU64 **TimeStamp**

Time stamp of the compressed bitstream in units of 90KHz. A value of MFX_TIMESTAMP_UNKNOWN indicates that there is no time stamp.

mfxU8 ***Data**

Bitstream buffer pointer, 32-bytes aligned.

mfxU32 **DataOffset**

Next reading or writing position in the bitstream buffer.

mfxU32 **DataLength**

Size of the actual bitstream data in bytes.

mfxU32 **MaxLength**

Allocated bitstream buffer size in bytes.

mfxU16 **PicStruct**

Type of the picture in the bitstream. Output parameter.

mfxU16 **FrameType**

Frame type of the picture in the bitstream. Output parameter.

mfxU16 **DataFlag**

Indicates additional bitstream properties. See the BitstreamDataFlag enumerator for details.

mfxU16 **reserved2**

Reserved for future use.

mfxFrameAllocator

struct mfxFrameAllocator

Describes the API callback functions Alloc, Lock, Unlock, GetHDL, and Free that the implementation might use for allocating internal frames. Applications that operate on OS-specific video surfaces must implement these API callback functions.

Using the default allocator implies that frame data passes in or out of functions through pointers, as opposed to using memory IDs.

Behavior is undefined when using an incompletely defined external allocator.

See the *Memory Allocation and External Allocators section* for additional information.

Public Members

mfxHDL **pthis**

Pointer to the allocator object.

mfxStatus (***Alloc**) (*mfxHDL* pthis, *mfxFrameAllocRequest* *request, *mfxFrameAllocResponse* *response)

Allocates surface frames. For decoders, MFXVideoDECODE_Init calls Alloc only once. That call includes all frame allocation requests. For encoders, MFXVideoENCODE_Init calls Alloc twice: once for the input surfaces and again for the internal reconstructed surfaces.

If two library components must share DirectX* surfaces, this function should pass the pre-allocated surface chain to the library instead of allocating new DirectX surfaces.

See the *Surface Pool Allocation section* for additional information.

Return MFX_ERR_NONE The function successfully allocated the memory block. MFX_ERR_MEMORY_ALLOC The function failed to allocate the video frames.

MFX_ERR_UNSUPPORTED The function does not support allocating the specified type of memory.

Parameters

- `[in] pthis`: Pointer to the allocator object.
- `[in] request`: Pointer to the *mfxFrameAllocRequest* structure that specifies the type and number of required frames.
- `[out] response`: Pointer to the *mfxFrameAllocResponse* structure that retrieves frames actually allocated.

mfxStatus (***Lock**) (*mfxHDL* pthis, *mfxMemId* mid, *mfxFrameData* *ptr)

Locks a frame and returns its pointer.

Return MFX_ERR_NONE The function successfully locked the memory block. MFX_ERR_LOCK_MEMORY This function failed to lock the frame.

Parameters

- `[in] pthis`: Pointer to the allocator object.
- `[in] mid`: Memory block ID.
- `[out] ptr`: Pointer to the returned frame structure.

mfxStatus (***Unlock**) (*mfxHDL* pthis, *mfxMemId* mid, *mfxFrameData* *ptr)

Unlocks a frame and invalidates the specified frame structure.

Return MFX_ERR_NONE The function successfully locked the memory block.

Parameters

- `[in] pthis`: Pointer to the allocator object.
- `[in] mid`: Memory block ID.
- `[out] ptr`: Pointer to the frame structure. This pointer can be NULL.

mfxStatus (***GetHDL**) (*mfxHDL* pthis, *mfxMemId* mid, *mfxHDL* *handle)
Returns the OS-specific handle associated with a video frame. If the handle is a COM interface, the reference counter must increase. The library will release the interface afterward.

Return MFX_ERR_NONE The function successfully returned the OS-specific handle. MFX_ERR_UNSUPPORTED The function does not support obtaining OS-specific handle..

Parameters

- `[in] pthis`: Pointer to the allocator object.
- `[in] mid`: Memory block ID.
- `[out] handle`: Pointer to the returned OS-specific handle.

mfxStatus (***Free**) (*mfxHDL* pthis, *mfxFrameAllocResponse* *response)
De-allocates all allocated frames.

Return MFX_ERR_NONE The function successfully de-allocated the memory block.

Parameters

- `[in] pthis`: Pointer to the allocator object.
- `[in] response`: Pointer to the *mfxFrameAllocResponse* structure returned by the Alloc function.

mfxFrameAllocRequest

struct mfxFrameAllocRequest
Describes multiple frame allocations when initializing encoders, decoders, and video preprocessors. A range specifies the number of video frames. Applications are free to allocate additional frames. In all cases, the minimum number of frames must be at least NumFrameMin or the called API function will return an error.

Public Members

mfxU32 **AllocId**
Unique (within the session) ID of component requested the allocation.

mfxFrameInfo **Info**
Describes the properties of allocated frames.

mfxU16 **Type**
Allocated memory type. See the ExtMemFrameType enumerator for details.

mfxU16 **NumFrameMin**
Minimum number of allocated frames.

mfxU16 **NumFrameSuggested**
Suggested number of allocated frames.

mfxFrameAllocResponse

struct mfxFrameAllocResponse
Describes the response to multiple frame allocations. The calling API function returns the number of video frames actually allocated and pointers to their memory IDs.

Public Members

mfxU32 **AllocId**
Unique (within the session) ID of component requested the allocation.

mfxMemId ***mids**
Pointer to the array of the returned memory IDs. The application allocates or frees this array.

mfxU16 **NumFrameActual**
Number of frames actually allocated.

mfxFrameData

struct mfxY410
Specifies "pixel" in Y410 color format

Public Members

mfxU32 **U**
U component.

mfxU32 **Y**
Y component.

mfxU32 **V**
V component.

mfxU32 **A**
A component.

struct mfxA2RGB10
Specifies "pixel" in A2RGB10 color format

Public Members

mfxU32 **B**
B component.

mfxU32 **G**
G component.

mfxU32 **R**
R component.

mfxU32 **A**
A component.

struct mfxFrameData
Describes frame buffer pointers.

Extension Buffers

mfxU16 **NumExtParam**
The number of extra configuration structures attached to this structure.

General members

mfxU16 **reserved**[9]
Reserved for future use.

mfxU16 **MemType**
Allocated memory type. See the ExtMemFrameType enumerator for details. Used for better integration of 3rd party plugins into the pipeline.

mfxU16 **PitchHigh**
Distance in bytes between the start of two consecutive rows in a frame.

mfxU64 **TimeStamp**
Time stamp of the video frame in units of 90KHz. Divide TimeStamp by 90,000 (90 KHz) to obtain the time in seconds. A value of MFX_TIMESTAMP_UNKNOWN indicates that there is no time stamp.

mfxU32 **FrameOrder**
Current frame counter for the top field of the current frame. An invalid value of MFX_FRAMEORDER_UNKNOWN indicates that API functions that generate the frame output do not use this frame.

mfxU16 **Locked**
Counter flag for the application. If Locked is greater than zero then the application locks the frame or field pair. Do not move, alter or delete the frame.

Color Planes

Data pointers to corresponding color channels (planes). The frame buffer pointers must be 16-byte aligned. The application has to specify pointers to all color channels even for packed formats. For example, for YUY2 format the application must specify Y, U, and V pointers. For RGB32 format, the application must specify R, G, B, and A pointers.

mfxU8 ***A**
A channel.

mfxMemId **MemId**
Memory ID of the data buffers. Ignored if any of the preceding data pointers is non-zero.

Additional Flags

mfxU16 **Corrupted**
Some part of the frame or field pair is corrupted. See the Corruption enumerator for details.

mfxU16 **DataFlag**
Additional flags to indicate frame data properties. See the FrameDataFlag enumerator for details.

Public Members

mfxExtBuffer ****ExtParam**
Points to an array of pointers to the extra configuration structures. See the ExtendedBufferID enumerator for a list of extended configurations.

mfxU16 **PitchLow**
Distance in bytes between the start of two consecutive rows in a frame.

mfxU8 ***Y**
Y channel.

mfxU16 ***Y16**
Y16 channel.

mfxU8 ***R**
R channel.

mfxU8 ***UV**
UV channel for UV merged formats.

mfxU8 ***VU**
YU channel for VU merged formats.

mfxU8 ***CbCr**
CbCr channel for CbCr merged formats.

mfxU8 ***CrCb**
CrCb channel for CrCb merged formats.

mfxU8 ***Cb**
Cb channel.

mfxU8 ***U**
U channel.

mfxU16 ***U16**
U16 channel.

mfxU8 ***G**
G channel.

mfxY410 ***Y410**
T410 channel for Y410 format (merged AVYU).

mfxU8 ***Cr**
Cr channel.

mfxU8 ***V**
V channel.

mfxU16 ***V16**
V16 channel.

mfxU8 ***B**
B channel.

mfxA2RGB10 ***A2RGB10**
A2RGB10 channel for A2RGB10 format (merged ARGB).

mfxFrameInfo

struct mfxFrameInfo
Specifies properties of video frames. See also "Configuration Parameter Constraints" chapter.

FrameRate

Specify the frame rate with the following formula: FrameRateExtN / FrameRateExtD.

For encoding, frame rate must be specified. For decoding, frame rate may be unspecified (FrameRateExtN and FrameRateExtD are all zeros.) In this case, the frame rate is defaulted to 30 frames per second.

mfxU32 **FrameRateExtN**
Frame rate numerator.

mfxU32 **FrameRateExtD**
Frame rate denominator.

AspectRatio

AspectRatioW and AspectRatioH are used to specify the sample aspect ratio. If sample aspect ratio is explicitly defined by the standards (see Table 6-3 in the MPEG-2 specification or Table E-1 in the H.264 specification), AspectRatioW and AspectRatioH should be the defined values. Otherwise, the sample aspect ratio can be derived as follows:

- `AspectRatioW=display_aspect_ratio_width*display_height`
- `AspectRatioH=display_aspect_ratio_height*display_width`

For MPEG-2, the above display aspect ratio must be one of the defined values in Table 6-3 in the MPEG-2 specification. For H.264, there is no restriction on display aspect ratio values.

If both parameters are zero, the encoder uses the default value of sample aspect ratio.

mfxU16 **AspectRatioW**
Aspect Ratio for width.

mfxU16 **AspectRatioH**
Aspect Ratio for height.

ROI

The region of interest of the frame. Specify the display width and height in *mfxVideoParam*.

mfxU16 **CropX**
X coordinate.

mfxU16 **CropY**
Y coordinate.

mfxU16 **CropW**
Width in pixels.

mfxU16 **CropH**
Height in pixels.

Public Members

mfxU32 **reserved**[4]
Reserved for future use.

mfxU16 **ChannelId**
The unique ID of each VPP channel set by application. It's required that during Init/Reset application fills ChannelId for each *mfxVideoChannelParam* provided by the application and the SDK sets it back to the correspondent mfxSurfaceArray::mfxFrameSurface1 to distinguish different channels. It's expected that surfaces for some channels might be returned with some delay so application has to use *mfxFrameInfo::ChannelId* to distinguish what returned surface belongs to what VPP channel. Decoder's initialization parameters are always sent through channel with *mfxFrameInfo::ChannelId* equals to zero. It's allowed to skip setting of decoder's parameters for simplified decoding procedure

mfxU16 **BitDepthLuma**
Number of bits used to represent luma samples.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU16 **BitDepthChroma**
Number of bits used to represent chroma samples.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU16 **Shift**
When the value is not zero, indicates that values of luma and chroma samples are shifted. Use BitDepthLuma and BitDepthChroma to calculate shift size. Use zero value to indicate absence of shift. See example data alignment below.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxFrameId **FrameId**
Describes the view and layer of a frame picture.

mfxU32 **FourCC**
FourCC code of the color format. See the ColorFourCC enumerator for details.

mfxU16 **Width**
Width of the video frame in pixels. Must be a multiple of 16.

mfxU16 **Height**
Height of the video frame in pixels. Must be a multiple of 16 for progressive frame sequence and a multiple of 32 otherwise.

mfxU64 **BufferSize**
Size of frame buffer in bytes. Valid only for plain formats (when FourCC is P8). In this case, Width, Height, and crop values are invalid.

mfxU16 **PicStruct**
Picture type as specified in the PicStruct enumerator.

mfxU16 **ChromaFormat**
Color sampling method. Value is the same as that of ChromaFormatIdc. ChromaFormat is not defined if FourCC is zero.

Note: Example data alignment for Shift = 0:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	Valid data									

Example data alignment for Shift != 0:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Valid data										0	0	0	0	0	0

---

mfxFrameSurface1

struct mfxFrameSurface1
Defines the uncompressed frames surface information and data buffers. The frame surface is in the frame or complementary field pairs of pixels up to four color-channels, in two parts: *mfxFrameInfo* and *mfxFrameData*.

Public Members

struct *mfxFrameSurfaceInterface* ***FrameInterface**
Specifies interface to work with surface.

mfxFrameInfo **Info**
Specifies surface properties.

mfxFrameData **Data**
Describes the actual frame buffer.

mfxFrameSurfaceInterface

struct mfxFrameSurfaceInterface

Public Members

mfxHDL **Context**
The context of the memory interface. User should not touch (change, set, null) this pointer.

mfxStructVersion **Version**
The version of the structure.

mfxStatus (***AddRef**) (*mfxFrameSurface1* *surface)
Increments the internal reference counter of the surface. The surface is not destroyed until the surface is released using the (*Release) function. (*AddRef) should be used each time a new link to the surface is created (for example, copy structure) for proper surface management.

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid surface.

mfxStatus (***Release**) (*mfxFrameSurface1* *surface)
Decrements the internal reference counter of the surface. (*Release) should be called after using the (*AddRef) function to add a surface or when allocation logic requires it. For example, call (*Release) to release a surface obtained with the GetSurfaceForXXX function.

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNDEFINED_BEHAVIOR If Reference Counter of surface is zero before call.

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid surface.

mfxStatus (***GetRefCounter**) (*mfxFrameSurface1* *surface, *mfxU32* *counter)
Returns current reference counter of *mfxFrameSurface1* structure.

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If surface or counter is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid surface.
- `[out] counter`: Sets counter to the current reference counter value.

mfxStatus (***Map**) (*mfxFrameSurface1* *surface, *mfxU32* flags)

Sets pointers of surface->Info.Data to actual pixel data, providing read-write access.

In case of video memory, the surface with data in video memory becomes mapped to system memory. An application can map a surface for read access with any value of mfxFrameSurface1::Data.Locked, but can map a surface for write access only when mfxFrameSurface1::Data.Locked equals to 0.

Note: A surface allows shared read access, but exclusive write access. Consider the following cases:

- Map with Write or Read|Write flags. A request during active another read or write access returns MFX_ERR_LOCK_MEMORY error immediately, without waiting. MFX_MAP_NOWAIT does not impact behavior. This type of request does not lead to any implicit synchronizations.
- Map with Read flag. A request during active write access will wait for resource to become free, or exits immediately with error if MFX_MAP_NOWAIT flag was set. This request may lead to the implicit synchronization (with same logic as Synchronize call) waiting for surface to become ready to use (all dependencies should be resolved and upstream components finished writing to this surface).

It is guaranteed that read access will be acquired right after synchronization without allowing another thread to acquire this surface for writing.

If MFX_MAP_NOWAIT was set and the surface is not ready yet (for example the surface has unresolved data dependencies or active processing), the read access request exits immediately with error.

Read-write access with MFX_MAP_READ_WRITE provides exclusive simultaneous reading and writing access.

Note Bitwise copying of *mfxFrameSurface1* object between map / unmap calls may result in having dangling data pointers in copies.

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNSUPPORTED If flags are invalid.

MFX_ERR_LOCK_MEMORY If user wants to map the surface for write and surface->Data.Locked does not equal to 0.

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid surface.
- `[out] flags`: Specify mapping mode.
- `[out] surface->Info.Data`: Pointers set to actual pixel data.

mfxStatus (***Unmap**) (*mfxFrameSurface1* *surface)

Invalidates pointers of surface->Info.Data and sets them to NULL. In case of video memory, the underlying texture becomes unmapped after last reader or writer unmap.

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX_ERR_UNSUPPORTED If surface is already unmapped.

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid surface.
- `[out] surface->Info.Data`: Pointers set to NULL.

mfxStatus (*__GetNativeHandle__) (*mfxFrameSurface1* *surface, *mfxHDL* *resource, *mfxResourceType* *resource_type)

Returns a native resource's handle and type. The handle is returned *as-is*, meaning that the reference counter of base resources is not incremented. The native resource is not detached from surface and the library still owns the resource. User must not destroy the native resource or assume that the resource will be alive after (*Release).

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If any of surface, resource or resource_type is NULL.

MFX_ERR_INVALID_HANDLE If any of surface, resource or resource_type is not valid object (no native resource was allocated).

MFX_ERR_UNSUPPORTED If surface is in system memory.

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid surface.
- `[out] resource`: Pointer is set to the native handle of the resource.
- `[out] resource_type`: Type of native resource. See mfxResourceType enumeration).

mfxStatus (*__GetDeviceHandle__) (*mfxFrameSurface1* *surface, *mfxHDL* *device_handle, *mfxHandleType* *device_type)

Returns a device abstraction that was used to create that resource. The handle is returned *as-is*, meaning that the reference counter for the device abstraction is not incremented. The native resource is not detached from the surface and the library still has a reference to the resource. User must not destroy the device or assume that the device will be alive after (*Release).

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If any of surface, device_handle or device_type is NULL.

MFX_ERR_INVALID_HANDLE If any of surface, resource or resource_type is not valid object (no native resource was allocated).

MFX_ERR_UNSUPPORTED If surface is in system memory.

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid surface.
- `[out] device_handle`: Pointer is set to the device which created the resource
- `[out] device_type`: Type of device (see mfxHandleType enumeration).

mfxStatus (*__Synchronize__) (*mfxFrameSurface1* *surface, *mfxU32* wait)

Guarantees readiness of both the data (pixels) and any frame's meta information (for example corruption flags) after a function completes.

Instead of MFXVideoCORE_SyncOperation, users may directly call the (*Synchronize) function after the corresponding Decode or VPP function calls (MFXVideoDECODE_DecodeFrameAsync or MFXVideoVPP_RunFrameVPPAsync). The prerequisites to call the functions are:

- The main processing functions return MFX_ERR_NONE.
- A valid *mfxFrameSurface1* object.

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If any of surface is not valid object .

MFX_WRN_IN_EXECUTION If the given timeout is expired and the surface is not ready.

MFX_ERR_ABORTED If the specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid surface.
- `[out] wait`: Wait time in milliseconds.

void (*`OnComplete`) (*mfxStatus* sts)

The library calls the function after complete of associated video operation notifying the application that frame surface is ready.

It is expected that the function is low-intrusive designed otherwise it may impact performance.

Attention This is callback function and intended to be called by the library only.

Parameters

- `[in] sts`: The status of completed operation.

Implementation Management

Structures used for implementation management.

API

- *mfxAdapterInfo*
- *mfxAdaptersInfo*
- *mfxExtThreadsParam*
- *mfxInitParam*
- *mfxPlatform*
- *mfxVersion*
- *mfxExtDeviceAffinityMask*
- *mfxInitializationParam*

mfxAdapterInfo

struct mfxAdapterInfo
Contains a description of the graphics adapter for the Legacy mode.

Public Members

mfxPlatform **Platform**
Platform type description. See *mfxPlatform* for details.

mfxU32 **Number**
Value which uniquely characterizes media adapter. On Windows* this number can be used for initialization through DXVA interface (see example).

mfxAdaptersInfo

struct mfxAdaptersInfo
Contains description of all graphics adapters available on the current system.

Public Members

mfxAdapterInfo ***Adapters**
Pointer to array of *mfxAdapterInfo* structs allocated by user.

mfxU32 **NumAlloc**
Length of Adapters array.

mfxU32 **NumActual**
Number of Adapters entries filled by MFXQueryAdapters.

mfxExtThreadsParam

struct mfxExtThreadsParam
Specifies options for threads created by this session. Attached to the *mfxInitParam* structure during legacy Intel(r) Media SDK session initialization.

Public Members

mfxExtBuffer **Header**
Must be MFX_EXTBUFF_THREADS_PARAM.

mfxU16 **NumThread**
The number of threads.

mfxI32 **SchedulingType**
Scheduling policy for all threads.

mfxI32 **Priority**
Priority for all threads.

mfxU16 **reserved**[55]
Reserved for future use.

mfxInitParam

struct mfxInitParam

Specifies advanced initialization parameters. A zero value in any of the fields indicates that the corresponding field is not explicitly specified.

Public Members

mfxIMPL **Implementation**

Enumerator that indicates the desired legacy Intel(r) Media SDK implementation.

mfxVersion **Version**

Structure which specifies minimum library version or zero, if not specified.

mfxU16 **ExternalThreads**

Desired threading mode. Value 0 means internal threading, 1 – external.

mfxExtBuffer ****ExtParam**

Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

mfxU16 **NumExtParam**

The number of extra configuration structures attached to this structure.

mfxU16 **GPUCopy**

Enables or disables GPU accelerated copying between video and system memory in legacy Intel(r) Media SDK components. See the GPUCopy enumerator for a list of valid values.

mfxPlatform

struct mfxPlatform

Contains information about hardware platform for the Legacy mode.

Public Members

mfxU16 **CodeName**

Microarchitecture code name. See the PlatformCodeName enumerator for a list of possible values.

mfxU16 **DeviceId**

Unique identifier of graphics device.

mfxU16 **MediaAdapterType**

Description of graphics adapter type. See the mfxMediaAdapterType enumerator for a list of possible values.

mfxU16 **reserved**[13]

Reserved for future use.

mfxVersion

union mfxVersion

#include <mfxcommon.h> The *mfxVersion* union describes the version of the implementation.

Major and Minor fields

Anonymous structure with Major and Minor fields.

mfxU16 **Minor**
Minor number of the implementation.

mfxU16 **Major**
Major number of the implementation.

Public Members

struct *mfxVersion*::**[anonymous] [anonymous]**

mfxU32 **Version**
Implementation version number.

mfxExtDeviceAffinityMask

struct mfxExtDeviceAffinityMask

The *mfxExtDeviceAffinityMask* structure is used by the application to specify affinity mask for the device with given device ID. See *mfxDeviceDescription* for the device ID definition and sub device indexes. If the implementation manages CPU threads for some purpose, the user can set the CPU thread affinity mask by using this structure with DeviceID set to "CPU".

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DEVICE_AFFINITY_MASK.

mfxChar **DeviceID**[**MFX_STRFIELD_LEN**]
Null terminated string with device ID. In case of CPU affinity mask it must be equal to "CPU".

mfxU32 **NumSubDevices**
Number of sub devices or threads in case of CPU in the mask.

mfxU8 ***Mask**
Mask array. Every bit represents sub-device (or thread for CPU). "1" means execution is allowed. "0" means that execution is prohibited on this sub-device (or thread). Length of the array is equal to the: "max_subdevices / 8" and rounded to the closest (from the right) integer. Bits order within each entry of the mask array is LSB: bit 0 holds data for sub device with index 0 and bit 8 for sub device with index 8. Index of sub device is defined by the *mfxDeviceDescription* structure.

mfxInitializationParam

struct mfxInitializationParam
Specifies initialization parameters for API version starting from 2.0.

Public Members

mfxAccelerationMode **AccelerationMode**
Hardware acceleration stack to use. OS dependent parameter. Use VA for Linux*, DX* for Windows* or HDDL.

mfxU16 **reserved**[3]
Reserved for future use.

mfxU16 **NumExtParam**
The number of extra configuration structures attached to this structure.

mfxExtBuffer ****ExtParam**
Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

mfxU32 **reserved2**[4]
Reserved for future use.

Cross-component Structures

Structures used across library components.

API

- *mfxComponentInfo*
- *mfxExtHEVCParam*
- *mfxExtJPEGHuffmanTables*
- *mfxExtJPEGQuantTables*
- *mfxExtMVCSeqDesc*
- *mfxExtMVCTargetViews*
- *mfxExtVideoSignalInfo*
- *mfxExtVP9Param*
- *mfxFrameId*
- *mfxInfoMFX*
- *mfxMVCOperationPoint*
- *mfxMVCViewDependency*
- *mfxPayload*
- *mfxVideoParam*

- *mfxVP9SegmentParam*
- *mfxExtAV1FilmGrainParam*
- *mfxAV1FilmGrainPoint*
- *mfxRect*

mfxComponentInfo

struct mfxComponentInfo

Contains workload description, which is accepted by MFXQueryAdapters function.

Public Members

mfxComponentType **Type**

Type of workload: Encode, Decode, VPP. See mfxComponentType enumerator for values.

mfxVideoParam **Requirements**

Detailed description of workload. See *mfxVideoParam* for details.

mfxExtHEVCParam

struct mfxExtHEVCParam

Public Members

mfxExtBuffer **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_HEVC_PARAM.

mfxU16 **PicWidthInLumaSamples**

Specifies the width of each coded picture in units of luma samples.

mfxU16 **PicHeightInLumaSamples**

Specifies the height of each coded picture in units of luma samples.

mfxU64 **GeneralConstraintFlags**

Additional flags to specify exact profile and constraints. See the GeneralConstraintFlags enumerator for values of this field.

mfxU16 **SampleAdaptiveOffset**

Controls SampleAdaptiveOffset encoding feature. See the SampleAdaptiveOffset enumerator for supported values (bit-ORed). Valid during encoder Init and Runtime.

mfxU16 **LCUSize**

Specifies largest coding unit size (max luma coding block). Valid during encoder Init.

mfxExtJPEGHuffmanTables

struct mfxExtJPEGHuffmanTables

Specifies Huffman tables. The application may specify up to 2 quantization table pairs for baseline process. The encoder assigns an ID to each table. That ID is equal to the table index in the DCTables and ACTables arrays. Table "0" is used for encoding of the Y component and table "1" is used for encoding of the U and V component. The application may specify only one table, in which case the table will be used for all components in the image. The following table illustrates this behavior.

Public Members

mfxExtBuffer **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_JPEG_HUFFMAN.

mfxU16 **NumDCTable**

Number of DC quantization table in DCTables array.

mfxU16 **NumACTable**

Number of AC quantization table in ACTables array.

mfxU8 **Bits**[16]

Number of codes for each code length.

mfxU8 **Values**[12]

List of the 8-bit symbol values.

Array of AC tables.

struct *mfxExtJPEGHuffmanTables*::**[anonymous]** **DCTables**[4]

Array of DC tables.

struct *mfxExtJPEGHuffmanTables*::**[anonymous]** **ACTables**[4]

List of the 8-bit symbol values.

Table ID	0	1
Number of tables		
0	Y, U, V	
1	Y	U, V

mfxExtJPEGQuantTables

struct mfxExtJPEGQuantTables

Specifies quantization tables. The application may specify up to 4 quantization tables. The encoder assigns an ID to each table. That ID is equal to the table index in the Qm array. Table "0" is used for encoding of the Y component, table "1" for the U component, and table "2" for the V component. The application may specify fewer tables than the number of components in the image. If two tables are specified, then table "1" is used for both U and V components. If only one table is specified then it is used for all components in the image. The following table illustrates this behavior.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_JPEG_QT.

mfxU16 **NumTable**
Number of quantization tables defined in Qmarray.

mfxU16 **Qm**[4][64]
Quantization table values.

Table ID	0	1	2
Number of tables			
0	Y, U, V		
1	Y	U, V	
2	Y	U	V

mfxExtMVCSeqDesc

struct mfxExtMVCSeqDesc
Describes the MVC stream information of view dependencies, view identifiers, and operation points. See the ITU*-T H.264 specification chapter H.7.3.2.1.4 for details.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MVC_SEQUENCE_DESCRIPTION.

mfxU32 **NumView**
Number of views.

mfxU32 **NumViewAlloc**
The allocated view dependency array size.

mfxMVCViewDependency ***View**
Pointer to a list of the *mfxMVCViewDependency*.

mfxU32 **NumViewId**
Number of view identifiers.

mfxU32 **NumViewIdAlloc**
The allocated view identifier array size.

mfxU16 ***ViewId**
Pointer to the list of view identifier.

mfxU32 **NumOP**
Number of operation points.

mfxU32 **NumOPAlloc**
The allocated operation point array size.

mfxMVCOperationPoint ***OP**
Pointer to a list of the *mfxMVCOperationPoint* structure.

mfxU16 **NumRefsTotal**
Total number of reference frames in all views required to decode the stream. This value is returned from the MFXVideoDECODE_Decodeheader function. Do not modify this value.

mfxExtMVCTargetViews

struct mfxExtMVCTargetViews
Configures views for the decoding output.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MVC_TARGET_VIEWS.

mfxU16 **TemporalId**
The temporal identifier to be decoded.

mfxU32 **NumView**
The number of views to be decoded.

mfxU16 **ViewId**[1024]
List of view identifiers to be decoded.

mfxExtVideoSignalInfo

struct mfxExtVideoSignalInfo
Defines the video signal information.

For H.264, see Annex E of the ISO/IEC 14496-10 specification for the definition of these parameters.

For MPEG-2, see section 6.3.6 of the ITU* H.262 specification for the definition of these parameters. The field VideoFullRange is ignored.

For VC-1, see section 6.1.14.5 of the SMPTE* 421M specification. The fields VideoFormat and VideoFullRange are ignored.

Note If ColourDescriptionPresent is zero, the color description information (including ColourPrimaries, TransferCharacteristics, and MatrixCoefficients) does not present in the bitstream.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VIDEO_SIGNAL_INFO.

mfxU16 **VideoFormat**

mfxU16 **VideoFullRange**

mfxU16 **ColourDescriptionPresent**

mfxU16 **ColourPrimaries**

mfxU16 **TransferCharacteristics**

mfxU16 **MatrixCoefficients**

mfxExtVP9Param

struct mfxExtVP9Param

Structure attached to the *mfxVideoParam* structure. Extends the *mfxVideoParam* structure with VP9-specific parameters. Used by both decoder and encoder.

Public Members

mfxExtBuffer **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VP9_PARAM.

mfxU16 **FrameWidth**

Width of the coded frame in pixels.

mfxU16 **FrameHeight**

Height of the coded frame in pixels.

mfxU16 **WriteIVFHeaders**

Set this option to ON to make the encoder insert IVF container headers to the output stream. The NumFrame field of the IVF sequence header will be zero. It is the responsibility of the application to update the NumFrame field with the correct value. See the CodingOptionValue enumerator for values of this option.

mfxI16 **QIndexDeltaLumaDC**

Specifies an offset for a particular quantization parameter.

mfxI16 **QIndexDeltaChromaAC**

Specifies an offset for a particular quantization parameter.

mfxI16 **QIndexDeltaChromaDC**

Specifies an offset for a particular quantization parameter.

mfxU16 **NumTileRows**

Number of tile rows. Should be power of two. The maximum number of tile rows is 4, per the VP9 specification. In addition, the maximum supported number of tile rows may depend on the underlying library implementation.

Use the Query API function to check if a particular pair of values (NumTileRows, NumTileColumns) is supported. In VP9, tile rows have dependencies and cannot be encoded or decoded in parallel. Therefore, tile rows are always encoded by the library in serial mode (one-by-one).

mfxU16 **NumTileColumns**

Number of tile columns. Should be power of two. Restricted with maximum and minimum tile width in luma pixels, as defined in the VP9 specification (4096 and 256 respectively). In addition, the maximum supported number of tile columns may depend on the underlying library implementation.

Use the Query API function to check if a particular pair of values (NumTileRows, NumTileColumns) is supported. In VP9, tile columns do not have dependencies and can be encoded/decoded in parallel. Therefore, tile columns can be encoded by the library in both parallel and serial modes.

Parallel mode is automatically utilized by the library when NumTileColumns exceeds 1 and does not exceed the number of tile coding engines on the platform. In other cases, serial mode is used. Parallel mode is capable of encoding more than 1 tile row (within limitations provided by VP9 specification and particular platform). Serial mode supports only tile grids 1xN and Nx1.

mfxFrameId

struct mfxFrameId
Describes the view and layer of a frame picture.

Public Members

mfxU16 **TemporalId**
The temporal identifier as defined in the annex H of the ITU*-T H.264 specification.

mfxU16 **PriorityId**
Reserved and must be zero.

mfxU16 **DependencyId**
Reserved for future use.

mfxU16 **QualityId**
Reserved for future use.

mfxU16 **ViewId**
The view identifier as defined in the annex H of the ITU-T H.264 specification.

mfxInfoMFX

struct mfxInfoMFX
Specifies configurations for decoding, encoding, and transcoding processes. A zero value in any of these fields indicates that the field is not explicitly specified.

Public Members

mfxU32 **reserved**[7]
Reserved for future use.

mfxU16 **LowPower**
For encoders, set this flag to ON to reduce power consumption and GPU usage. See the CodingOptionValue enumerator for values of this option. Use the Query API function to check if this feature is supported.

mfxU16 **BRCParamMultiplier**
Specifies a multiplier for bitrate control parameters. Affects the following variables: InitialDelayInKB, BufferSizeInKB, TargetKbps, MaxKbps. If this value is not equal to zero, the encoder calculates BRC parameters as `value * BRCParamMultiplier`.

mfxFrameInfo **FrameInfo**
mfxFrameInfo structure that specifies frame parameters.

mfxU32 **CodecId**
Specifies the codec format identifier in the FourCC code; see the CodecFormatFourCC enumerator for details. This is a mandated input parameter for the QueryIOSurf and Init API functions.

mfxU16 **CodecProfile**
Specifies the codec profile; see the CodecProfile enumerator for details. Specify the codec profile explicitly or the API functions will determine the correct profile from other sources, such as resolution and bitrate.

mfxU16 **CodecLevel**
Codec level; see the CodecLevel enumerator for details. Specify the codec level explicitly or the functions will determine the correct level from other sources, such as resolution and bitrate.

mfxU16 **TargetUsage**
Target usage model that guides the encoding process; see the TargetUsage enumerator for details.

mfxU16 **GopPicSize**
Number of pictures within the current GOP (Group of Pictures); if GopPicSize = 0, then the GOP size is unspecified. If GopPicSize = 1, only I-frames are used. The following pseudo-code that shows how the library uses this parameter:

```
mfxU16 get_gop_sequence (...) {
   pos=display_frame_order;
   if (pos == 0)
       return MFX_FRAMETYPE_I | MFX_FRAMETYPE_IDR | MFX_FRAMETYPE_REF;

   If (GopPicSize == 1) // Only I-frames
       return MFX_FRAMETYPE_I | MFX_FRAMETYPE_REF;

   if (GopPicSize == 0)
               frameInGOP = pos;    //Unlimited GOP
           else
               frameInGOP = pos%GopPicSize;

   if (frameInGOP == 0)
       return MFX_FRAMETYPE_I | MFX_FRAMETYPE_REF;

   if (GopRefDist == 1 || GopRefDist == 0)    // Only I,P frames
               return MFX_FRAMETYPE_P | MFX_FRAMETYPE_REF;

   frameInPattern = (frameInGOP-1)%GopRefDist;
   if (frameInPattern == GopRefDist - 1)
       return MFX_FRAMETYPE_P | MFX_FRAMETYPE_REF;

   return MFX_FRAMETYPE_B;
 }
```

mfxU16 **GopRefDist**
Distance between I- or P (or GPB) - key frames; if it is zero, the GOP structure is unspecified. Note: If GopRefDist = 1, there are no regular B-frames used (only P or GPB); if *mfxExtCodingOption3::GPB* is ON, GPB frames (B without backward references) are used instead of P.

mfxU16 **GopOptFlag**
ORs of the GopOptFlag enumerator indicate the additional flags for the GOP specification.

mfxU16 **IdrInterval**
For H.264, specifies IDR-frame interval in terms of I-frames. For example:

- If IdrInterval = 0, then every I-frame is an IDR-frame.
- If IdrInterval = 1, then every other I-frame is an IDR-frame.

For HEVC, if IdrInterval = 0, then only first I-frame is an IDR-frame. For example:

- If IdrInterval = 1, then every I-frame is an IDR-frame.
- If IdrInterval = 2, then every other I-frame is an IDR-frame.

For MPEG2, IdrInterval defines sequence header interval in terms of I-frames. For example:

- If IdrInterval = 0 (default), then the sequence header is inserted once at the beginning of the stream.
- If IdrInterval = N, then the sequence header is inserted before every Nth I-frame.

If GopPicSize or GopRefDist is zero, IdrInterval is undefined.

mfxU16 **InitialDelayInKB**
Initial size of the Video Buffering Verifier (VBV) buffer.

Note In this context, KB is 1000 bytes and Kbps is 1000 bps.

mfxU16 **QPI**
Quantization Parameter (QP) for I-frames for constant QP mode (CQP). Zero QP is not valid and means that the default value is assigned by the library. Non-zero QPI might be clipped to supported QPI range.

Note Default QPI value is implementation dependent and subject to change without additional notice in this document.

mfxU16 **Accuracy**
Specifies accuracy range in the unit of tenth of percent.

mfxU16 **BufferSizeInKB**
Represents the maximum possible size of any compressed frames.

mfxU16 **TargetKbps**
Constant bitrate TargetKbps. Used to estimate the targeted frame size by dividing the frame rate by the bitrate.

mfxU16 **QPP**
Quantization Parameter (QP) for P-frames for constant QP mode (CQP). Zero QP is not valid and means that the default value is assigned by the library. Non-zero QPP might be clipped to supported QPI range.

Note Default QPP value is implementation dependent and subject to change without additional notice in this document.

mfxU16 **ICQQuality**
Used by the Intelligent Constant Quality (ICQ) bitrate control algorithm. Values are in the 1 to 51 range, where 1 corresponds the best quality.

mfxU16 **MaxKbps**
The maximum bitrate at which the encoded data enters the Video Buffering Verifier (VBV) buffer.

mfxU16 **QPB**
Quantization Parameter (QP) for B-frames for constant QP mode (CQP). Zero QP is not valid and means that the default value is assigned by the library. Non-zero QPI might be clipped to supported QPB range.

Note Default QPB value is implementation dependent and subject to change without additional notice in this document.

mfxU16 **Convergence**
Convergence period in the unit of 100 frames.

mfxU16 **NumSlice**
Number of slices in each video frame. Each slice contains one or more macro-block rows. If NumSlice equals zero, the encoder may choose any slice partitioning allowed by the codec standard. See also *mfxExtCodingOption2::NumMbPerSlice*.

mfxU16 **NumRefFrame**
Max number of all available reference frames (for AVC/HEVC, NumRefFrame defines DPB size). If NumRefFrame = 0, this parameter is not specified. See also NumRefActiveP, NumRefActiveBL0, and NumRefActiveBL1 in the *mfxExtCodingOption3* structure, which set a number of active references.

mfxU16 **EncodedOrder**
If not zero, specifies that ENCODE takes the input surfaces in the encoded order and uses explicit frame type control. The application must still provide GopRefDist and *mfxExtCodingOption2::BRefType* so the library can pack headers and build reference lists correctly.

mfxU16 **DecodedOrder**
For AVC and HEVC, used to instruct the decoder to return output frames in the decoded order. Must be zero for all other decoders. When enabled, correctness of *mfxFrameData::TimeStamp* and FrameOrder for output surface is not guaranteed, the application should ignore them.

mfxU16 **ExtendedPicStruct**
Instructs DECODE to output extended picture structure values for additional display attributes. See the PicStruct description for details.

mfxU16 **TimeStampCalc**
Time stamp calculation method. See the TimeStampCalc description for details.

mfxU16 **SliceGroupsPresent**
Nonzero value indicates that slice groups are present in the bitstream. Used only by AVC decoder.

mfxU16 **MaxDecFrameBuffering**
Nonzero value specifies the maximum required size of the decoded picture buffer in frames for AVC and HEVC decoders.

mfxU16 **EnableReallocRequest**
For decoders supporting dynamic resolution change (VP9), set this option to ON to allow MFXVideoDECODE_DecodeFrameAsync return MFX_ERR_REALLOC_SURFACE. See the CodingOptionValue enumerator for values of this option. Use the Query API function to check if this feature is supported.

mfxU16 **FilmGrain**
Special parameter for AV1 decoder. Indicates presence/absence of film grain parameters in bitstream. Also controls decoding behavior for streams with film grain parameters. MFXVideoDECODE_DecodeHeader returns nonzero FilmGrain for streams with film grain parameters and zero for streams w/o them. Decoding with film grain requires additional output surfaces. If FilmGrain` is non-zero then MFXVideoDECODE_QueryIOSurf will request more surfaces in case of external allocated video memory at decoder output. FilmGrain is passed to MFXVideoDECODE_Init function to control decoding operation for AV1 streams with film grain parameters. If FilmGrain is nonzero decoding of each frame require two output surfaces (one for reconstructed frame and one for output frame with film grain applied). The decoder returns MFX_ERR_MORE_SURFACE from MFXVideoDECODE_DecodeFrameAsync if it has insufficient output surfaces to decode frame. Application can forcibly disable the feature passing zero value of `FilmGrain` to `MFXVideoDECODE_Init`. In this case the decoder will output reconstructed frames w/o film grain applied. Application can retrieve film grain parameters for a frame by attaching extended buffer *mfxExtAV1FilmGrainParam* to *mfxFrameSurface1*. If stream has no film grain parameters `FilmGrain` passed to `MFXVideoDECODE_Init` is ignored by the decoder.

mfxU16 **IgnoreLevelConstrain**
If not zero, it forces SDK to attempt to decode bitstream even if a decoder may not support all features associated with given CodecLevel. Decoder may produce visual artifacts. Only AVC decoder supports this field.

mfxU16 **SkipOutput**
This flag is used to disable output of main decoding channel. When it's ON SkipOutput = MFX_CODINGOPTION_ON decoder outputs only video processed channels. For pure decode this flag should be always disabled.

mfxU16 **JPEGChromaFormat**
Specify the chroma sampling format that has been used to encode a JPEG picture. See the ChromaFormat enumerator for details.

mfxU16 **Rotation**
Rotation option of the output JPEG picture. See the Rotation enumerator for details.

mfxU16 **JPEGColorFormat**

Specify the color format that has been used to encode a JPEG picture. See the JPEGColorFormat enumerator for details.

mfxU16 **InterleavedDec**
Specify JPEG scan type for decoder. See the JPEGScanType enumerator for details.

mfxU8 **SamplingFactorH**[4]
Horizontal sampling factor.

mfxU8 **SamplingFactorV**[4]
Vertical sampling factor.

mfxU16 **Interleaved**
Specify interleaved or non-interleaved scans. If it is equal to MFX_SCANTYPE_INTERLEAVED then the image is encoded as interleaved, all components are encoded in one scan. See the JPEG Scan Type enumerator for details.

mfxU16 **Quality**
Specifies the image quality if the application does not specified quantization table. The value is from 1 to 100 inclusive. "100" is the best quality.

mfxU16 **RestartInterval**
Specifies the number of MCU in the restart interval. "0" means no restart interval.

Note: The *mfxInfoMFX::InitialDelayInKB*, *mfxInfoMFX::TargetKbps*, *mfxInfoMFX::MaxKbps* parameters are used by the constant bitrate (CBR), variable bitrate control (VBR), and CQP HRD algorithms.

Encoders follow the Hypothetical Reference Decoding (HRD) model. The HRD model assumes that data flows into a buffer of the fixed size BufferSizeInKB with a constant bitrate of TargetKbps. (Estimate the targeted frame size by dividing frame rate by bitrate.)

The decoder starts decoding after the buffer reaches the initial size InitialDelayInKB, which is equivalent to reaching an initial delay of InitialDelayInKB*8000/TargetKbpsms. *In this context, KB is 1000 bytes and Kbps is 1000 bps.*

If InitialDelayInKB or BufferSizeInKB is equal to zero, the value is calculated using bitrate, frame rate, profile, level, and so on.

TargetKbps must be specified for encoding initialization.

For variable bitrate control, the MaxKbps parameter specifies the maximum bitrate at which the encoded data enters the Video Buffering Verifier (VBV) buffer. If MaxKbps is equal to zero, the value is calculated from bitrate, frame rate, profile, and level.

Note: The *mfxInfoMFX::TargetKbps*, *mfxInfoMFX::Accuracy*, *mfxInfoMFX::Convergence* parameters are used by the average variable bitrate control (AVBR) algorithm. The algorithm focuses on overall encoding quality while meeting the specified bitrate, TargetKbps, within the accuracy range, Accuracy, after a Convergence period. This method does not follow HRD and the instant bitrate is not capped or padded.

---

mfxMVCOperationPoint

struct mfxMVCOperationPoint
Describes the MVC operation point.

Public Members

mfxU16 **TemporalId**
Temporal identifier of the operation point.

mfxU16 **LevelIdc**
Level value signaled for the operation point.

mfxU16 **NumViews**
Number of views required for decoding the target output views that correspond to the operation point.

mfxU16 **NumTargetViews**
Number of target output views for the operation point.

mfxU16 ***TargetViewId**
Target output view identifiers for operation point.

mfxMVCViewDependency

struct mfxMVCViewDependency
Describes MVC view dependencies.

Public Members

mfxU16 **ViewId**
View identifier of this dependency structure.

mfxU16 **NumAnchorRefsL0**
Number of view components for inter-view prediction in the initial reference picture list RefPicList0 for anchor view components.

mfxU16 **NumAnchorRefsL1**
Number of view components for inter-view prediction in the initial reference picture list RefPicList1 for anchor view components.

mfxU16 **AnchorRefL0**[16]
View identifiers of the view components for inter-view prediction in the initial reference picture list RefPicList0 for anchor view components.

mfxU16 **AnchorRefL1**[16]
View identifiers of the view components for inter-view prediction in the initial reference picture list RefPicList1 for anchor view components.

mfxU16 **NumNonAnchorRefsL0**
Number of view components for inter-view prediction in the initial reference picture list RefPicList0 for non-anchor view components.

mfxU16 **NumNonAnchorRefsL1**
Number of view components for inter-view prediction in the initial reference picture list RefPicList1 for non-anchor view components.

mfxU16 **NonAnchorRefL0**[16]
View identifiers of the view components for inter-view prediction in the initial reference picture list RefPicList0 for non-anchor view components.

mfxPayload

struct mfxPayload
Describes user data payload in MPEG-2 or SEI message payload in H.264.

For encoding, these payloads can be inserted into the bitstream. The payload buffer must contain a valid formatted payload.

For H.264, this is the sei_message() as specified in the section 7.3.2.3.1 'Supplemental enhancement information message syntax' of the ISO/IEC 14496-10 specification.

For MPEG-2, this is the section 6.2.2.2.2 'User data' of the ISO/IEC 13818-2 specification, excluding the user data start_code.

For decoding, these payloads can be retrieved as the decoder parses the bitstream and caches them in an internal buffer.

Public Members

mfxU32 **CtrlFlags**
Additional payload properties. See the PayloadCtrlFlags enumerator for details.

mfxU8 ***Data**
Pointer to the actual payload data buffer.

mfxU32 **NumBit**
Number of bits in the payload data

mfxU16 **Type**
MPEG-2 user data start code or H.264 SEI message type.

mfxU16 **BufSize**
Payload buffer size in bytes.

Codec	Supported Types
MPEG2	0x01B2 //User Data
AVC	02 //pan_scan_rect 03 //filler_payload 04 //user_data_registered_itu_t_t35 05 //user_data_unregistered 06 //recovery_point 09 //scene_info 13 //full_frame_freeze 14 //full_frame_freeze_release 15 //full_frame_snapshot 16 //progressive_refinement_segment_start 17 //progressive_refinement_segment_end 19 //film_grain_characteristics 20 //deblocking_filter_display_preference 21 //stereo_video_info 45 //frame_packing_arrangement
HEVC	All

mfxVideoParam

struct mfxVideoParam
Configuration parameters for encoding, decoding, transcoding, and video processing.

Public Members

mfxU32 **AllocId**
Unique component ID that will be passed by the library to *mfxFrameAllocRequest*. Useful in pipelines where several components of the same type share the same allocator.

mfxU16 **AsyncDepth**
Specifies how many asynchronous operations an application performs before the application explicitly synchronizes the result. If zero, the value is not specified.

mfxInfoMFX **mfx**
Configurations related to encoding, decoding, and transcoding. See the definition of the *mfxInfoMFX* structure for details.

mfxInfoVPP **vpp**
Configurations related to video processing. See the definition of the *mfxInfoVPP* structure for details.

mfxU16 **Protected**
Specifies the content protection mechanism. See the Protected enumerator for a list of supported protection schemes.

mfxU16 **IOPattern**
Input and output memory access types for functions. See the enumerator IOPattern for details. The Query API functions return the natively supported IOPattern if the Query input argument is NULL. This parameter is a mandated input for QueryIOSurf and Init API functions. The output pattern must be specified for DECODE. The input pattern must be specified for ENCODE. Both input and output pattern must be specified for VPP.

mfxExtBuffer ****ExtParam**
The number of extra configuration structures attached to this structure.

mfxU16 **NumExtParam**
Points to an array of pointers to the extra configuration structures. See the ExtendedBufferID enumerator for a list of extended configurations. The list of extended buffers should not contain duplicated entries, such as entries of the same type. If the *mfxVideoParam* structure is used to query library capability, then the list of extended buffers attached to the input and output *mfxVideoParam* structure should be equal, that is, it should contain the same number of extended buffers of the same type.

mfxVP9SegmentParam

struct mfxVP9SegmentParam
Contains features and parameters for the segment.

Public Members

mfxU16 **FeatureEnabled**
Indicates which features are enabled for the segment. See the SegmentFeature enumerator for values for this option. Values from the enumerator can be bit-OR'ed. Support of a particular feature depends on underlying hardware platform. Application can check which features are supported by calling Query.

mfxI16 **QIndexDelta**
Quantization index delta for the segment. Ignored if MFX_VP9_SEGMENT_FEATURE_QINDEX isn't set in FeatureEnabled. Valid range for this parameter is [-255, 255]. If QIndexDelta is out of this range, it will be ignored. If QIndexDelta is within valid range, but sum of base quantization index and QIndexDelta is out of [0, 255], QIndexDelta will be clamped.

mfxI16 **LoopFilterLevelDelta**
Loop filter level delta for the segment. Ignored if MFX_VP9_SEGMENT_FEATURE_LOOP_FILTER is not set in FeatureEnabled. Valid range for this parameter is [-63, 63]. If LoopFilterLevelDelta is out of this range, it will be ignored. If LoopFilterLevelDelta is within valid range, but sum of base loop filter level and LoopFilterLevelDelta is out of [0, 63], LoopFilterLevelDelta will be clamped.

mfxU16 **ReferenceFrame**
Reference frame for the segment. See VP9ReferenceFrame enumerator for values for this option. Ignored if MFX_VP9_SEGMENT_FEATURE_REFERENCE isn't set in FeatureEnabled.

mfxExtAV1FilmGrainParam

struct mfxExtAV1FilmGrainParam
The structure is used by AV-1 decoder to report film grain parameters for decoded frame.

Public Members

mfxU16 **FilmGrainFlags**
Bit map with bit-ORed flags from FilmGrainFlags enum.

mfxU16 **GrainSeed**
Starting value for pseudo-random numbers used during film grain synthesis.

mfxU8 **RefIdx**
Indicate which reference frame contains the film grain parameters to be used for this frame.

mfxU8 **NumYPoints**
The number of points for the piece-wise linear scaling function of the luma component.

mfxU8 **NumCbPoints**
The number of points for the piece-wise linear scaling function of the Cb component.

mfxU8 **NumCrPoints**
The number of points for the piece-wise linear scaling function of the Cr component.

mfxAV1FilmGrainPoint **PointY**[14]
The array of points for luma component.

mfxAV1FilmGrainPoint **PointCb**[10]
The array of points for Cb component.

mfxAV1FilmGrainPoint **PointCr**[10]
The array of points for Cr component.

mfxU8 **GrainScalingMinus8**
The shift – 8 applied to the values of the chroma component. The grain_scaling_minus_8 can take values of 0..3 and determines the range and quantization step of the standard deviation of film grain.

mfxU8 **ArCoeffLag**
The number of auto-regressive coefficients for luma and chroma.

mfxU8 **ArCoeffsYPlus128**[24]
Auto-regressive coefficients used for the Y plane.

mfxU8 **ArCoeffsCbPlus128**[25]
Auto-regressive coefficients used for the Cb plane.

mfxU8 **ArCoeffsCrPlus128**[25]
The number of points for the piece-wise linear scaling function of the Cr component.

mfxU8 **ArCoeffShiftMinus6**
The range of the auto-regressive coefficients. Values of 0, 1, 2, and 3 correspond to the ranges for auto-regressive coefficients of [-2, 2), [-1, 1), [-0.5, 0.5) and [-0.25, 0.25) respectively.

mfxU8 **GrainScaleShift**
Downscaling factor of the grain synthesis process for the Gaussian random numbers .

mfxU8 **CbMult**
The multiplier for the Cb component used in derivation of the input index to the Cb component scaling function.

mfxU8 **CbLumaMult**
The multiplier for the average luma component used in derivation of the input index to the Cb component scaling function.

mfxU16 **CbOffset**
The offset used in derivation of the input index to the Cb component scaling function.

mfxU8 **CrMult**
The multiplier for the Cr component used in derivation of the input index to the Cr component scaling function.

mfxU8 **CrLumaMult**
The multiplier for the average luma component used in derivation of the input index to the Cr component scaling function.

mfxU16 **CrOffset**
The offset used in derivation of the input index to the Cr component scaling function.

mfxAV1FilmGrainPoint

struct mfxAV1FilmGrainPoint
Defines film grain point.

Public Members

mfxU8 **Value**
The x coordinate for the i-th point of the piece-wise linear scaling function for luma/Cb/Cr component.

mfxU8 **Scaling**
The scaling (output) value for the i-th point of the piecewise linear scaling function for luma/Cb/Cr component.

mfxRect

struct mfxRect
The structure describes rectangle coordinates wat can bse used for ROI or for Cropping.

Public Members

mfxU16 **Left**
X coordinate of region of top-left corner of rectangle.

mfxU16 **Top**
Y coordinate of region of top-left corner of rectangle.

mfxU16 **Right**
X coordinate of region of bottom-right corner of rectangle.

mfxU16 **Bottom**
Y coordinate of region of bottom-right corner of rectangle.

Decode Structures

Structures used by Decode only.

API

- *mfxDecodeStat*
- *mfxExtDecodeErrorReport*
- *mfxExtDecodedFrameInfo*
- *mfxExtTimeCode*

mfxDecodeStat

struct mfxDecodeStat
Returns statistics collected during decoding.

Public Members

mfxU32 **NumFrame**
Number of total decoded frames.

mfxU32 **NumSkippedFrame**
Number of skipped frames.

mfxU32 **NumError**
Number of errors recovered.

mfxU32 **NumCachedFrame**
Number of internally cached frames.

mfxExtDecodeErrorReport

struct mfxExtDecodeErrorReport
Used by the decoders to report bitstream error information right after DecodeHeader or DecodeFrameAsync. The application can attach this extended buffer to the *mfxBitstream* structure at runtime.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DECODE_ERROR_REPORT.

mfxU32 **ErrorTypes**
Bitstream error types (bit-ORed values). See ErrorTypes enumerator for the list of types.

mfxExtDecodedFrameInfo

struct mfxExtDecodedFrameInfo
Used by the decoders to report additional information about a decoded frame. The application can attach this extended buffer to the mfxFrameSurface1::mfxFrameData structure at runtime.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DECODED_FRAME_INFO.

mfxU16 **FrameType**
Frame type. See FrameType enumerator for the list of types.

mfxExtTimeCode

struct mfxExtTimeCode

Used by the library to pass MPEG 2 specific timing information.

See ISO/IEC 13818-2 and ITU-T H.262, MPEG-2 Part 2 for the definition of these parameters.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_TIME_CODE.

mfxU16 **DropFrameFlag**
Indicated dropped frame.

mfxU16 **TimeCodeHours**
Hours.

mfxU16 **TimeCodeMinutes**
Minutes.

mfxU16 **TimeCodeSeconds**
Seconds.

mfxU16 **TimeCodePictures**
Pictures.

Encode Structures

Structures used by Encode only.

API

- *mfxBRCFrameCtrl*
- *mfxBRCFrameParam*
- *mfxBRCFrameStatus*
- *mfxEncodeCtrl*
- *mfxEncodedUnitInfo*
- *mfxEncodeStat*
- *mfxExtAVCEncodedFrameInfo*
- *mfxExtAVCRefListCtrl*
- *mfxExtAVCRefLists*
- *mfxExtAVCRoundingOffset*
- *mfxExtAvcTemporalLayers*
- *mfxExtBRC*
- *mfxExtChromaLocInfo*

- *mfxExtCodingOption*
- *mfxExtCodingOption2*
- *mfxExtCodingOption3*
- *mfxExtCodingOptionSPSPPS*
- *mfxExtCodingOptionVPS*
- *mfxExtContentLightLevelInfo*
- *mfxExtDirtyRect*
- *mfxExtEncodedUnitsInfo*
- *mfxExtEncoderCapability*
- *mfxExtEncoderIPCMArea*
- *mfxExtEncoderResetOption*
- *mfxExtEncoderROI*
- *mfxExtHEVCRegion*
- *mfxExtHEVCTiles*
- *mfxExtInsertHeaders*
- *mfxExtMasteringDisplayColourVolume*
- *mfxExtMBDisableSkipMap*
- *mfxExtMBForceIntra*
- *mfxExtMBQP*
- *mfxExtMoveRect*
- *mfxExtMVOverPicBoundaries*
- *mfxExtPartialBitstreamParam*
- *mfxExtPictureTimingSEI*
- *mfxExtPredWeightTable*
- *mfxExtVP8CodingOption*
- *mfxExtVP9Segmentation*
- *mfxExtVP9TemporalLayers*
- *mfxQPandMode*
- *mfxVP9TemporalLayer*

mfxBRCFrameCtrl

struct mfxBRCFrameCtrl
Specifies controls for next frame encoding provided by external BRC functions.

Public Members

mfxI32 **QpY**
Frame-level Luma QP.

mfxU32 **InitialCpbRemovalDelay**
See initial_cpb_removal_delay in codec standard. Ignored if no HRD control: *mfxExtCodingOption::VuiNalHrdParameters* = MFX_CODINGOPTION_OFF. Calculated by encoder if initial_cpb_removal_delay==0 && initial_cpb_removal_offset == 0 && HRD control is switched on.

mfxU32 **InitialCpbRemovalOffset**
See initial_cpb_removal_offset in codec standard. Ignored if no HRD control: *mfxExtCodingOption::VuiNalHrdParameters* = MFX_CODINGOPTION_OFF. Calculated by encoder if initial_cpb_removal_delay==0 && initial_cpb_removal_offset == 0 && HRD control is switched on.

mfxU32 **MaxFrameSize**
Max frame size in bytes. Option for repack feature. Driver calls PAK until current frame size is less than or equal to maxFrameSize, or number of repacking for this frame is equal to maxNumRePak. Repack is available if there is driver support, MaxFrameSize !=0, and MaxNumRePak != 0. Ignored if maxNumRePak == 0.

mfxU8 **DeltaQP**[8]
Option for repack feature. Ignored if maxNumRePak == 0 or maxNumRePak==0. If current frame size > maxFrameSize and/or number of repacking (nRepack) for this frame <= maxNumRePak, PAK is called with QP = *mfxBRCFrameCtrl::QpY* + Sum(DeltaQP[i]), where i = [0,nRepack]. Non zero DeltaQP[nRepack] are ignored if nRepack > maxNumRePak. If repacking feature is on (maxFrameSize & maxNumRePak are not zero), it is calculated by the encoder.

mfxU16 **MaxNumRepak**
Number of possible repacks in driver if current frame size > maxFrameSize. Ignored if maxFrameSize==0. See maxFrameSize description. Possible values are in the range of 0 to 8.

mfxU16 **NumExtParam**
Reserved for future use.

mfxExtBuffer ****ExtParam**
Reserved for future use.

mfxBRCFrameParam

struct mfxBRCFrameParam
Describes frame parameters required for external BRC functions.

Public Members

mfxU16 **SceneChange**
Frame belongs to a new scene if non zero.

mfxU16 **LongTerm**
Frame is a Long Term Reference frame if non zero.

mfxU32 **FrameCmplx**
Frame Complexity Frame spatial complexity if non zero. Zero if complexity is not available.

mfxU32 **EncodedOrder**
The frame number in a sequence of reordered frames starting from encoder Init.

mfxU32 **DisplayOrder**
The frame number in a sequence of frames in display order starting from last IDR.

mfxU32 **CodedFrameSize**
Size of the frame in bytes after encoding.

mfxU16 **FrameType**
Frame type. See FrameType enumerator for possible values.

mfxU16 **PyramidLayer**
B-pyramid or P-pyramid layer that the frame belongs to.

mfxU16 **NumRecode**
Number of recodings performed for this frame.

mfxU16 **NumExtParam**
Reserved for future use.

mfxExtBuffer ****ExtParam**
Reserved for future use.

Frame spatial complexity is calculated according to the following formula:

$$R = \frac{16}{WH}\sum_{k=0}^{\frac{W}{4}-1}\sum_{l=0}^{\frac{H}{4}-1}\left[\frac{\sum_{i=0}^{3}\sum_{j=0}^{3}|P[k*4+i][l*4+j]-P[k*4+i-1][l*4+j]|}{16}\right]$$

$$C = \frac{16}{WH}\sum_{k=0}^{\frac{W}{4}-1}\sum_{l=0}^{\frac{H}{4}-1}\left[\frac{\sum_{i=0}^{3}\sum_{j=0}^{3}|P[k*4+i][l*4+j]-P[k*4+i][l*4+j-1]|}{16}\right]$$

$$FrameCmplx = \sqrt{R^2 + C^2}$$

mfxBRCFrameStatus

struct mfxBRCFrameStatus
Specifies instructions for the encoder provided by external BRC after each frame encoding. See the BRCStatus enumerator for details.

Public Members

mfxU32 **MinFrameSize**
Size in bytes, coded frame must be padded to when Status = MFX_BRC_PANIC_SMALL_FRAME.

mfxU16 **BRCStatus**
BRC status. See the BRCStatus enumerator for possible values.

mfxEncodeCtrl

struct mfxEncodeCtrl
Contains parameters for per-frame based encoding control.

Public Members

mfxExtBuffer **Header**
Extension buffer header.

mfxU16 **MfxNalUnitType**
Type of NAL unit that contains encoding frame. All supported values are defined by MfxNalUnitType enumerator. Other values defined in ITU-T H.265 specification are not supported.

The encoder uses this field only if application sets *mfxExtCodingOption3::EnableNalUnitType* option to ON during encoder initialization.

Note Only encoded order is supported. If application specifies this value in display order or uses value inappropriate for current frame or invalid value, then the encoder silently ignores it.

mfxU16 **SkipFrame**
Indicates that current frame should be skipped or the number of missed frames before the current frame. See *mfxExtCodingOption2::SkipFrame* for details.

mfxU16 **QP**
If nonzero, this value overwrites the global QP value for the current frame in the constant QP mode.

mfxU16 **FrameType**
Encoding frame type. See the FrameType enumerator for details. If the encoder works in the encoded order, the application must specify the frame type. If the encoder works in the display order, only key frames are enforceable.

mfxU16 **NumExtParam**
Number of extra control buffers.

mfxU16 **NumPayload**
Number of payload records to insert into the bitstream.

mfxExtBuffer ****ExtParam**
Pointer to an array of pointers to external buffers that provide additional information or control to the encoder for this frame or field pair. A typical use is to pass the VPP auxiliary data generated by the video processing pipeline to the encoder. See the ExtendedBufferID for the list of extended buffers.

mfxPayload ****Payload**
Pointer to an array of pointers to user data (MPEG-2) or SEI messages (H.264) for insertion into the bitstream. For field pictures, odd payloads are associated with the first field and even payloads are associated with the second field. See the *mfxPayload* structure for payload definitions.

mfxEncodedUnitInfo

struct mfxEncodedUnitInfo
Used to report encoded unit information.

Public Members

mfxU16 **Type**
Codec-dependent coding unit type (NALU type for AVC/HEVC, start_code for MPEG2 etc).

mfxU32 **Offset**
Offset relative to the associated *mfxBitstream::DataOffset*.

mfxU32 **Size**
Unit size, including delimiter.

mfxEncodeStat

struct mfxEncodeStat
Returns statistics collected during encoding.

Public Members

mfxU32 **NumFrame**
Number of encoded frames.

mfxU64 **NumBit**
Number of bits for all encoded frames.

mfxU32 **NumCachedFrame**
Number of internally cached frames.

mfxExtAVCEncodedFrameInfo

struct mfxExtAVCEncodedFrameInfo
Used by the encoder to report additional information about the encoded picture. The application can attach this buffer to the *mfxBitstream* structure before calling MFXVideoENCODE_EncodeFrameAsync function. For interlaced content the encoder requires two such structures. They correspond to fields in encoded order.

Note Not all implementations of the encoder support this extended buffer. The application must use query mode 1 to determine if the functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE then the functionality is supported.

Reference Lists

The following structure members are used by the reference lists contained in the parent structure.

mfxU32 **FrameOrder**
Frame order of encoded picture.

Frame order of reference picture.

mfxU16 **PicStruct**
Picture structure of encoded picture.

Picture structure of reference picture.

mfxU16 **LongTermIdx**
Long term index of encoded picture if applicable.

Long term index of reference picture if applicable.

mfxU16 **reserved**[2]

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODED_FRAME_INFO.

mfxU32 **MAD**
Mean Absolute Difference between original pixels of the frame and motion compensated (for inter macroblocks) or spatially predicted (for intra macroblocks) pixels. Only luma component, Y plane, is used in calculation.

mfxU16 **BRCPanicMode**
Bitrate control was not able to allocate enough bits for this frame. Frame quality may be unacceptably low.

mfxU16 **QP**
Luma QP.

mfxU32 **SecondFieldOffset**
Offset to second field. Second field starts at *mfxBitstream::Data* + *mfxBitstream::DataOffset* + *mfxExtAVCEncodedFrameInfo::SecondFieldOffset*.

struct *mfxExtAVCEncodedFrameInfo*::**[anonymous]** **UsedRefListL0**[32]
Reference list that has been used to encode picture.

struct *mfxExtAVCEncodedFrameInfo*::**[anonymous]** **UsedRefListL1**[32]
Reference list that has been used to encode picture.

mfxExtAVCRefListCtrl

struct mfxExtAVCRefListCtrl
Configures reference frame options for the H.264 encoder.

See the *Reference List Selection* and *Long Term Reference Frame* sections for more details.

Note Not all implementations of the encoder support LongTermIdx and ApplyLongTermIdx fields in this structure. The application must use query mode 1 to determine if such functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE and these fields were set to one, then

the functionality is supported. If the function fails or sets fields to zero, then the functionality is not supported.

Reference Lists

The following structure members are used by the reference lists contained in the parent structure.

mfxU32 **FrameOrder**
Together FrameOrder and PicStruct fields are used to identify reference picture. Use FrameOrder = MFX_FRAMEORDER_UNKNOWN to mark unused entry.

mfxU16 **PicStruct**
Together FrameOrder and PicStruct fields are used to identify reference picture. Use FrameOrder = MFX_FRAMEORDER_UNKNOWN to mark unused entry.

mfxU16 **ViewId**
Reserved and must be zero.

mfxU16 **LongTermIdx**
Index that should be used by the encoder to mark long-term reference frame.

mfxU16 **reserved**[3]
Reserved

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AVC_REFLIST_CTRL.

mfxU16 **NumRefIdxL0Active**
Specify the number of reference frames in the active reference list L0. This number should be less or equal to the NumRefFrame parameter from encoding initialization.

mfxU16 **NumRefIdxL1Active**
Specify the number of reference frames in the active reference list L1. This number should be less or equal to the NumRefFrame parameter from encoding initialization.

struct *mfxExtAVCRefListCtrl*::**[anonymous]** **PreferredRefList**[32]
Reference list that specifies the list of frames that should be used to predict the current frame.

struct *mfxExtAVCRefListCtrl*::**[anonymous]** **RejectedRefList**[16]
Reference list that specifies the list of frames that should not be used for prediction.

struct *mfxExtAVCRefListCtrl*::**[anonymous]** **LongTermRefList**[16]
Reference list that specifies the list of frames that should be marked as long-term reference frame.

mfxU16 **ApplyLongTermIdx**
If it is equal to zero, the encoder assigns long-term index according to internal algorithm. If it is equal to one, the encoder uses LongTermIdx value as long-term index.

mfxExtAVCRefLists

struct mfxExtAVCRefLists

Specifies reference lists for the encoder. It may be used together with the *mfxExtAVCRefListCtrl* structure to create customized reference lists. If both structures are used together, then the encoder takes reference lists from the *mfxExtAVCRefLists* structure and modifies them according to the *mfxExtAVCRefListCtrl* instructions. In case of interlaced coding, the first *mfxExtAVCRefLists* structure affects TOP field and the second – BOTTOM field.

Note Not all implementations of the encoder support this structure. The application must use the Query API function to determine if it is supported.

Public Members

mfxExtBuffer **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AVC_REFLISTS.

mfxU16 **NumRefIdxL0Active**

Specify the number of reference frames in the active reference list L0. This number should be less than or equal to the NumRefFrame parameter from encoding initialization.

mfxU16 **NumRefIdxL1Active**

Specify the number of reference frames in the active reference list L1. This number should be less than or equal to the NumRefFrame parameter from encoding initialization.

struct *mfxExtAVCRefLists*::*mfxRefPic* **RefPicList0**[32]

Specify L0 reference list.

struct *mfxExtAVCRefLists*::*mfxRefPic* **RefPicList1**[32]

Specify L1 reference list.

struct mfxRefPic

Used by the reference lists contained in the parent structure. Together these fields are used to identify reference picture.

Public Members

mfxU32 **FrameOrder**

Use FrameOrder = MFX_FRAMEORDER_UNKNOWN to mark unused entry.

mfxU16 **PicStruct**

Use PicStruct = MFX_PICSTRUCT_FIELD_TFF for TOP field, PicStruct = MFX_PICSTRUCT_FIELD_BFF for BOTTOM field.

mfxExtAVCRoundingOffset

struct mfxExtAVCRoundingOffset

Used by encoders to set rounding offset parameters for quantization. It is per-frame based encoding control, and can be attached to some frames and skipped for others. When the extension buffer is set the application can attach it to the *mfxEncodeCtrl* during runtime.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AVC_ROUNDING_OFFSET.

mfxU16 **EnableRoundingIntra**
Enable rounding offset for intra blocks. See the CodingOptionValue enumerator for values of this option.

mfxU16 **RoundingOffsetIntra**
Intra rounding offset. Value must be in the range of 0 to 7, inclusive.

mfxU16 **EnableRoundingInter**
Enable rounding offset for inter blocks. See the CodingOptionValue enumerator for values of this option.

mfxU16 **RoundingOffsetInter**
Inter rounding offset. Value must be in the range of 0 to 7, inclusive.

mfxExtAvcTemporalLayers

struct mfxExtAvcTemporalLayers
Configures the H.264 temporal layers hierarchy.

If the application attaches it to the *mfxVideoParam* structure during initialization, the encoder generates the temporal layers and inserts the prefix NAL unit before each slice to indicate the temporal and priority IDs of the layer.

This structure can be used with the display-order encoding mode only.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_AVC_TEMPORAL_LAYERS.

mfxU16 **BaseLayerPID**
The priority ID of the base layer. The encoder increases the ID for each temporal layer and writes to the prefix NAL unit.

mfxU16 **Scale**
The ratio between the frame rates of the current temporal layer and the base layer.

mfxExtBRC

struct mfxExtBRC
Contains a set of callbacks to perform external bitrate control. Can be attached to the *mfxVideoParam* structure during encoder initialization. Set the *mfxExtCodingOption2::ExtBRC* option to ON to make the encoder use the external BRC instead of the native one.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_BRC.

mfxHDL **pthis**
Pointer to the BRC object.

mfxStatus (***Init**) (*mfxHDL* pthis, *mfxVideoParam* *par)
Initializes the BRC session according to parameters from input *mfxVideoParam* and attached structures. It does not modify the input *mfxVideoParam* and attached structures. Invoked during MFXVideoENCODE_Init.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_UNSUPPORTED The function detected unsupported video parameters.

Parameters

- `[in] pthis`: Pointer to the BRC object.
- `[in] par`: Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

mfxStatus (***Reset**) (*mfxHDL* pthis, *mfxVideoParam* *par)
Resets BRC session according to new parameters. It does not modify the input *mfxVideoParam* and attached structures. Invoked during MFXVideoENCODE_Reset.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_UNSUPPORTED The function detected unsupported video parameters.

MFX_ERR_INCOMPATIBLE_VIDEO_PARAM The function detected that the video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed.

Parameters

- `[in] pthis`: Pointer to the BRC object.
- `[in] par`: Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

mfxStatus (***Close**) (*mfxHDL* pthis)
Deallocates any internal resources acquired in Init for this BRC session. Invoked during MFXVideoENCODE_Close.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] pthis`: Pointer to the BRC object.

mfxStatus (***GetFrameCtrl**) (*mfxHDL* pthis, *mfxBRCFrameParam* *par, *mfxBRCFrameCtrl* *ctrl)
Returns controls (`ctrl`) to encode next frame based on info from input *mfxBRCFrameParam* structure (`par`) and internal BRC state. Invoked asynchronously before each frame encoding or recoding.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] pthis`: Pointer to the BRC object.
- `[in] par`: Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

- `[out] ctrl`: Pointer to the output *mfxBRCFrameCtrl* structure.

mfxStatus (***Update**) (*mfxHDL* pthis, *mfxBRCFrameParam* *par, *mfxBRCFrameCtrl* *ctrl, *mfxBRCFrameStatus* *status)

Updates internal BRC state and returns status to instruct encoder whether it should recode the previous frame, skip the previous frame, do padding, or proceed to next frame based on info from input *mfxBRCFrameParam* and *mfxBRCFrameCtrl* structures. Invoked asynchronously after each frame encoding or recoding.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- `[in] pthis`: Pointer to the BRC object.
- `[in] par`: Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.
- `[in] ctrl`: Pointer to the output *mfxBRCFrameCtrl* structure.
- `[in] status`: Pointer to the output *mfxBRCFrameStatus* structure.

mfxExtChromaLocInfo

struct mfxExtChromaLocInfo

Members of this structure define the location of chroma samples information.

See Annex E of the ISO*\/IEC* 14496-10 specification for the definition of these parameters.

Public Members

mfxExtBuffer **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CHROMA_LOC_INFO.

mfxU16 **ChromaLocInfoPresentFlag**

mfxU16 **ChromaSampleLocTypeTopField**

mfxU16 **ChromaSampleLocTypeBottomField**

mfxU16 **reserved**[9]

mfxExtCodingOption

struct mfxExtCodingOption

Specifies additional options for encoding.

The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION.

mfxU16 **RateDistortionOpt**
Set this flag if rate distortion optimization is needed. See the CodingOptionValue enumerator for values of this option.

mfxU16 **MECostType**
Motion estimation cost type. This value is reserved and must be zero.

mfxU16 **MESearchType**
Motion estimation search algorithm. This value is reserved and must be zero.

mfxI16Pair **MVSearchWindow**
Rectangular size of the search window for motion estimation. This parameter is reserved and must be (0, 0).

mfxU16 **FramePicture**
Set this flag to encode interlaced fields as interlaced frames. This flag does not affect progressive input frames. See the CodingOptionValue enumerator for values of this option.

mfxU16 **CAVLC**
If set, CAVLC is used; if unset, CABAC is used for encoding. See the CodingOptionValue enumerator for values of this option.

mfxU16 **RecoveryPointSEI**
Set this flag to insert the recovery point SEI message at the beginning of every intra refresh cycle. See the description of IntRefType in *mfxExtCodingOption2* structure for details on how to enable and configure intra refresh.

If intra refresh is not enabled then this flag is ignored.

See the CodingOptionValue enumerator for values of this option.

mfxU16 **ViewOutput**
Set this flag to instruct the MVC encoder to output each view in separate bitstream buffer. See the CodingOptionValue enumerator for values of this option and the Multi-View Video Coding section for more details about usage of this flag.

mfxU16 **NalHrdConformance**
If this option is turned ON, then AVC encoder produces an HRD conformant bitstream. If it is turned OFF, then the AVC encoder may (but not necessarily) violate HRD conformance. That is, this option can force the encoder to produce an HRD conformant stream, but cannot force it to produce a non-conformant stream.

See the CodingOptionValue enumerator for values of this option.

mfxU16 **SingleSeiNalUnit**
If set, encoder puts all SEI messages in the singe NAL unit. It includes messages provided by application and created by encoder. It is a three-states option. See CodingOptionValue enumerator for values of this option. The three states are:

- UNKNOWN Put each SEI in its own NAL unit.
- ON Put all SEI messages in the same NAL unit.
- OFF The same as unknown.

mfxU16 **VuiVclHrdParameters**
If set and VBR rate control method is used, then VCL HRD parameters are written in bitstream with values identical to the values of the NAL HRD parameters. See the CodingOptionValue enumerator for values of this option.

mfxU16 **RefPicListReordering**
Set this flag to activate reference picture list reordering. This value is reserved and must be zero.

mfxU16 **ResetRefList**
Set this flag to reset the reference list to non-IDR I-frames of a GOP sequence. See the CodingOptionValue enumerator for values of this option.

mfxU16 **RefPicMarkRep**
Set this flag to write the reference picture marking repetition SEI message into the output bitstream. See the CodingOptionValue enumerator for values of this option.

mfxU16 **FieldOutput**
Set this flag to instruct the AVC encoder to output bitstreams immediately after the encoder encodes a field, in the field-encoding mode. See the CodingOptionValue enumerator for values of this option.

mfxU16 **IntraPredBlockSize**
Minimum block size of intra-prediction. This value is reserved and must be zero.

mfxU16 **InterPredBlockSize**
Minimum block size of inter-prediction. This value is reserved and must be zero.

mfxU16 **MVPrecision**
Specify the motion estimation precision. This parameter is reserved and must be zero.

mfxU16 **MaxDecFrameBuffering**
Specifies the maximum number of frames buffered in a DPB. A value of zero means unspecified.

mfxU16 **AUDelimiter**
Set this flag to insert the Access Unit Delimiter NAL. See the CodingOptionValue enumerator for values of this option.

mfxU16 **PicTimingSEI**
Set this flag to insert the picture timing SEI with pic_struct syntax element. See sub-clauses D.1.2 and D.2.2 of the ISO/IEC 14496-10 specification for the definition of this syntax element. See the CodingOptionValue enumerator for values of this option. The default value is ON.

mfxU16 **VuiNalHrdParameters**
Set this flag to insert NAL HRD parameters in the VUI header. See the CodingOptionValue enumerator for values of this option.

mfxExtCodingOption2

struct mfxExtCodingOption2
Used with the *mfxExtCodingOption* structure to specify additional options for encoding.

The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization and to the *mfxEncodeCtrl* during runtime.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION2.

mfxU16 **IntRefType**
Specifies intra refresh type. See the IntraRefreshTypes. The major goal of intra refresh is improvement of error resilience without significant impact on encoded bitstream size caused by I-frames. The encoder achieves this by encoding part of each frame in the refresh cycle using intra MBs.

This parameter is valid during initialization and runtime. When used with temporal scalability, intra refresh applied only to base layer.

MFX_REFRESH_NO No refresh.

MFX_REFRESH_VERTICAL Vertical refresh, by column of MBs.

MFX_REFRESH_HORIZONTAL Horizontal refresh, by rows of MBs.

MFX_REFRESH_SLICE Horizontal refresh by slices without overlapping.

MFX_REFRESH_SLICE Library ignores IntRefCycleSize (size of refresh cycle equals number slices).

mfxU16 **IntRefCycleSize**
Specifies number of pictures within refresh cycle starting from 2. 0 and 1 are invalid values. This parameter is valid only during initialization.

mfxI16 **IntRefQPDelta**
Specifies QP difference for inserted intra MBs. Signed values are in the -51 to 51 range. This parameter is valid during initialization and runtime.

mfxU32 **MaxFrameSize**
Specify maximum encoded frame size in byte. This parameter is used in VBR based bitrate control modes and ignored in others. The encoder tries to keep frame size below specified limit but minor overshoots are possible to preserve visual quality. This parameter is valid during initialization and runtime. It is recommended to set MaxFrameSize to 5x-10x target frame size ((TargetKbps*1000)/(8* FrameRateExtN/FrameRateExtD)) for I-frames and 2x-4x target frame size for P- and B-frames.

mfxU32 **MaxSliceSize**
Specify maximum slice size in bytes. If this parameter is specified other controls over number of slices are ignored.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU16 **BitrateLimit**
Modifies bitrate to be in the range imposed by the encoder. The default value is ON, that is, bitrate is limited. Setting this flag to OFF may lead to violation of HRD conformance.Specifying bitrate below the encoder range might significantly affect quality.

If set to ON, this option takes effect in non CQP modes: if TargetKbps is not in the range imposed by the encoder, it will be changed to be in the range.

This parameter is valid only during initialization. Flag works with MFX_CODEC_AVC only, it is ignored with other codecs. See the CodingOptionValue enumerator for values of this option.

mfxU16 **MBBRC**
Setting this flag enables macroblock level bitrate control that generally improves subjective visual quality. Enabling this flag may have negative impact on performance and objective visual quality metric. See the CodingOptionValue enumerator for values of this option. The default value depends on target usage settings.

mfxU16 **ExtBRC**
Set this option to ON to enable external BRC. See the CodingOptionValue enumerator for values of this option. Use the Query API function to check if this feature is supported.

mfxU16 **LookAheadDepth**
Specifies the depth of the look ahead rate control algorithm. The depth value is the number of frames that the encoder analyzes before encoding. Values are in the 10 to 100 range, inclusive. To instruct the encoder to use the default value the application should zero this field.

mfxU16 **Trellis**
Used to control trellis quantization in AVC encoder. See TrellisControl enumerator for values of this option. This parameter is valid only during initialization.

mfxU16 **RepeatPPS**
Controls picture parameter set repetition in AVC encoder. Set this flag to ON to repeat PPS with each frame. See the CodingOptionValue enumerator for values of this option. The default value is ON. This parameter is valid only during initialization.

mfxU16 **BRefType**
Controls usage of B-frames as reference. See BRefControl enumerator for values of this option. This parameter is valid only during initialization.

mfxU16 **AdaptiveI**
Controls insertion of I-frames by the encoder. Set this flag to ON to allow changing of frame type from P and B to I. This option is ignored if GopOptFlag in *mfxInfoMFX* structure is equal to MFX_GOP_STRICT. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

mfxU16 **AdaptiveB**
Controls changing of frame type from B to P. Set this flag to ON enable changing of frame type from B to P. This option is ignored if GopOptFlag in *mfxInfoMFX* structure is equal to MFX_GOP_STRICT. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

mfxU16 **LookAheadDS**
Controls down sampling in look ahead bitrate control mode. See LookAheadDownSampling enumerator for values of this option. This parameter is valid only during initialization.

mfxU16 **NumMbPerSlice**
Specifies suggested slice size in number of macroblocks. The library can adjust this number based on platform capability. If this option is specified, that is, if it is not equal to zero, the library ignores *mfxInfoMFX::NumSlice* parameter.

mfxU16 **SkipFrame**
Enables usage of mfxEncodeCtrl::SkipFrameparameter. See the SkipFrame enumerator for values of this option.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU8 **MinQPI**
Minimum allowed QP value for I-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU8 **MaxQPI**
Maximum allowed QP value for I-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU8 **MinQPP**
Minimum allowed QP value for P-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU8 **MaxQPP**
Maximum allowed QP value for P-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU8 **MinQPB**
Minimum allowed QP value for B-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU8 **MaxQPB**
Maximum allowed QP value for B-frame types. Valid range is 1 to 51 inclusive. Zero means default value, that is, no limitations on QP.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU16 **FixedFrameRate**
Sets fixed_frame_rate_flag in VUI.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU16 **DisableDeblockingIdc**
Disables deblocking.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU16 **DisableVUI**
Completely disables VUI in the output bitstream.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU16 **BufferingPeriodSEI**
Controls insertion of buffering period SEI in the encoded bitstream. It should be one of the following values:

MFX_BPSEI_DEFAULT Encoder decides when to insert BP SEI,

MFX_BPSEI_IFRAME BP SEI should be inserted with every I-frame.

mfxU16 **EnableMAD**
Set this flag to ON to enable per-frame reporting of Mean Absolute Difference. This parameter is valid only during initialization.

mfxU16 **UseRawRef**
Set this flag to ON to use raw frames for reference instead of reconstructed frames. This parameter is valid during initialization and runtime (only if was turned ON during initialization).

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxExtCodingOption3

struct mfxExtCodingOption3
Used with *mfxExtCodingOption* and *mfxExtCodingOption2* structures to specify additional options for encoding. The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization and to the *mfxEncodeCtrl* during runtime.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION3.

mfxU16 **NumSliceI**
The number of slices for I-frames.

Note Not all codecs and implementations support these values. Use the Query API function to check if this feature is supported

mfxU16 **NumSliceP**
The number of slices for P-frames.

Note Not all codecs and implementations support these values. Use the Query API function to check if this feature is supported

mfxU16 **NumSliceB**
The number of slices for B-frames.

Note Not all codecs and implementations support these values. Use the Query API function to check if this feature is supported

mfxU16 **WinBRCMaxAvgKbps**
When rate control method is MFX_RATECONTROL_VBR, MFX_RATECONTROL_LA, MFX_RATECONTROL_LA_HRD, or MFX_RATECONTROL_QVBR this parameter specifies the maximum bitrate averaged over a sliding window specified by WinBRCSize. For MFX_RATECONTROL_CBR this parameter is ignored and equals TargetKbps.

mfxU16 **WinBRCSize**
When rate control method is MFX_RATECONTROL_CBR, MFX_RATECONTROL_VBR, MFX_RATECONTROL_LA, MFX_RATECONTROL_LA_HRD, or MFX_RATECONTROL_QVBR this parameter specifies sliding window size in frames. Set this parameter to zero to disable sliding window.

mfxU16 **QVBRQuality**
When rate control method is MFX_RATECONTROL_QVBR, this parameter specifies quality factor. Values are in the 1 to 51 range, where 1 corresponds to the best quality.

mfxU16 **EnableMBQP**
Set this flag to ON to enable per-macroblock QP control. Rate control method must be MFX_RATECONTROL_CQP. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

mfxU16 **IntRefCycleDist**
Distance between the beginnings of the intra-refresh cycles in frames. Zero means no distance between cycles.

mfxU16 **DirectBiasAdjustment**
Set this flag to ON to enable the ENC mode decision algorithm to bias to fewer B Direct/Skip types. Applies only to B-frames, all other frames will ignore this setting. See the CodingOptionValue enumerator for values of this option.

mfxU16 **GlobalMotionBiasAdjustment**
Enables global motion bias. See the CodingOptionValue enumerator for values of this option.

mfxU16 **MVCostScalingFactor**
Values are:

- 0: Set MV cost to be 0.
- 1: Scale MV cost to be 1/2 of the default value.
- 2: Scale MV cost to be 1/4 of the default value.
- 3: Scale MV cost to be 1/8 of the default value.

mfxU16 **MBDisableSkipMap**
Set this flag to ON to enable usage of *mfxExtMBDisableSkipMap*. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

mfxU16 **WeightedPred**
Weighted prediction mode. See the WeightedPred enumerator for values of these options.

mfxU16 **WeightedBiPred**
Weighted prediction mode. See the WeightedPred enumerator for values of these options.

mfxU16 **AspectRatioInfoPresent**
Instructs encoder whether aspect ratio info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

mfxU16 **OverscanInfoPresent**
Instructs encoder whether overscan info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

mfxU16 **OverscanAppropriate**
ON indicates that the cropped decoded pictures output are suitable for display using overscan. OFF indicates that the cropped decoded pictures output contain visually important information in the entire region out to the edges of the cropping rectangle of the picture. See the CodingOptionValue enumerator for values of this option.

mfxU16 **TimingInfoPresent**
Instructs encoder whether frame rate info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

mfxU16 **BitstreamRestriction**
Instructs encoder whether bitstream restriction info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

mfxU16 **LowDelayHrd**
Corresponds to AVC syntax element low_delay_hrd_flag (VUI). See the CodingOptionValue enumerator for values of this option.

mfxU16 **MotionVectorsOverPicBoundaries**
When set to OFF, no sample outside the picture boundaries and no sample at a fractional sample position for which the sample value is derived using one or more samples outside the picture boundaries is used for inter prediction of any sample.

When set to ON, one or more samples outside picture boundaries may be used in inter prediction.

See the CodingOptionValue enumerator for values of this option.

mfxU16 **Log2MaxMvLengthHorizontal**

mfxU16 **Log2MaxMvLengthVertical**

mfxU16 **ScenarioInfo**
Provides a hint to encoder about the scenario for the encoding session. See the ScenarioInfo enumerator for values of this option.

mfxU16 **ContentInfo**
Provides a hint to encoder about the content for the encoding session. See the ContentInfo enumerator for values of this option.

mfxU16 **PRefType**
When GopRefDist=1, specifies the model of reference list construction and DPB management. See the PRefType enumerator for values of this option.

mfxU16 **FadeDetection**
Instructs encoder whether internal fade detection algorithm should be used for calculation of weigh/offset values for pred_weight_table unless application provided *mfxExtPredWeightTable* for this frame. See the CodingOptionValue enumerator for values of this option.

mfxI16 **DeblockingAlphaTcOffset**

mfxI16 **DeblockingBetaOffset**

mfxU16 **GPB**
Set this flag to OFF to make HEVC encoder use regular P-frames instead of GPB. See the CodingOption-Value enumerator for values of this option.

mfxU32 **MaxFrameSizeI**
Same as *mfxExtCodingOption2::MaxFrameSize* but affects only I-frames. MaxFrameSizeI must be set if MaxFrameSizeP is set. If MaxFrameSizeI is not specified or greater than spec limitation, spec limitation will be applied to the sizes of I-frames.

mfxU32 **MaxFrameSizeP**
Same as *mfxExtCodingOption2::MaxFrameSize* but affects only P/B-frames. If MaxFrameSizeP equals 0, the library sets MaxFrameSizeP equal to MaxFrameSizeI. If MaxFrameSizeP is not specified or greater than spec limitation, spec limitation will be applied to the sizes of P/B-frames.

mfxU32 **reserved3**[3]

mfxU16 **EnableQPOffset**
Enables QPOffset control. See the CodingOptionValue enumerator for values of this option.

mfxI16 **QPOffset**[8]
Specifies QP offset per pyramid layer when EnableQPOffset is set to ON and RateControlMethod is CQP.

For B-pyramid, B-frame QP = QPB + QPOffset[layer].

For P-pyramid, P-frame QP = QPP + QPOffset[layer].

mfxU16 **NumRefActiveP**[8]
Max number of active references for P-frames. Array index is pyramid layer.

mfxU16 **NumRefActiveBL0**[8]
Max number of active references for B-frames in reference picture list 0. Array index is pyramid layer.

mfxU16 **NumRefActiveBL1**[8]
Max number of active references for B-frames in reference picture list 1. Array index is pyramid layer.

mfxU16 **ConstrainedIntraPredFlag**

mfxU16 **TransformSkip**
For HEVC if this option is turned ON, the transform_skip_enabled_flag will be set to 1 in PPS. OFF specifies that transform_skip_enabled_flag will be set to 0.

mfxU16 **TargetChromaFormatPlus1**
Minus 1 specifies target encoding chroma format (see ChromaFormatIdc enumerator). May differ from the source format. TargetChromaFormatPlus1 = 0 specifies the default target chroma format which is equal to source (mfxVideoParam::mfx::FrameInfo::ChromaFormat + 1), except RGB4 source format. In case of RGB4 source format default target , chroma format is 4:2:0 (instead of 4:4:4) for the purpose of backward compatibility.

mfxU16 **TargetBitDepthLuma**
Target encoding bit-depth for luma samples. May differ from source bit-depth. 0 specifies a default target bit-depth that is equal to source (mfxVideoParam::mfx::FrameInfo::BitDepthLuma).

mfxU16 **TargetBitDepthChroma**
Target encoding bit-depth for chroma samples. May differ from source bit-depth. 0 specifies a default target bit-depth that is equal to source (mfxVideoParam::mfx::FrameInfo::BitDepthChroma).

mfxU16 **BRCPanicMode**
Controls panic mode in AVC and MPEG2 encoders.

mfxU16 **LowDelayBRC**
When rate control method is MFX_RATECONTROL_VBR, MFX_RATECONTROL_QVBR or MFX_RATECONTROL_VCM this parameter specifies frame size tolerance. Set this parameter to MFX_CODINGOPTION_ON to allow strictly obey average frame size set by MaxKbps, for example cases when MaxFrameSize == (MaxKbps*1000)/(8* FrameRateExtN/FrameRateExtD). Also MaxFrameSizeI and MaxFrameSizeP can be set separately.

mfxU16 **EnableMBForceIntra**
Set this flag to ON to enable usage of *mfxExtMBForceIntra* for AVC encoder. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

mfxU16 **AdaptiveMaxFrameSize**
If this flag is set to ON, BRC may decide a larger P- or B-frame size than what MaxFrameSizeP dictates when the scene change is detected. It may benefit the video quality. AdaptiveMaxFrameSize feature is not supported with LowPower ON or if the value of MaxFrameSizeP = 0.

mfxU16 **RepartitionCheckEnable**
Controls AVC encoder attempts to predict from small partitions. Default value allows encoder to choose preferred mode. MFX_CODINGOPTION_ON forces encoder to favor quality and MFX_CODINGOPTION_OFF forces encoder to favor performance.

mfxU16 **QuantScaleType**

mfxU16 **IntraVLCFormat**

mfxU16 **ScanType**

mfxU16 **EncodedUnitsInfo**
Set this flag to ON to make encoded units info available in *mfxExtEncodedUnitsInfo*.

mfxU16 **EnableNalUnitType**
If this flag is set to ON, the HEVC encoder uses the NAL unit type provided by the application in the *mfxEncodeCtrl::MfxNalUnitType* field. This parameter is valid only during initialization.

Note Not all codecs and implementations support this value. Use the Query API function to check if this feature is supported.

mfxU16 **ExtBrcAdaptiveLTR**
Turn OFF to prevent Adaptive marking of Long Term Reference Frames when using ExtBRC. When set to ON and using ExtBRC, encoders will mark, modify, or remove LTR frames based on encoding parameters and content properties. The application must set each input frame's *mfxFrameData::FrameOrder* for correct operation of LTR.

mfxU16 **reserved**[163]

mfxExtCodingOptionSPSPPS

struct mfxExtCodingOptionSPSPPS
Attach this structure as part of the extended buffers to configure the encoder during MFXVideoENCODE_Init. The sequence or picture parameters specified by this structure overwrite any parameters specified by the structure or any other attached extended buffers attached.

For H.264, SPSBuffer and PPSBuffer must point to valid bitstreams that contain the sequence parameter set and picture parameter set, respectively.

For MPEG-2, SPSBuffer must point to valid bitstreams that contain the sequence header followed by any sequence header extension. The PPSBuffer pointer is ignored.

The encoder imports parameters from these buffers. If the encoder does not support the specified parameters, the encoder does not initialize and returns the status code MFX_ERR_INCOMPATIBLE_VIDEO_PARAM.

Check with the MFXVideoENCODE_Query function for the support of this multiple segment encoding feature. If this feature is not supported, the query returns MFX_ERR_UNSUPPORTED.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION_SPSPPS.

mfxU8 ***SPSBuffer**
Pointer to a valid bitstream that contains the SPS (sequence parameter set for H.264 or sequence header followed by any sequence header extension for MPEG-2) buffer. Can be NULL to skip specifying the SPS.

mfxU8 ***PPSBuffer**
Pointer to a valid bitstream that contains the PPS (picture parameter set for H.264 or picture header followed by any picture header extension for MPEG-2) buffer. Can be NULL to skip specifying the PPS.

mfxU16 **SPSBufSize**
Size of the SPS in bytes.

mfxU16 **PPSBufSize**
Size of the PPS in bytes.

mfxU16 **SPSId**
SPS identifier. The value is reserved and must be zero.

mfxU16 **PPSId**
PPS identifier. The value is reserved and must be zero.

mfxExtCodingOptionVPS

struct mfxExtCodingOptionVPS

Attach this structure as part of the extended buffers to configure the encoder during MFXVideoENCODE_Init. The sequence or picture parameters specified by this structure overwrite any parameters specified by the structure or any other attached extended buffers attached.

If the encoder does not support the specified parameters, the encoder does not initialize and returns the status code MFX_ERR_INCOMPATIBLE_VIDEO_PARAM.

Check with the MFXVideoENCODE_Query function for the support of this multiple segment encoding feature. If this feature is not supported, the query returns MFX_ERR_UNSUPPORTED.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CODING_OPTION_VPS.

mfxU8 ***VPSBuffer**
Pointer to a valid bitstream that contains the VPS (video parameter set for HEVC) buffer.

mfxU16 **VPSBufSize**
Size of the VPS in bytes.

mfxU16 **VPSId**
VPS identifier; the value is reserved and must be zero.

mfxExtContentLightLevelInfo

struct mfxExtContentLightLevelInfo

Configures the HDR SEI message.

If the application attaches this structure to the *mfxEncodeCtrl* structure at runtime, the encoder inserts the HDR SEI message for the current frame and ignores InsertPayloadToggle.

If the application attaches this structure to the *mfxVideoParam* structure during initialization or reset, the encoder inserts the HDR SEI message based on InsertPayloadToggle.

Field semantics are defined in ITU-T* H.265 Annex D.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to EXTBUFF_CONTENT_LIGHT_LEVEL_INFO.

mfxU16 **InsertPayloadToggle**
InsertHDRPayload enumerator value.

mfxU16 **MaxContentLightLevel**
Maximum luminance level of the content. Field range is 1 to 65535.

mfxU16 **MaxPicAverageLightLevel**
Maximum average per-frame luminance level of the content. Field range is 1 to 65535.

mfxExtDirtyRect

struct mfxExtDirtyRect

Used by the application to specify dirty regions within a frame during encoding. It may be used at initialization or at runtime.

Dirty rectangle definition is using end-point exclusive notation. In other words, the pixel with (Right, Bottom) coordinates lies immediately outside of the dirty rectangle. Left, Top, Right, Bottom should be aligned by codec-specific block boundaries (should be dividable by 16 for AVC, or by block size (8, 16, 32 or 64, depends on platform) for HEVC).

Every dirty rectangle with unaligned coordinates will be expanded to a minimal-area block-aligned dirty rectangle, enclosing the original one. For example, a (5, 5, 15, 31) dirty rectangle will be expanded to (0, 0, 16, 32) for AVC encoder, or to (0, 0, 32, 32) for HEVC, if block size is 32.

Dirty rectangle (0, 0, 0, 0) is a valid dirty rectangle and means that the frame is not changed.

Dirty rectangle coordinates

The following structure members are used by the Rect array contained in the parent structure.

mfxU32 **Left**
Dirty region left coordinate.

mfxU32 **Top**
Dirty region top coordinate.

mfxU32 **Right**
Dirty region right coordinate.

mfxU32 **Bottom**
Dirty region bottom coordinate.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DIRTY_RECTANGLES.

mfxU16 **NumRect**
Number of dirty rectangles.

struct *mfxExtDirtyRect*::**[anonymous]** **Rect**[256]
Array of dirty rectangles.

mfxExtEncodedUnitsInfo

struct mfxExtEncodedUnitsInfo

If *mfxExtCodingOption3::EncodedUnitsInfo* was set to MFX_CODINGOPTION_ON during encoder initialization, the *mfxExtEncodedUnitsInfo* structure is attached to the *mfxBitstream* structure during encoding. It is used to report information about coding units in the resulting bitstream.

The number of filled items in UnitInfo is min(NumUnitsEncoded, NumUnitsAlloc).

For counting a minimal amount of encoded units you can use the following algorithm:

```
nSEI = amountOfApplicationDefinedSEI;
if (CodingOption3.NumSlice[IPB] != 0 || mfxVideoParam.mfx.NumSlice != 0)
  ExpectedAmount = 10 + nSEI + Max(CodingOption3.NumSlice[IPB], mfxVideoParam.mfx.
↪NumSlice);
else if (CodingOption2.NumMBPerSlice != 0)
  ExpectedAmount = 10 + nSEI + (FrameWidth * FrameHeight) / (256 * CodingOption2.
↪NumMBPerSlice);
else if (CodingOption2.MaxSliceSize != 0)
  ExpectedAmount = 10 + nSEI + Round(MaxBitrate / (FrameRate*CodingOption2.
↪MaxSliceSize));
else
  ExpectedAmount = 10 + nSEI;

if (mfxFrameInfo.PictStruct != MFX_PICSTRUCT_PROGRESSIVE)
  ExpectedAmount = ExpectedAmount * 2;

if (temporalScaleabilityEnabled)
  ExpectedAmount = ExpectedAmount * 2;
```

Note Only supported by the AVC encoder.

Public Members

mfxExtBuffer **Header**
: Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODED_UNITS_INFO.

mfxEncodedUnitInfo ***UnitInfo**
: Pointer to an array of mfxEncodedUnitsInfo structures whose size is equal to or greater than NumUnitsAlloc.

mfxU16 **NumUnitsAlloc**
: UnitInfo array size.

mfxU16 **NumUnitsEncoded**
: Output field. Number of coding units to report. If NumUnitsEncoded is greater than NumUnitsAlloc, the UnitInfo array will contain information only for the first NumUnitsAlloc units. User may consider reallocating the UnitInfo array to avoid this for subsequent frames.

mfxExtEncoderCapability

struct mfxExtEncoderCapability
: Used to retrieve encoder capability. See the description of mode 4 of the MFXVideoENCODE_Query function for details on how to use this structure.

Note Not all implementations of the encoder support this extended buffer. The application must use query mode 1 to determine if the functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE then the functionality is supported.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODER_CAPABILITY.

mfxU32 **MBPerSec**
Specify the maximum processing rate in macro blocks per second.

mfxExtEncoderIPCMArea

struct mfxExtEncoderIPCMArea
Specifies rectangle areas for IPCM coding mode.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODER_IPCM_AREA.

struct *mfxExtEncoderIPCMArea*::*area* ***Areas**
Array of areas.

struct area
Number of areas

Public Members

mfxU32 **Left**
Left area coordinate.

mfxU32 **Top**
Top area coordinate.

mfxU32 **Right**
Right area coordinate.

mfxU32 **Bottom**
Bottom area coordinate.

mfxExtEncoderResetOption

struct mfxExtEncoderResetOption
Used to control the encoder behavior during reset. By using this structure, the application instructs the encoder to start a new coded sequence after reset or to continue encoding of the current sequence.

This structure is also used in mode 3 of the MFXVideoENCODE_Query function to check for reset outcome before actual reset. The application should set StartNewSequence to the required behavior and call the query function. If the query fails (see status codes below), then reset is not possible in current encoder state. If the application sets StartNewSequence to MFX_CODINGOPTION_UNKNOWN, then the query function replaces the coding option with the actual reset type: MFX_CODINGOPTION_ON if the encoder will begin a new sequence after reset or MFX_CODINGOPTION_OFF if the encoder will continue the current sequence.

Using this structure may cause one of the following status codes from the MFXVideoENCODE_Reset and MFXVideoENCODE_Queryfunctions:

- MFX_ERR_INVALID_VIDEO_PARAM If a reset is not possible. For example, the application sets StartNewSequence to off and requests resolution change.
- MFX_ERR_INCOMPATIBLE_VIDEO_PARAM If the application requests change that leads to memory allocation. For example, the application sets StartNewSequence to on and requests resolution change to greater than the initialization value.
- MFX_ERR_NONE If reset is possible.

The following limited list of parameters can be changed without starting a new coded sequence:

- The bitrate parameters, TargetKbps and MaxKbps, in the *mfxInfoMFX* structure.
- The number of slices, NumSlice, in the *mfxInfoMFX* structure. Number of slices should be equal to or less than the number of slices during initialization.
- The number of temporal layers in the *mfxExtAvcTemporalLayers* structure. Reset should be called immediately before encoding of frame from base layer and number of reference frames should be large enough for the new temporal layers structure.
- The quantization parameters, QPI, QPP and QPB, in the *mfxInfoMFX* structure.

The application should retrieve all cached frames before calling reset. When the Query API function checks for reset outcome, it expects that this requirement be satisfied. If it is not true and there are some cached frames inside the encoder, then the query result may differ from the reset result, because the encoder may insert an IDR frame to produce valid coded sequence.

See the *Configuration Change* section for more information.

See the *Streaming and Video Conferencing Features* section for more information.

Note Not all implementations of the encoder support this extended buffer. The application must use query mode 1 to determine if the functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE, then the functionality is supported.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODER_RESET_OPTION.

mfxU16 **StartNewSequence**
Instructs encoder to start new sequence after reset. Use one of the CodingOptionValue options:

- MFX_CODINGOPTION_ON The encoder completely reset internal state and begins new coded sequence after reset, including insertion of IDR frame, sequence, and picture headers.
- MFX_CODINGOPTION_OFF The encoder continues encoding of current coded sequence after reset, without insertion of IDR frame.
- MFX_CODINGOPTION_UNKNOWN Depending on the current encoder state and changes in configuration parameters, the encoder may or may not start new coded sequence. This value is also used to query reset outcome.

mfxExtEncoderROI

struct mfxExtEncoderROI

Used by the application to specify different Region Of Interests during encoding. It may be used at initialization or at runtime.

ROI location rectangle

The ROI rectangle definition uses end-point exclusive notation. In other words, the pixel with (Right, Bottom) coordinates lies immediately outside of the ROI. Left, Top, Right, Bottom should be aligned by codec-specific block boundaries (should be dividable by 16 for AVC, or by 32 for HEVC). Every ROI with unaligned coordinates will be expanded by the library to minimal-area block-aligned ROI, enclosing the original one. For example (5, 5, 15, 31) ROI will be expanded to (0, 0, 16, 32) for AVC encoder, or to (0, 0, 32, 32) for HEVC.

mfxU32 **Left**
Left ROI's coordinate.

mfxU32 **Top**
Top ROI's coordinate.

mfxU32 **Right**
Right ROI's coordinate.

mfxU32 **Bottom**
Bottom ROI's coordinate.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODER_ROI.

mfxU16 **NumROI**
Number of ROI descriptions in array. The Query API function mode 2 returns maximum supported value (set it to 256 and query will update it to maximum supported value).

mfxU16 **ROIMode**
QP adjustment mode for ROIs. Defines if Priority or DeltaQP is used during encoding.

mfxI16 **Priority**
Priority of ROI. Used if ROIMode = MFX_ROI_MODE_PRIORITY.This is an absolute value in the range of -3 to 3, which will be added to the MB QP. Priority is deprecated mode and is used only for backward compatibility. Bigger value produces better quality.

mfxI16 **DeltaQP**
Delta QP of ROI. Used if ROIMode = MFX_ROI_MODE_QP_DELTA. This is an absolute value in the range of -51 to 51, which will be added to the MB QP. Lesser value produces better quality.

struct *mfxExtEncoderROI*::**[anonymous] ROI**[256]
Array of ROIs. Different ROI may overlap each other. If macroblock belongs to several ROI, Priority from ROI with lowest index is used.

mfxExtHEVCRegion

struct mfxExtHEVCRegion
Attached to the *mfxVideoParam* structure during HEVC encoder initialization. Specifies the region to encode.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_HEVC_REGION.

mfxU32 **RegionId**
ID of region.

mfxU16 **RegionType**
Type of region. See HEVCRegionType enumerator for the list of types.

mfxU16 **RegionEncoding**
Set to MFX_HEVC_REGION_ENCODING_ON to encode only specified region.

mfxExtHEVCTiles

struct mfxExtHEVCTiles
Configures tiles options for the HEVC encoder. The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_HEVC_TILES.

mfxU16 **NumTileRows**
Number of tile rows.

mfxU16 **NumTileColumns**
Number of tile columns.

mfxExtInsertHeaders

struct mfxExtInsertHeaders
Runtime ctrl buffer for SPS/PPS insertion with current encoding frame.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_INSERT_HEADERS.

mfxU16 **SPS**
Tri-state option to insert SPS.

mfxU16 **PPS**
Tri-state option to insert PPS.

mfxU16 **reserved**[8]

mfxExtMasteringDisplayColourVolume

struct mfxExtMasteringDisplayColourVolume

Configures the HDR SEI message.

If the application attaches this structure to the *mfxEncodeCtrl* structure at runtime, the encoder inserts the HDR SEI message for the current frame and ignores InsertPayloadToggle.

If the application attaches this structure to the *mfxVideoParam* structure during initialization or reset, the encoder inserts the HDR SEI message based on InsertPayloadToggle.

Field semantics are defined in ITU-T* H.265 Annex D.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME.

mfxU16 **InsertPayloadToggle**
InsertHDRPayload enumerator value.

mfxU16 **DisplayPrimariesX**[3]
Color primaries for a video source in increments of 0.00002. Consist of RGB x coordinates and define how to convert colors from RGB color space to CIE XYZ color space. Fields range is 0 to 50000.

mfxU16 **DisplayPrimariesY**[3]
Color primaries for a video source in increments of 0.00002. Consists of RGB y coordinates and defines how to convert colors from RGB color space to CIE XYZ color space. Field range is 0 to 50000.

mfxU16 **WhitePointX**
White point X coordinate.

mfxU16 **WhitePointY**
White point Y coordinate.

mfxU32 **MaxDisplayMasteringLuminance**
Specify maximum luminance of the display on which the content was authored in units of 0.00001 candelas per square meter. Field range is 1 to 65535.

mfxU32 **MinDisplayMasteringLuminance**
Specify minimum luminance of the display on which the content was authored in units of 0.00001 candelas per square meter. Field range is 1 to 65535.

mfxExtMBDisableSkipMap

struct mfxExtMBDisableSkipMap

Specifies macroblock map for current frame which forces specified macroblocks to be non-skip if *mfxExtCodingOption3::MBDisableSkipMap* was turned ON during encoder initialization. The application can attach this extended buffer to the *mfxEncodeCtrl* structure during runtime.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MB_DISABLE_SKIP_MAP.

mfxU32 **MapSize**
Macroblock map size.

mfxU8 ***Map**
Pointer to a list of non-skip macroblock flags in raster scan order. Each flag is one byte in map. Set flag to 1 to force corresponding macroblock to be non-skip. In case of interlaced encoding, the first half of map affects the top field and the second half of map affects the bottom field.

mfxExtMBForceIntra

struct mfxExtMBForceIntra
Specifies macroblock map for current frame which forces specified macroblocks to be encoded as intra if *mfxExtCodingOption3::EnableMBForceIntra* was turned ON during encoder initialization. The application can attach this extended buffer to the *mfxEncodeCtrl* structure during runtime.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MB_FORCE_INTRA.

mfxU32 **MapSize**
Macroblock map size.

mfxU8 ***Map**
Pointer to a list of force intra macroblock flags in raster scan order. Each flag is one byte in map. Set flag to 1 to force corresponding macroblock to be encoded as intra. In case of interlaced encoding, the first half of map affects top field and the second half of map affects the bottom field.

mfxExtMBQP

struct mfxExtMBQP
Specifies per-macroblock QP for current frame if *mfxExtCodingOption3::EnableMBQP* was turned ON during encoder initialization. The application can attach this extended buffer to the *mfxEncodeCtrl* structure during runtime.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MBQP.

mfxU16 **Mode**
Defines QP update mode. See MBQPMode enumerator for more details.

mfxU16 **BlockSize**
QP block size, valid for HEVC only during Init and Runtime.

mfxU32 **NumQPAlloc**
Size of allocated by application QP or DeltaQP array.

mfxU8 ***QP**
Pointer to a list of per-macroblock QP in raster scan order. In case of interlaced encoding the first half of QP array affects the top field and the second half of QP array affects the bottom field. Valid when Mode = MFX_MBQP_MODE_QP_VALUE.

For AVC, the valid range is 1 to 51.

For HEVC, the valid range is 1 to 51. Application's provided QP values should be valid. Otherwise invalid QP values may cause undefined behavior. MBQP map should be aligned for 16x16 block size. The alignment rule is (width +15 /16) && (height +15 /16).

For MPEG2, QP corresponds to quantizer_scale of the ISO*\/IEC* 13818-2 specification and has a valid range of 1 to 112.

mfxI8 ***DeltaQP**
Pointer to a list of per-macroblock QP deltas in raster scan order. For block i: QP[i] = BrcQP[i] + DeltaQP[i]. Valid when Mode = MFX_MBQP_MODE_QP_DELTA.

mfxQPandMode ***QPmode**
Block-granularity modes when MFX_MBQP_MODE_QP_ADAPTIVE is set.

mfxExtMoveRect

struct mfxExtMoveRect
Used by the application to specify moving regions within a frame during encoding.

Destination rectangle location should be aligned to MB boundaries (should be dividable by 16). If not, the encoder truncates it to MB boundaries, for example, both 17 and 31 will be truncated to 16.

Destination and source rectangle location

The following structure members are used by the Rect array contained in the parent structure.

mfxU32 **DestLeft**
Destination rectangle location.

mfxU32 **DestTop**
Destination rectangle location.

mfxU32 **DestRight**
Destination rectangle location.

mfxU32 **DestBottom**
Destination rectangle location.

mfxU32 **SourceLeft**
Source rectangle location.

mfxU32 **SourceTop**
Source rectangle location.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MOVING_RECTANGLE.

mfxU16 **NumRect**
Number of moving rectangles.

struct *mfxExtMoveRect*::**[anonymous]** **Rect**[256]
Array of moving rectangles.

mfxExtMVOverPicBoundaries

struct mfxExtMVOverPicBoundaries
Instructs encoder to use or not use samples over specified picture border for inter prediction. Attached to the *mfxVideoParam* structure.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_MV_OVER_PIC_BOUNDARIES.

mfxU16 **StickTop**
When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

mfxU16 **StickBottom**
When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

mfxU16 **StickLeft**
When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

mfxU16 **StickRight**
When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

mfxExtPartialBitstreamParam

struct mfxExtPartialBitstreamParam
Used by an encoder to output parts of the bitstream as soon as they are ready. The application can attach this extended buffer to the *mfxVideoParam* structure at initialization. If this option is turned ON (Granularity != MFX_PARTIAL_BITSTREAM_NONE), then the encoder can output bitstream by part based on the required granularity.

This parameter is valid only during initialization and reset. Absence of this buffer means default or previously configured bitstream output behavior.

Note Not all codecs and implementations support this feature. Use the Query API function to check if this feature is supported.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_PARTIAL_BITSTREAM_PARAM.

mfxU32 **BlockSize**
Output block granularity for PartialBitstreamGranularity. Valid only for MFX_PARTIAL_BITSTREAM_BLOCK.

mfxU16 **Granularity**
Granularity of the partial bitstream: slice/block/any, all types of granularity state in PartialBitstreamOutput enum.

mfxExtPictureTimingSEI

struct mfxExtPictureTimingSEI
Configures the H.264 picture timing SEI message. The encoder ignores it if HRD information in the stream is absent and the PicTimingSEI option in the *mfxExtCodingOption* structure is turned off. See *mfxExtCodingOption* for details.

If the application attaches this structure to the *mfxVideoParam* structure during initialization, the encoder inserts the picture timing SEI message based on provided template in every access unit of coded bitstream.

If application attaches this structure to the *mfxEncodeCtrl* structure at runtime, the encoder inserts the picture timing SEI message based on provided template in access unit that represents current frame.

These parameters define the picture timing information. An invalid value of 0xFFFF indicates that application does not set the value and encoder must calculate it.

See Annex D of the ISO*\/IEC* 14496-10 specification for the definition of these parameters.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_PICTURE_TIMING_SEI.

mfxU32 **reserved**[14]

mfxU16 **ClockTimestampFlag**

mfxU16 **CtType**

mfxU16 **NuitFieldBasedFlag**

mfxU16 **CountingType**

mfxU16 **FullTimestampFlag**

mfxU16 **DiscontinuityFlag**

mfxU16 **CntDroppedFlag**

mfxU16 **NFrames**

mfxU16 **SecondsFlag**

mfxU16 **MinutesFlag**

mfxU16 **HoursFlag**

mfxU16 **SecondsValue**

mfxU16 **MinutesValue**

mfxU16 **HoursValue**

mfxU32 **TimeOffset**

struct *mfxExtPictureTimingSEI*::**[anonymous] TimeStamp**[3]

mfxExtPredWeightTable

struct mfxExtPredWeightTable

Specifies weighted prediction table for current frame when all of the following conditions are met:

- *mfxExtCodingOption3::WeightedPred* was set to explicit during encoder Init or Reset .
- The current frame is P-frame or *mfxExtCodingOption3::WeightedBiPred* was set to explicit during encoder Init or Reset.
- The current frame is B-frame and is attached to the *mfxEncodeCtrl* structure.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_PRED_WEIGHT_TABLE.

mfxU16 **LumaLog2WeightDenom**
Base 2 logarithm of the denominator for all luma weighting factors. Value must be in the range of 0 to 7, inclusive.

mfxU16 **ChromaLog2WeightDenom**
Base 2 logarithm of the denominator for all chroma weighting factors. Value must be in the range of 0 to 7, inclusive.

mfxU16 **LumaWeightFlag**[2][32]
LumaWeightFlag[L][R] equal to 1 specifies that the weighting factors for the luma component are specified for R's entry of RefPicList L.

mfxU16 **ChromaWeightFlag**[2][32]
ChromaWeightFlag[L][R] equal to 1 specifies that the weighting factors for the chroma component are specified for R's entry of RefPicList L.

mfxI16 **Weights**[2][32][3][2]
The values of the weights and offsets used in the encoding processing. The value of Weights[i][j][k][m] is interpreted as: i refers to reference picture list 0 or 1; j refers to reference list entry 0-31; k refers to data for the luma component when it is 0, the Cb chroma component when it is 1 and the Cr chroma component when it is 2; m refers to weight when it is 0 and offset when it is 1

mfxExtVP8CodingOption

struct mfxExtVP8CodingOption
Describes VP8 coding options.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VP8_CODING_OPTION.

mfxU16 **Version**
Determines the bitstream version. Corresponds to the same VP8 syntax element in frame_tag.

mfxU16 **EnableMultipleSegments**
Set this option to ON to enable segmentation. This is tri-state option. See the CodingOptionValue enumerator for values of this option.

mfxU16 **LoopFilterType**
Select the type of filter (normal or simple). Corresponds to VP8 syntax element filter_type.

mfxU16 **LoopFilterLevel**[4]
Controls the filter strength. Corresponds to VP8 syntax element loop_filter_level.

mfxU16 **SharpnessLevel**
Controls the filter sensitivity. Corresponds to VP8 syntax element sharpness_level.

mfxU16 **NumTokenPartitions**
Specifies number of token partitions in the coded frame.

mfxI16 **LoopFilterRefTypeDelta**[4]
Loop filter level delta for reference type (intra, last, golden, altref).

mfxI16 **LoopFilterMbModeDelta**[4]
Loop filter level delta for MB modes.

mfxI16 **SegmentQPDelta**[4]
QP delta for segment.

mfxI16 **CoeffTypeQPDelta**[5]
QP delta for coefficient type (YDC, Y2AC, Y2DC, UVAC, UVDC).

mfxU16 **WriteIVFHeaders**
Set this option to ON to enable insertion of IVF container headers into bitstream. This is tri-state option. See the CodingOptionValue enumerator for values of this option

mfxU32 **NumFramesForIVFHeader**
Specifies number of frames for IVF header when WriteIVFHeaders is ON.

mfxExtVP9Segmentation

struct mfxExtVP9Segmentation
In the VP9 encoder it is possible to divide a frame into up to 8 segments and apply particular features (like delta for quantization index or for loop filter level) on a per-segment basis. “Uncompressed header” of every frame indicates if segmentation is enabled for the current frame, and (if segmentation enabled) contains full information about features applied to every segment. Every “Mode info block” of a coded frame has segment_id in the range of 0 to 7.

To enable Segmentation, the *mfxExtVP9Segmentation* structure with correct settings should be passed to the encoder. It can be attached to the *mfxVideoParam* structure during initialization or the MFXVideoENCODE_Reset call (static configuration). If the *mfxExtVP9Segmentation* buffer isn't attached during initialization, segmentation is disabled for static configuration. If the buffer isn't attached for the Reset call, the encoder continues to use static configuration for segmentation which was the default before this Reset call. If the *mfxExtVP9Segmentation* buffer with NumSegments=0 is provided during initialization or Reset call, segmentation becomes disabled for static configuration.

The buffer can be attached to the *mfxEncodeCtrl* structure during runtime (dynamic configuration). Dynamic configuration is applied to the current frame only. After encoding of the current frame, the encoder will switch to the next dynamic configuration or to static configuration if dynamic configuration is not provided for next frame).

The SegmentIdBlockSize, NumSegmentIdAlloc, and SegmentId parameters represent a segmentation map. Here, the segmentation map is an array of segment_ids (one byte per segment_id) for blocks of size NxN in raster scan order. The size NxN is specified by the application and is constant for the whole frame. If *mfxExtVP9Segmentation* is attached during initialization and/or during runtime, all three parameters should be set to proper values that do not conflict with each other and with NumSegments. If any of the parameters are not set or any conflict or error in these parameters is detected by the library, the segmentation map will be discarded.

Public Members

mfxExtBuffer **Header**
: Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VP9_SEGMENTATION.

mfxU16 **NumSegments**
: Number of segments for frame. Value 0 means that segmentation is disabled. Sending 0 for a particular frame will disable segmentation for this frame only. Sending 0 to the Reset API function will disable segmentation permanently. Segmentation can be enabled again by a subsequent Reset call.

mfxVP9SegmentParam **Segment**[8]
: Array of *mfxVP9SegmentParam* structures containing features and parameters for every segment. Entries with indexes bigger than NumSegments-1 are ignored. See the *mfxVP9SegmentParam* structure for definitions of segment features and their parameters.

mfxU16 **SegmentIdBlockSize**
: Size of block (NxN) for segmentation map. See SegmentIdBlockSize enumerator for values for this option. An encoded block that is bigger than SegmentIdBlockSize uses segment_id taken from it's top-left sub-block from the segmentation map. The application can check if a particular block size is supported by calling Query.

mfxU32 **NumSegmentIdAlloc**
: Size of buffer allocated for segmentation map (in bytes). Application must assure that NumSegmentIdAlloc is large enough to cover frame resolution with blocks of size SegmentIdBlockSize. Otherwise the segmentation map will be discarded.

mfxU8 ***SegmentId**
: Pointer to the segmentation map buffer which holds the array of segment_ids in raster scan order. The application is responsible for allocation and release of this memory. The buffer pointed to by SegmentId, provided during initialization or Reset call should be considered in use until another SegmentId is provided via Reset call (if any), or until MFXVideoENCODE_Close is called. The buffer pointed to by SegmentId provided with *mfxEncodeCtrl* should be considered in use while the input surface is locked by the library. Every segment_id in the map should be in the range of 0 to NumSegments-1. If some segment_id is out of valid range, the segmentation map cannot be applied. If the *mfxExtVP9Segmentation* buffer is attached to the *mfxEncodeCtrl* structure in runtime, SegmentId can be zero. In this case, the segmentation map from static configuration will be used.

mfxExtVP9TemporalLayers

struct mfxExtVP9TemporalLayers

API allows the encoding of VP9 bitstreams that contain several subset bitstreams that differ in frame rates, also called "temporal layers".

When decoding, each temporal layer can be extracted from the coded stream and decoded separately. The *mfxExtVP9TemporalLayers* structure configures the temporal layers for the VP9 encoder. It can be attached to the *mfxVideoParam* structure during initialization or the MFXVideoENCODE_Reset call. If the *mfxExtVP9TemporalLayers* buffer isn't attached during initialization, temporal scalability is disabled. If the buffer isn't attached for the Reset call, the encoder continues to use the temporal scalability configuration that was defined before the Reset call.

In the API, temporal layers are ordered by their frame rates in ascending order. Temporal layer 0 (having the lowest frame rate) is called the base layer. Each subsequent temporal layer includes all previous layers.

The temporal scalability feature requires a minimum number of allocated reference frames (controlled by the NumRefFrame parameter). If the NumRefFrame value set by the application isn't enough to build the reference structure for the requested number of temporal layers, the library corrects the NumRefFrame value. The temporal layer structure is reset (re-started) after key-frames.

Public Members

mfxExtBuffer **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VP9_TEMPORAL_LAYERS.

mfxVP9TemporalLayer **Layer**[8]

The array of temporal layers. Layer[0] specifies the base layer.

The library reads layers from the array when they are defined (FrameRateScale > 0). All layers starting from first layer with FrameRateScale = 0 are ignored. The last layer that is not ignored is considered the "highest layer".

The frame rate of the highest layer is specified in the *mfxVideoParam* structure. Frame rates of lower layers are calculated using their FrameRateScale.

TargetKbps of the highest layer should be equal to the TargetKbps value specified in the *mfxVideoParam* structure. If it is not true, TargetKbps of highest temporal layers has priority.

If there are no defined layers in the Layer array, the temporal scalability feature is disabled. For example, to disable temporal scalability in runtime, the application should pass *mfxExtVP9TemporalLayers* buffer to Reset with all FrameRateScales set to 0.

mfxQPandMode

struct mfxQPandMode

Specifies per-MB or per-CU mode and QP or deltaQP value depending on the mode type.

Public Members

mfxU8 **QP**
QP for MB or CU. Valid when Mode = MFX_MBQP_MODE_QP_VALUE.

For AVC, the valid range is 1 to 51.

For HEVC, the valid range is 1 to 51. The application's provided QP values should be valid, otherwise invalid QP values may cause undefined behavior.

MBQP map should be aligned for 16x16 block size. The align rule is: (width +15 /16) && (height +15 /16).

For MPEG2, the valid range is 1 to 112. QP corresponds to quantizer_scale of the ISO*\/IEC* 13818-2 specification.

mfxI8 **DeltaQP**
Per-macroblock QP delta. Valid when Mode = MFX_MBQP_MODE_QP_DELTA.

mfxU16 **Mode**
Defines QP update mode. Can be equal to MFX_MBQP_MODE_QP_VALUE or MFX_MBQP_MODE_QP_DELTA.

mfxVP9TemporalLayer

struct mfxVP9TemporalLayer
Specifies temporal layer.

Public Members

mfxU16 **FrameRateScale**
The ratio between the frame rates of the current temporal layer and the base layer. The library treats a particular temporal layer as "defined" if it has FrameRateScale > 0. If the base layer is defined, it must have FrameRateScale = 1. FrameRateScale of each subsequent layer (if defined) must be a multiple of and greater than the FrameRateScale value of previous layer.

mfxU16 **TargetKbps**
Target bitrate for the current temporal layer. Ignored if RateControlMethod is CQP. If RateControlMethod is not CQP, the application must provide TargetKbps for every defined temporal layer. TargetKbps of each subsequent layer (if defined) must be greater than the TargetKbps value of the previous layer.

VPP Structures

Structures used by VPP only.

API

mfxExtColorConversion

struct mfxExtColorConversion

A hint structure that tunes the VPP Color Conversion algorithm when attached to the *mfxVideoParam* structure during VPP Init.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_COLOR_CONVERSION.

mfxU16 **ChromaSiting**
See ChromaSiting enumerator for details.

ChromaSiting is applied on input or output surface depending on the scenario:

VPP Input	VPP Output	ChromaSiting Indicates
MFX_CHROMAFORMAT_YUV420 MFX_CHROMAFORMAT_YUV422	MFX_CHROMAFORMAT_YUV444	Chroma location for input
MFX_CHROMAFORMAT_YUV444	MFX_CHROMAFORMAT_YUV420 MFX_CHROMAFORMAT_YUV422	Chroma location for output
MFX_CHROMAFORMAT_YUV420	MFX_CHROMAFORMAT_YUV420	Chroma location for input and output
MFX_CHROMAFORMAT_YUV420	MFX_CHROMAFORMAT_YUV422	Horizontal location for input and output, vertical location for input

mfxExtDecVideoProcessing

struct mfxExtDecVideoProcessing
If attached to the *mfxVideoParam* structure during the Init stage, this buffer will instruct the decoder to resize output frames via the fixed function resize engine (if supported by hardware), utilizing direct pipe connection and bypassing intermediate memory operations. The main benefits of this mode of pipeline operation are offloading resize operation to a dedicated engine, thus reducing power consumption and memory traffic.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_DEC_VIDEO_PROCESSING.

struct *mfxExtDecVideoProcessing*::*mfxIn* **In**
Input surface description.

struct *mfxExtDecVideoProcessing*::*mfxOut* **Out**
Output surface description.

struct mfxIn
Input surface description.

Public Members

mfxU16 **CropX**
X coordinate of region of interest of the input surface.

mfxU16 **CropY**
Y coordinate of region of interest of the input surface.

mfxU16 **CropW**
Width coordinate of region of interest of the input surface.

mfxU16 **CropH**
Height coordinate of region of interest of the input surface.

struct mfxOut
Output surface description.

Public Members

mfxU32 **FourCC**
FourCC of output surface Note: Should be MFX_FOURCC_NV12.

mfxU16 **ChromaFormat**
Chroma Format of output surface.
Note Should be MFX_CHROMAFORMAT_YUV420

mfxU16 **Width**
Width of output surface.

mfxU16 **Height**
Height of output surface.

mfxU16 **CropX**
X coordinate of region of interest of the output surface.

mfxU16 **CropY**
Y coordinate of region of interest of the output surface.

mfxU16 **CropW**
Width coordinate of region of interest of the output surface.

mfxU16 **CropH**
Height coordinate of region of interest of the output surface.

mfxExtEncodedSlicesInfo

struct mfxExtEncodedSlicesInfo
Used by the encoder to report additional information about encoded slices. The application can attach this buffer to the *mfxBitstream* structure before calling the MFXVideoENCODE_EncodeFrameAsync function.

Note Not all implementations of the encoder support this extended buffer. The application must use query mode 1 to determine if the functionality is supported. To do this, the application must attach this extended buffer to the *mfxVideoParam* structure and call the MFXVideoENCODE_Query function. If the function returns MFX_ERR_NONE, then the functionality is supported.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCODED_SLICES_INFO.

mfxU16 **SliceSizeOverflow**
When *mfxExtCodingOption2::MaxSliceSize* is used, indicates the requested slice size was not met for one or more generated slices.

mfxU16 **NumSliceNonCopliant**
When *mfxExtCodingOption2::MaxSliceSize* is used, indicates the number of generated slices exceeds specification limits.

mfxU16 **NumEncodedSlice**
Number of encoded slices.

mfxU16 **NumSliceSizeAlloc**
SliceSize array allocation size. Must be specified by application.

mfxU16 ***SliceSize**
Slice size in bytes. Array must be allocated by application.

mfxExtVppAuxData

struct mfxExtVppAuxData
Returns auxiliary data generated by the video processing pipeline. The encoding process may use the auxiliary data by attaching this structure to the *mfxEncodeCtrl* structure.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_AUXDATA.

mfxU16 **PicStruct**
Detected picture structure - top field first, bottom field first, progressive or unknown if video processor cannot detect picture structure. See the PicStruct enumerator for definition of these values.

mfxExtVPPColorFill

struct mfxExtVPPColorFill
Configures the VPP ColorFill filter algorithm.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_COLORFILL.

mfxU16 **Enable**
Set to ON makes VPP fill the area between Width/Height and Crop borders. See the CodingOptionValue enumerator for values of this option.

mfxExtVPPComposite

struct mfxExtVPPComposite

Used to control composition of several input surfaces in one output. In this mode, the VPP skips any other filters. The VPP returns an error if any mandatory filter is specified and returns the filter skipped warning if an optional filter is specified. The only supported filters are deinterlacing and interlaced scaling. The only supported combinations of input and output color formats are:

- RGB to RGB,
- NV12 to NV12,
- RGB and NV12 to NV12, for per the pixel alpha blending use case.

The VPP returns MFX_ERR_MORE_DATA for additional input until an output is ready. When the output is ready, the VPP returns MFX_ERR_NONE. The application must process the output frame after synchronization.

The composition process is controlled by:

- mfxFrameInfo::CropXYWH in the input surface defines the location of the picture in the input frame.
- InputStream[i].DstXYWH defines the location of the cropped input picture in the output frame.
- mfxFrameInfo::CropXYWH in the output surface defines the actual part of the output frame. All pixels in the output frame outside this region will be filled by the specified color.

If the application uses the composition process on video streams with different frame sizes, the application should provide maximum frame size in the *mfxVideoParam* structure during the initialization, reset, or query operations.

If the application uses the composition process, the MFXVideoVPP_QueryIOSurf function returns the cumulative number of input surfaces, that is, the number required to process all input video streams. The function sets the frame size in the *mfxFrameAllocRequest* equal to the size provided by the application in the *mfxVideoParam* structure.

The composition process supports all types of surfaces.

All input surfaces should have the same type and color format, except for the per pixel alpha blending case, where it is allowable to mix NV12 and RGB surfaces.

There are three different blending use cases:

- **Luma keying.** All input surfaces should have the NV12 color format specified during VPP initialization. Part of each surface, including the first one, may be rendered transparent by using LumaKeyEnable, LumaKeyMin, and LumaKeyMax values.
- **Global alpha blending.** All input surfaces should have the same color format, NV12 or RGB, specified during VPP initialization. Each input surface, including the first one, can be blended with underlying surfaces by using GlobalAlphaEnable and GlobalAlpha values.
- **Per-pixel alpha blending.** It is allowed to mix NV12 and RGB input surfaces. Each RGB input surface, including the first one, can be blended with underlying surfaces by using PixelAlphaEnable value.

It is not allowed to mix different blending use cases in the same function call.

In the special case where the destination region of the output surface defined by output crops is fully covered with destination sub-regions of the surfaces, the fast compositing mode can be enabled. The main use case for this mode is a video-wall scenario with a fixed destination surface partition into sub-regions of potentially different size.

In order to trigger this mode, the application must cluster input surfaces into tiles, defining at least one tile by setting the NumTiles field to be greater than 0, and assigning surfaces to the corresponding tiles by setting the TileId field to the value within the 0 to NumTiles range per input surface. Tiles should also satisfy the following additional constraints:

- Each tile should not have more than 8 surfaces assigned to it.
- Tile bounding boxes, as defined by the enclosing rectangles of a union of a surfaces assigned to this tile, should not intersect.

Background color may be changed dynamically through Reset. There is no default value. YUV black is (0;128;128) or (16;128;128) depending on the sample range. The library uses a YUV or RGB triple depending on output color format.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_COMPOSITE.

mfxU16 **Y**
Y value of the background color.

mfxU16 **R**
R value of the background color.

mfxU16 **U**
U value of the background color.

mfxU16 **G**
G value of the background color.

mfxU16 **V**
V value of the background color.

mfxU16 **B**
B value of the background color.

mfxU16 **NumTiles**
Number of input surface clusters grouped together to enable fast compositing. May be changed dynamically at runtime through Reset.

mfxU16 **NumInputStream**
Number of input surfaces to compose one output. May be changed dynamically at runtime through Reset. Number of surfaces can be decreased or increased, but should not exceed the number specified during initialization. Query mode 2 should be used to find the maximum supported number.

mfxVPPCompInputStream ***InputStream**
An array of *mfxVPPCompInputStream* structures that describe composition of input video streams. It should consist of exactly NumInputStream elements.

mfxExtVPPDeinterlacing

struct mfxExtVPPDeinterlacing
Used by the application to specify different deinterlacing algorithms.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DEINTERLACING.

mfxU16 **Mode**
Deinterlacing algorithm. See the DeinterlacingMode enumerator for details.

mfxU16 **TelecinePattern**
Specifies telecine pattern when Mode = MFX_DEINTERLACING_FIXED_TELECINE_PATTERN. See the TelecinePattern enumerator for details.

mfxU16 **TelecineLocation**
Specifies position inside a sequence of 5 frames where the artifacts start when TelecinePattern = MFX_TELECINE_POSITION_PROVIDED

mfxU16 **reserved**[9]
Reserved for future use.

mfxExtVPPDenoise

struct mfxExtVPPDenoise
A hint structure that configures the VPP denoise filter algorithm.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DENOISE.

mfxU16 **DenoiseFactor**
Indicates the level of noise to remove. Value range of 0 to 100 (inclusive).

mfxExtVPPDetail

struct mfxExtVPPDetail
A hint structure that configures the VPP detail/edge enhancement filter algorithm.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DETAIL.

mfxU16 **DetailFactor**
Indicates the level of details to be enhanced. Value range of 0 to 100 (inclusive).

mfxExtVPPDoNotUse

struct mfxExtVPPDoNotUse

Tells the VPP not to use certain filters in pipeline. See "Configurable VPP filters" table for complete list of configurable filters. The user can attach this structure to the *mfxVideoParam* structure when initializing video processing.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DONOTUSE.

mfxU32 **NumAlg**
Number of filters (algorithms) not to use

mfxU32 ***AlgList**
Pointer to a list of filters (algorithms) not to use

mfxExtVPPDoUse

struct mfxExtVPPDoUse

Tells the VPP to include certain filters in the pipeline.

Each filter may be included in the pipeline in one of two different ways:

- Adding a filter ID to this structure. In this method, the default filter parameters are used.
- Attaching a filter configuration structure directly to the *mfxVideoParam* structure. In this method, adding filter ID to the *mfxExtVPPDoUse* structure is optional.

See Table "Configurable VPP filters" for complete list of configurable filters, their IDs, and configuration structures.

The user can attach this structure to the *mfxVideoParam* structure when initializing video processing.

Note MFX_EXTBUFF_VPP_COMPOSITE cannot be enabled using *mfxExtVPPDoUse* because default parameters are undefined for this filter. The application must attach the appropriate filter configuration structure directly to the *mfxVideoParam* structure to enable it.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_DOUSE.

mfxU32 **NumAlg**
Number of filters (algorithms) to use

mfxU32 ***AlgList**
Pointer to a list of filters (algorithms) to use

mfxExtVPPFieldProcessing

struct mfxExtVPPFieldProcessing

Configures the VPP field processing algorithm. The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization and/or to the *mfxFrameData* during runtime. Runtime configuration has priority over initialization configuration. If the field processing algorithm was activated via the *mfxExtVPPDoUse* structure and the *mfxExtVPPFieldProcessing* extended buffer was not provided during initialization, this buffer must be attached to the *mfxFrameData* structure of each input surface.

Public Members

mfxExtBuffer **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_FIELD_PROCESSING.

mfxU16 **Mode**

Specifies the mode of the field processing algorithm. See the VPPFieldProcessingMode enumerator for values of this option.

mfxU16 **InField**

When Mode is MFX_VPP_COPY_FIELD, specifies input field. See the PicType enumerator for values of this parameter.

mfxU16 **OutField**

When Mode is MFX_VPP_COPY_FIELD, specifies output field. See the PicType enumerator for values of this parameter.

mfxExtVPPFrameRateConversion

struct mfxExtVPPFrameRateConversion

Configures the VPP frame rate conversion filter. The user can attach this structure to the *mfxVideoParam* structure when initializing, resetting, or querying capability of video processing.

On some platforms the advanced frame rate conversion algorithm (the algorithm based on frame interpolation) is not supported. To query its support, the application should add the MFX_FRCALGM_FRAME_INTERPOLATION flag to the Algorithm value in the *mfxExtVPPFrameRateConversion* structure, attach it to the structure, and call the MFXVideoVPP_Query function. If the filter is supported, the function returns a MFX_ERR_NONE status and copies the content of the input structure to the output structure. If an advanced filter is not supported, then a simple filter will be used and the function returns MFX_WRN_INCOMPATIBLE_VIDEO_PARAM, copies content of the input structure to the output structure, and corrects the Algorithm value.

If advanced FRC algorithm is not supported, both MFXVideoVPP_Init and MFXVideoVPP_Reset functions return the MFX_WRN_INCOMPATIBLE_VIDEO_PARAM status.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION.

mfxU16 **Algorithm**
See the FrcAlgm enumerator for a list of frame rate conversion algorithms.

mfxExtVPPImageStab

struct mfxExtVPPImageStab
A hint structure that configures the VPP image stabilization filter.

On some platforms this filter is not supported. To query its support, the application should use the same approach that it uses to configure VPP filters: adding the filter ID to the *mfxExtVPPDoUse* structure or by attaching the *mfxExtVPPImageStab* structure directly to the *mfxVideoParam* structure and calling the MFXVideoVPP_Query function.

If this filter is supported, the function returns a MFX_ERR_NONE status and copies the content of the input structure to the output structure. If the filter is not supported, the function returns MFX_WRN_FILTER_SKIPPED, removes the filter from the *mfxExtVPPDoUse* structure, and zeroes the *mfxExtVPPImageStab* structure.

If the image stabilization filter is not supported, both MFXVideoVPP_Init and MFXVideoVPP_Reset functions return a MFX_WRN_FILTER_SKIPPED status.

The application can retrieve the list of active filters by attaching the *mfxExtVPPDoUse* structure to the *mfxVideoParam* structure and calling the MFXVideoVPP_GetVideoParam function. The application must allocate enough memory for the filter list.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_IMAGE_STABILIZATION.

mfxU16 **Mode**
Image stabilization mode. See ImageStabMode enumerator for values.

mfxExtVppMctf

struct mfxExtVppMctf
Provides setup for the Motion-Compensated Temporal Filter (MCTF) during the VPP initialization and for control parameters at runtime. By default, MCTF is off. An application may enable it by adding MFX_EXTBUFF_VPP_MCTF to the *mfxExtVPPDoUse* buffer or by attaching *mfxExtVppMctf* to the *mfxVideoParam* structure during initialization or reset.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_MCTF.

mfxU16 **FilterStrength**
Value in range of 0 to 20 (inclusive) to indicate the filter strength of MCTF.

The strength of the MCTF process controls the degree of possible change of pixel values eligible for MCTF - the greater the strength value, the larger the change. It is a dimensionless quantity - values in the range of 1 to 20 inclusively imply strength; value 0 stands for AUTO mode and is valid during initialization or reset only

If an invalid value is given, it is fixed to the default value of 0. If the field value is in the range of 1 to 20 inclusive, MCTF operates in fixed-strength mode with the given strength of MCTF process.

At runtime, values of 0 and greater than 20 are ignored.

mfxExtVPPMirroring

struct mfxExtVPPMirroring
Configures the VPP Mirroring filter algorithm.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_MIRRORING.

mfxU16 **Type**
Mirroring type. See MirroringType for values.

mfxExtVPPProcAmp

struct mfxExtVPPProcAmp
A hint structure that configures the VPP ProcAmp filter algorithm. The structure parameters will be clipped to their corresponding range and rounded by their corresponding increment.

Note There are no default values for fields in this structure, all settings must be explicitly specified every time this buffer is submitted for processing.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to FX_EXTBUFF_VPP_PROCAMP.

mfxF64 **Brightness**
The brightness parameter is in the range of -100.0F to 100.0F, in increments of 0.1F. Setting this field to 0.0F will disable brightness adjustment.

mfxF64 **Contrast**
The contrast parameter in the range of 0.0F to 10.0F, in increments of 0.01F, is used for manual contrast adjustment. Setting this field to 1.0F will disable contrast adjustment. If the parameter is negative, contrast will be adjusted automatically.

mfxF64 **Hue**
The hue parameter is in the range of -180F to 180F, in increments of 0.1F. Setting this field to 0.0F will disable hue adjustment.

mfxF64 **Saturation**
The saturation parameter is in the range of 0.0F to 10.0F, in increments of 0.01F. Setting this field to 1.0F will disable saturation adjustment.

mfxExtVPPRotation

struct mfxExtVPPRotation
Configures the VPP Rotation filter algorithm.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_ROTATION.

mfxU16 **Angle**
Rotation angle. See Angle enumerator for supported values.

mfxExtVPPScaling

struct mfxExtVPPScaling
Configures the VPP Scaling filter algorithm. Not all combinations of ScalingMode and InterpolationMethod are supported in the library. The application must use the Query API function to determine if a combination is supported.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_SCALING.

mfxU16 **ScalingMode**
Scaling mode. See ScalingMode for values.

mfxU16 **InterpolationMethod**
Interpolation mode for scaling algorithm. See InterpolationMode for values.

mfxExtVPPVideoSignalInfo

struct mfxExtVPPVideoSignalInfo
Used to control transfer matrix and nominal range of YUV frames. The application should provide this during initialization. Supported for multiple conversions, for example YUV to YUV, YUV to RGB, and RGB to YUV.

Note This structure is used by VPP only and is not compatible with *mfxExtVideoSignalInfo*.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_VPP_VIDEO_SIGNAL_INFO.

mfxU16 **TransferMatrix**
Transfer matrix.

mfxU16 **NominalRange**
Nominal range.

mfxInfoVPP

struct mfxInfoVPP
Specifies configurations for video processing. A zero value in any of the fields indicates that the corresponding field is not explicitly specified.

Public Members

mfxFrameInfo **In**
Input format for video processing.

mfxFrameInfo **Out**
Output format for video processing.

mfxVPPCompInputStream

struct mfxVPPCompInputStream
Used to specify input stream details for composition of several input surfaces in the one output.

Public Members

mfxU32 **DstX**
X coordinate of location of input stream in output surface.

mfxU32 **DstY**
Y coordinate of location of input stream in output surface.

mfxU32 **DstW**
Width of of location of input stream in output surface.

mfxU32 **DstH**
Height of of location of input stream in output surface.

mfxU16 **LumaKeyEnable**
Non-zero value enables luma keying for the input stream. Luma keying is used to mark some of the areas of the frame with specified luma values as transparent. It may, for example, be used for closed captioning.

mfxU16 **LumaKeyMin**
Minimum value of luma key, inclusive. Pixels whose luma values fit in this range are rendered transparent.

mfxU16 **LumaKeyMax**
Maximum value of luma key, inclusive. Pixels whose luma values fit in this range are rendered transparent.

mfxU16 **GlobalAlphaEnable**
Non-zero value enables global alpha blending for this input stream.

mfxU16 **GlobalAlpha**
Alpha value for this stream. Should be in the range of 0 to 255, where 0 is transparent and 255 is opaque.

mfxU16 **PixelAlphaEnable**
Non-zero value enables per pixel alpha blending for this input stream. The stream should have RGB color format.

mfxU16 **TileId**
Specify the tile this video stream is assigned to. Should be in the range of 0 to NumTiles. Valid only if NumTiles > 0.

mfxVPPStat

struct mfxVPPStat
Returns statistics collected during video processing.

Public Members

mfxU32 **NumFrame**
Total number of frames processed.

mfxU32 **NumCachedFrame**
Number of internally cached frames.

Protected Structures

Protected structures.

API

- *mfxExtCencParam*

mfxExtCencParam

struct _mfxExtCencParam
Used to pass the decryption status report index for the Common Encryption usage model. The application can attach this extended buffer to the *mfxBitstream* structure at runtime.

Public Members

mfxExtBuffer **Header**
Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_CENC_PARAM.

mfxU32 **StatusReportIndex**
Decryption status report index.

DECODDE_VPP Structures

Structures used by *DECODE_VPP* only.

API

- *mfxSurfaceArray*
- *mfxVideoChannelParam*
- *mfxExtInCrops*

mfxSurfaceArray

struct mfxSurfaceArray
The structure is reference counted object to return array of surfaces allocated and processed by the library.

Public Members

mfxHDL **Context**
The context of the memory interface. User should not touch (change, set, null) this pointer.

mfxStructVersion **Version**
The version of the structure.

mfxStatus (***AddRef**)(**struct** *mfxSurfaceArray* *surface_array)
Increments the internal reference counter of the surface. The surface is not destroyed until the surface is released using the (*Release) function. (*AddRef) should be used each time a new link to the surface is created (for example, copy structure) for proper surface management.

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfaceArray->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid *mfxSurfaceArray*.

mfxStatus (***Release**)(**struct** *mfxSurfaceArray* *surface_array)
Decrements the internal reference counter of the surface. (*Release) should be called after using the (*AddRef) function to add a surface or when allocation logic requires it.

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If surface is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfaceArray->Context is invalid (for example NULL).

MFX_ERR_UNDEFINED_BEHAVIOR If Reference Counter of surface is zero before call.

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface_array`: Valid *mfxSurfaceArray*.

mfxStatus (***GetRefCounter**)(**struct** *mfxSurfaceArray* *surface_array, *mfxU32* *counter)
Returns current reference counter of *mfxSurfaceArray* structure.

Return MFX_ERR_NONE If no error. MFX_ERR_NULL_PTR If surface or counter is NULL.

MFX_ERR_INVALID_HANDLE If mfxSurfaceArray->Context is invalid (for example NULL).

MFX_ERR_UNKNOWN Any internal error.

Parameters

- `[in] surface`: Valid surface_array.
- `[out] counter`: Sets counter to the current reference counter value.

mfxFrameSurface1 ****Surfaces**
The array of pointers to *mfxFrameSurface1*. *mfxFrameSurface1* surfaces are allocated by the same agent who allocates *mfxSurfaceArray*.

mfxU32 **NumSurfaces**
The size of array of pointers to *mfxFrameSurface1*.

mfxVideoChannelParam

struct mfxVideoChannelParam
The structure is used for VPP channels initializtion in Decode_VPP component.

Public Members

mfxFrameInfo **VPP**
The configuration parameters of VPP filters per each channel.

mfxU16 **Protected**
Specifies the content protection mechanism.

mfxU16 **IOPattern**
Output memory access types for SDK functions.

mfxExtBuffer ****ExtParam**
Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

mfxU16 **NumExtParam**
The number of extra configuration structures attached to the structure.

mfxExtInCrops

struct mfxExtInCrops
The structure contains crop parameters which applied by Decode_VPP component to input surfaces before video processing operation. It is used for letterboxing operations.

Public Members

mfxRect **Crops**
Extension buffer header. BufferId must be equal to MFX_EXTBUFF_CROPS. Crops parameters for letterboxing operations.

11.4.3 Enumerator Reference

Angle

The Angle enumerator itemizes valid rotation angles.

enumerator MFX_ANGLE_0
0 degrees.

enumerator MFX_ANGLE_90
90 degrees.

enumerator MFX_ANGLE_180
180 degrees.

enumerator MFX_ANGLE_270
270 degrees.

BitstreamDataFlag

The BitstreamDataFlag enumerator uses bit-ORed values to itemize additional information about the bitstream buffer.

enumerator MFX_BITSTREAM_NO_FLAG
The bitstream doesn't contain any flags.

enumerator MFX_BITSTREAM_COMPLETE_FRAME
The bitstream buffer contains a complete frame or complementary field pair of data for the bitstream. For decoding, this means that the decoder can proceed with this buffer without waiting for the start of the next frame, which effectively reduces decoding latency. If this flag is set, but the bitstream buffer contains incomplete frame or pair of field, then decoder will produce corrupted output.

enumerator MFX_BITSTREAM_EOS
The bitstream buffer contains the end of the stream. For decoding, this means that the application does not have any additional bitstream data to send to decoder.

BPSEIControl

The BPSEIControl enumerator is used to control insertion of buffering period SEI in the encoded bitstream.

enumerator MFX_BPSEI_DEFAULT
encoder decides when to insert BP SEI.

enumerator MFX_BPSEI_IFRAME
BP SEI should be inserted with every I-frame

BRCStatus

The BRCStatus enumerator itemizes instructions to the encoder by `mfxExtBrc::Update`.

enumerator MFX_BRC_OK
CodedFrameSize is acceptable, no further recoding/padding/skip required, proceed to next frame.

enumerator MFX_BRC_BIG_FRAME
Coded frame is too big, recoding required.

enumerator MFX_BRC_SMALL_FRAME
Coded frame is too small, recoding required.

enumerator MFX_BRC_PANIC_BIG_FRAME
Coded frame is too big, no further recoding possible - skip frame.

enumerator MFX_BRC_PANIC_SMALL_FRAME
Coded frame is too small, no further recoding possible - required padding to *mfxBRCFrameStatus::MinFrameSize*.

BRefControl

The BRefControl enumerator is used to control usage of B frames as reference in AVC encoder.

enumerator MFX_B_REF_UNKNOWN
Default value, it is up to the encoder to use B-frames as reference.

enumerator MFX_B_REF_OFF
Do not use B-frames as reference.

enumerator MFX_B_REF_PYRAMID
Arrange B-frames in so-called “B pyramid” reference structure.

ChromaFormatIdc

The ChromaFormatIdc enumerator itemizes color-sampling formats.

enumerator MFX_CHROMAFORMAT_MONOCHROME
Monochrome.

enumerator MFX_CHROMAFORMAT_YUV420
4:2:0 color.

enumerator MFX_CHROMAFORMAT_YUV422
4:2:2 color.

enumerator MFX_CHROMAFORMAT_YUV444
4:4:4 color.

enumerator MFX_CHROMAFORMAT_YUV400
Equal to monochrome.

enumerator MFX_CHROMAFORMAT_YUV411
4:1:1 color.

enumerator MFX_CHROMAFORMAT_YUV422H
4:2:2 color, horizontal sub-sampling. It is equal to 4:2:2 color.

enumerator MFX_CHROMAFORMAT_YUV422V
4:2:2 color, vertical sub-sampling.

enumerator MFX_CHROMAFORMAT_RESERVED1
Reserved.

enumerator MFX_CHROMAFORMAT_JPEG_SAMPLING
Color sampling specified via *mfxInfoMFX::SamplingFactorH* and SamplingFactorV.

ChromaSiting

The ChromaSiting enumerator defines chroma location. Use bit-OR'ed values to specify the desired location.

enumerator MFX_CHROMA_SITING_UNKNOWN
Unspecified.

enumerator MFX_CHROMA_SITING_VERTICAL_TOP
Chroma samples are co-sited vertically on the top with the luma samples.

enumerator MFX_CHROMA_SITING_VERTICAL_CENTER
Chroma samples are not co-sited vertically with the luma samples.

enumerator MFX_CHROMA_SITING_VERTICAL_BOTTOM
Chroma samples are co-sited vertically on the bottom with the luma samples.

enumerator MFX_CHROMA_SITING_HORIZONTAL_LEFT
Chroma samples are co-sited horizontally on the left with the luma samples.

enumerator MFX_CHROMA_SITING_HORIZONTAL_CENTER
Chroma samples are not co-sited horizontally with the luma samples.

CodecFormatFourCC

The CodecFormatFourCC enumerator itemizes codecs in the FourCC format.

enumerator MFX_CODEC_AVC
AVC, H.264, or MPEG-4, part 10 codec.

enumerator MFX_CODEC_HEVC
HEVC codec.

enumerator MFX_CODEC_MPEG2
MPEG-2 codec.

enumerator MFX_CODEC_VC1
VC-1 codec.

enumerator MFX_CODEC_VP9
VP9 codec.

enumerator MFX_CODEC_AV1
AV1 codec.

enumerator MFX_CODEC_JPEG
JPEG codec

CodecLevel

The CodecLevel enumerator itemizes codec levels for all codecs.

enumerator MFX_LEVEL_UNKNOWN
Unspecified level.

H.264 Level 1-1.3

enumerator MFX_LEVEL_AVC_1

enumerator MFX_LEVEL_AVC_1b

enumerator MFX_LEVEL_AVC_11

enumerator MFX_LEVEL_AVC_12

enumerator MFX_LEVEL_AVC_13

H.264 Level 2-2.2

enumerator MFX_LEVEL_AVC_2

enumerator MFX_LEVEL_AVC_21

enumerator MFX_LEVEL_AVC_22

H.264 Level 3-3.2

enumerator MFX_LEVEL_AVC_3

enumerator MFX_LEVEL_AVC_31

enumerator MFX_LEVEL_AVC_32

H.264 Level 4-4.2

enumerator MFX_LEVEL_AVC_4

enumerator MFX_LEVEL_AVC_41

enumerator MFX_LEVEL_AVC_42

H.264 Level 5-5.2

enumerator MFX_LEVEL_AVC_5

enumerator MFX_LEVEL_AVC_51

enumerator MFX_LEVEL_AVC_52

MPEG2 Levels

enumerator MFX_LEVEL_MPEG2_LOW

enumerator MFX_LEVEL_MPEG2_MAIN

enumerator MFX_LEVEL_MPEG2_HIGH

enumerator MFX_LEVEL_MPEG2_HIGH1440

VC-1 Level Low (Simple and Main Profiles)

enumerator MFX_LEVEL_VC1_LOW

enumerator MFX_LEVEL_VC1_MEDIAN

enumerator MFX_LEVEL_VC1_HIGH

VC-1 Advanced Profile Levels

enumerator MFX_LEVEL_VC1_0

enumerator MFX_LEVEL_VC1_1

enumerator MFX_LEVEL_VC1_2

enumerator MFX_LEVEL_VC1_3

enumerator MFX_LEVEL_VC1_4

HEVC Levels

enumerator MFX_LEVEL_HEVC_1

enumerator MFX_LEVEL_HEVC_2

enumerator MFX_LEVEL_HEVC_21

enumerator MFX_LEVEL_HEVC_3

enumerator MFX_LEVEL_HEVC_31

enumerator MFX_LEVEL_HEVC_4

enumerator MFX_LEVEL_HEVC_41

enumerator MFX_LEVEL_HEVC_5

enumerator MFX_LEVEL_HEVC_51

enumerator MFX_LEVEL_HEVC_52

enumerator MFX_LEVEL_HEVC_6

enumerator MFX_LEVEL_HEVC_61

enumerator MFX_LEVEL_HEVC_62

AV1 Levels

enumerator MFX_LEVEL_AV1_2

enumerator MFX_LEVEL_AV1_21

enumerator MFX_LEVEL_AV1_22

enumerator MFX_LEVEL_AV1_23

enumerator MFX_LEVEL_AV1_3

enumerator MFX_LEVEL_AV1_31

enumerator MFX_LEVEL_AV1_32

enumerator MFX_LEVEL_AV1_33

enumerator MFX_LEVEL_AV1_4

enumerator MFX_LEVEL_AV1_41

enumerator MFX_LEVEL_AV1_42

enumerator MFX_LEVEL_AV1_43

enumerator MFX_LEVEL_AV1_5

enumerator MFX_LEVEL_AV1_51

enumerator MFX_LEVEL_AV1_52

enumerator MFX_LEVEL_AV1_53

enumerator MFX_LEVEL_AV1_6

enumerator MFX_LEVEL_AV1_61

enumerator MFX_LEVEL_AV1_62

enumerator MFX_LEVEL_AV1_63

enumerator MFX_LEVEL_AV1_7

enumerator MFX_LEVEL_AV1_71

enumerator MFX_LEVEL_AV1_72

enumerator MFX_LEVEL_AV1_73

CodecProfile

The CodecProfile enumerator itemizes codec profiles for all codecs.

enumerator MFX_PROFILE_UNKNOWN
Unspecified profile.

H.264 Profiles

enumerator MFX_PROFILE_AVC_BASELINE

enumerator MFX_PROFILE_AVC_MAIN

enumerator MFX_PROFILE_AVC_EXTENDED

enumerator MFX_PROFILE_AVC_HIGH

enumerator MFX_PROFILE_AVC_HIGH10

enumerator MFX_PROFILE_AVC_HIGH_422

enumerator MFX_PROFILE_AVC_CONSTRAINED_BASELINE

enumerator MFX_PROFILE_AVC_CONSTRAINED_HIGH

AV1 Profiles

enumerator MFX_PROFILE_AV1_MAIN

enumerator MFX_PROFILE_AV1_HIGH

enumerator MFX_PROFILE_AV1_PRO

VC-1 Profiles

enumerator MFX_PROFILE_VC1_SIMPLE

enumerator MFX_PROFILE_VC1_MAIN

enumerator MFX_PROFILE_VC1_ADVANCED

VP8 Profiles

enumerator MFX_PROFILE_VP8_0

enumerator MFX_PROFILE_VP8_1

enumerator MFX_PROFILE_VP8_2

enumerator MFX_PROFILE_VP8_3

VP9 Profiles

enumerator MFX_PROFILE_VP9_0

enumerator MFX_PROFILE_VP9_1

enumerator MFX_PROFILE_VP9_2

enumerator MFX_PROFILE_VP9_3

H.264 Constraints

Combined with H.264 profile, these flags impose additional constraints. See the H.264 specification for the list of constraints.

enumerator MFX_PROFILE_AVC_CONSTRAINT_SET0

enumerator MFX_PROFILE_AVC_CONSTRAINT_SET1

enumerator MFX_PROFILE_AVC_CONSTRAINT_SET2

enumerator MFX_PROFILE_AVC_CONSTRAINT_SET3

enumerator MFX_PROFILE_AVC_CONSTRAINT_SET4

enumerator MFX_PROFILE_AVC_CONSTRAINT_SET5

JPEG Profiles

enumerator MFX_PROFILE_JPEG_BASELINE
Baseline JPEG profile.

CodingOptionValue

The CodingOptionValue enumerator defines a three-state coding option setting.

enumerator MFX_CODINGOPTION_UNKNOWN
Unspecified.

enumerator MFX_CODINGOPTION_ON
Coding option set.

enumerator MFX_CODINGOPTION_OFF
Coding option not set.

enumerator MFX_CODINGOPTION_ADAPTIVE
Reserved.

ColorFourCC

The ColorFourCC enumerator itemizes color formats.

enumerator MFX_FOURCC_NV12
NV12 color planes. Native format for 4:2:0/8b Gen hardware implementation.

enumerator MFX_FOURCC_NV21
Same as NV12 but with weaved V and U values.

enumerator MFX_FOURCC_YV12
YV12 color planes.

enumerator MFX_FOURCC_IYUV
Same as YV12 except that the U and V plane order is reversed.

enumerator MFX_FOURCC_I420
Alias for the IYUV color format.

enumerator MFX_FOURCC_NV16
4:2:2 color format with similar to NV12 layout.

enumerator MFX_FOURCC_YUY2
YUY2 color planes.

enumerator MFX_FOURCC_RGB565
2 bytes per pixel, uint16 in little-endian format, where 0-4 bits are blue, bits 5-10 are green and bits 11-15 are red.

enumerator MFX_FOURCC_RGBP
RGB 24 bit planar layout (3 separate channels, 8-bits per sample each). This format should be mapped to D3DFMT_R8G8B8 or VA_FOURCC_RGBP.

enumerator MFX_FOURCC_RGB4
RGB4 (RGB32) color planes. BGRA is the order, 'B' is 8 MSBs, then 8 bits for 'G' channel, then 'R' and 'A' channels.

enumerator MFX_FOURCC_BGRA
Alias for the RGB4 color format.

enumerator MFX_FOURCC_P8
Internal color format. The application should use the following functions to create a surface that corresponds to the Direct3D* version in use.

For Direct3D* 9: IDirectXVideoDecoderService::CreateSurface()

For Direct3D* 11: ID3D11Device::CreateBuffer()

enumerator MFX_FOURCC_P8_TEXTURE
Internal color format. The application should use the following functions to create a surface that corresponds to the Direct3D* version in use.

For Direct3D 9: IDirectXVideoDecoderService::CreateSurface()

For Direct3D 11: ID3D11Device::CreateTexture2D()

enumerator MFX_FOURCC_P010
P010 color format. This is 10 bit per sample format with similar to NV12 layout. This format should be mapped to DXGI_FORMAT_P010.

enumerator MFX_FOURCC_I010
10-bit YUV 4:2:0, each component has its own plane.

enumerator MFX_FOURCC_P016
P016 color format. This is 16 bit per sample format with similar to NV12 layout. This format should be mapped to DXGI_FORMAT_P016.

enumerator MFX_FOURCC_P210
10 bit per sample 4:2:2 color format with similar to NV12 layout.

enumerator MFX_FOURCC_BGR4
RGBA color format. It is similar to MFX_FOURCC_RGB4 but with different order of channels. 'R' is 8 MSBs, then 8 bits for 'G' channel, then 'B' and 'A' channels.

enumerator MFX_FOURCC_A2RGB10
10 bits ARGB color format packed in 32 bits. 'A' channel is two MSBs, then 'R', then 'G' and then 'B' channels. This format should be mapped to DXGI_FORMAT_R10G10B10A2_UNORM or D3DFMT_A2R10G10B10.

enumerator MFX_FOURCC_ARGB16
10 bits ARGB color format packed in 64 bits. 'A' channel is 16 MSBs, then 'R', then 'G' and then 'B' channels. This format should be mapped to DXGI_FORMAT_R16G16B16A16_UINT or D3DFMT_A16B16G16R16 formats.

enumerator MFX_FOURCC_ABGR16
10 bits ABGR color format packed in 64 bits. 'A' channel is 16 MSBs, then 'B', then 'G' and then 'R' channels. This format should be mapped to DXGI_FORMAT_R16G16B16A16_UINT or D3DFMT_A16B16G16R16 formats.

enumerator MFX_FOURCC_R16
16 bits single channel color format. This format should be mapped to DXGI_FORMAT_R16_TYPELESS or D3DFMT_R16F.

enumerator MFX_FOURCC_AYUV
YUV 4:4:4, AYUV color format. This format should be mapped to DXGI_FORMAT_AYUV.

enumerator MFX_FOURCC_AYUV_RGB4
RGB4 stored in AYUV surface. This format should be mapped to DXGI_FORMAT_AYUV.

enumerator MFX_FOURCC_UYVY
UYVY color planes. Same as YUY2 except the byte order is reversed.

enumerator MFX_FOURCC_Y210
10 bit per sample 4:2:2 packed color format with similar to YUY2 layout. This format should be mapped to DXGI_FORMAT_Y210.

enumerator MFX_FOURCC_Y410
10 bit per sample 4:4:4 packed color format. This format should be mapped to DXGI_FORMAT_Y410.

enumerator MFX_FOURCC_Y216
16 bit per sample 4:2:2 packed color format with similar to YUY2 layout. This format should be mapped to DXGI_FORMAT_Y216.

enumerator MFX_FOURCC_Y416
16 bit per sample 4:4:4 packed color format. This format should be mapped to DXGI_FORMAT_Y416.

ContentInfo

The ContentInfo enumerator itemizes content types for the encoding session.

enumerator MFX_CONTENT_UNKNOWN

enumerator MFX_CONTENT_FULL_SCREEN_VIDEO

enumerator MFX_CONTENT_NON_VIDEO_SCREEN

Corruption

The Corruption enumerator itemizes the decoding corruption types. It is a bit-OR'ed value of the following.

enumerator MFX_CORRUPTION_NO
No corruption.

enumerator MFX_CORRUPTION_MINOR
Minor corruption in decoding certain macro-blocks.

enumerator MFX_CORRUPTION_MAJOR
Major corruption in decoding the frame - incomplete data, for example.

enumerator MFX_CORRUPTION_ABSENT_TOP_FIELD
Top field of frame is absent in bitstream. Only bottom field has been decoded.

enumerator MFX_CORRUPTION_ABSENT_BOTTOM_FIELD
Bottom field of frame is absent in bitstream. Only top filed has been decoded.

enumerator MFX_CORRUPTION_REFERENCE_FRAME
Decoding used a corrupted reference frame. A corrupted reference frame was used for decoding this frame. For example, if the frame uses a reference frame that was decoded with minor/major corruption flag, then this frame is also marked with a reference corruption flag.

enumerator MFX_CORRUPTION_REFERENCE_LIST
The reference list information of this frame does not match what is specified in the Reference Picture Marking Repetition SEI message. (ITU-T H.264 D.1.8 dec_ref_pic_marking_repetition)

Note: Flag MFX_CORRUPTION_ABSENT_TOP_FIELD/MFX_CORRUPTION_ABSENT_BOTTOM_FIELD is set by the AVC decoder when it detects that one of fields is not present in the bitstream. Which field is absent depends on value of bottom_field_flag (ITU-T* H.264 7.4.3).

DeinterlacingMode

The DeinterlacingMode enumerator itemizes VPP deinterlacing modes.

enumerator MFX_DEINTERLACING_BOB
BOB deinterlacing mode.

enumerator MFX_DEINTERLACING_ADVANCED
Advanced deinterlacing mode.

enumerator MFX_DEINTERLACING_AUTO_DOUBLE
Auto mode with deinterlacing double frame rate output.

enumerator MFX_DEINTERLACING_AUTO_SINGLE
Auto mode with deinterlacing single frame rate output.

enumerator MFX_DEINTERLACING_FULL_FR_OUT
Deinterlace only mode with full frame rate output.

enumerator MFX_DEINTERLACING_HALF_FR_OUT
Deinterlace only Mode with half frame rate output.

enumerator MFX_DEINTERLACING_24FPS_OUT
24 fps fixed output mode.

enumerator MFX_DEINTERLACING_FIXED_TELECINE_PATTERN
Fixed telecine pattern removal mode.

enumerator MFX_DEINTERLACING_30FPS_OUT
30 fps fixed output mode.

enumerator MFX_DEINTERLACING_DETECT_INTERLACE
Only interlace detection.

enumerator MFX_DEINTERLACING_ADVANCED_NOREF
Advanced deinterlacing mode without using of reference frames.

enumerator MFX_DEINTERLACING_ADVANCED_SCD
Advanced deinterlacing mode with scene change detection.

enumerator MFX_DEINTERLACING_FIELD_WEAVING
Field weaving.

ErrorTypes

The ErrorTypes enumerator uses bit-ORed values to itemize bitstream error types.

enumerator MFX_ERROR_NO
No error in bitstream.

enumerator MFX_ERROR_PPS
Invalid/corrupted PPS.

enumerator MFX_ERROR_SPS
Invalid/corrupted SPS.

enumerator MFX_ERROR_SLICEHEADER
Invalid/corrupted slice header.

enumerator MFX_ERROR_SLICEDATA
Invalid/corrupted slice data.

enumerator MFX_ERROR_FRAME_GAP
Missed frames.

ExtendedBufferID

The ExtendedBufferID enumerator itemizes and defines identifiers (BufferId) for extended buffers or video processing algorithm identifiers.

enumerator MFX_EXTBUFF_THREADS_PARAM
mfxExtThreadsParam buffer ID

enumerator MFX_EXTBUFF_CODING_OPTION
This extended buffer defines additional encoding controls. See the *mfxExtCodingOption* structure for details. The application can attach this buffer to the structure for encoding initialization.

enumerator MFX_EXTBUFF_CODING_OPTION_SPSPPS
This extended buffer defines sequence header and picture header for encoders and decoders. See the *mfxExtCodingOptionSPSPPS* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization, and for obtaining raw headers from the decoders and encoders.

enumerator MFX_EXTBUFF_VPP_DONOTUSE
This extended buffer defines a list of VPP algorithms that applications should not use. See the *mfxExtVPPDoNotUse* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

enumerator MFX_EXTBUFF_VPP_AUXDATA
This extended buffer defines auxiliary information at the VPP output. See the *mfxExtVppAuxData* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding control.

enumerator MFX_EXTBUFF_VPP_DENOISE
The extended buffer defines control parameters for the VPP denoise filter algorithm. See the *mfxExtVPPDenoise* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

enumerator MFX_EXTBUFF_VPP_SCENE_ANALYSIS

enumerator MFX_EXTBUFF_VPP_PROCAMP
The extended buffer defines control parameters for the VPP ProcAmp filter algorithm. See the *mfxExtVPPProcAmp* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization or to the *mfxFrameData* structure in the *mfxFrameSurface1* structure of output surface for per-frame processing configuration.

enumerator MFX_EXTBUFF_VPP_DETAIL
The extended buffer defines control parameters for the VPP detail filter algorithm. See the *mfxExtVPPDetail* structure for details. The application can attach this buffer to the structure for video processing initialization.

enumerator MFX_EXTBUFF_VIDEO_SIGNAL_INFO
This extended buffer defines video signal type. See the *mfxExtVideoSignalInfo* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization, and for retrieving such information from the decoders.

enumerator MFX_EXTBUFF_VPP_DOUSE
This extended buffer defines a list of VPP algorithms that applications should use. See the *mfxExtVPPDoUse* structure for details. The application can attach this buffer to the structure for video processing initialization.

enumerator MFX_EXTBUFF_AVC_REFLIST_CTRL
This extended buffer defines additional encoding controls for reference list. See the *mfxExtAVCRefListCtrl* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding & decoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION
This extended buffer defines control parameters for the VPP frame rate conversion algorithm. See the *mfxExtVPPFrameRateConversion* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

enumerator MFX_EXTBUFF_PICTURE_TIMING_SEI
This extended buffer configures the H.264 picture timing SEI message. See the *mfxExtPictureTimingSEI* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator MFX_EXTBUFF_AVC_TEMPORAL_LAYERS
This extended buffer configures the structure of temporal layers inside the encoded H.264 bitstream. See the *mfxExtAvcTemporalLayers* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator MFX_EXTBUFF_CODING_OPTION2
This extended buffer defines additional encoding controls. See the *mfxExtCodingOption2* structure for details. The application can attach this buffer to the structure for encoding initialization.

enumerator MFX_EXTBUFF_VPP_IMAGE_STABILIZATION
This extended buffer defines control parameters for the VPP image stabilization filter algorithm. See the *mfxExtVPPImageStab* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

enumerator MFX_EXTBUFF_ENCODER_CAPABILITY
This extended buffer is used to retrieve encoder capability. See the *mfxExtEncoderCapability* structure for details. The application can attach this buffer to the *mfxVideoParam* structure before calling MFXVideoENCODE_Query function.

enumerator MFX_EXTBUFF_ENCODER_RESET_OPTION
This extended buffer is used to control encoder reset behavior and also to query possible encoder reset outcome. See the *mfxExtEncoderResetOption* structure for details. The application can attach this buffer to the *mfxVideoParam* structure before calling MFXVideoENCODE_Query or MFXVideoENCODE_Reset functions.

enumerator MFX_EXTBUFF_ENCODED_FRAME_INFO
This extended buffer is used by the encoder to report additional information about encoded picture. See the *mfxExtAVCEncodedFrameInfo* structure for details. The application can attach this buffer to the *mfxBitstream* structure before calling MFXVideoENCODE_EncodeFrameAsync function.

enumerator MFX_EXTBUFF_VPP_COMPOSITE
This extended buffer is used to control composition of several input surfaces in the one output. In this mode, the VPP skips any other filters. The VPP returns error if any mandatory filter is specified and filter skipped warning for optional filter. The only supported filters are deinterlacing and interlaced scaling.

enumerator MFX_EXTBUFF_VPP_VIDEO_SIGNAL_INFO
This extended buffer is used to control transfer matrix and nominal range of YUV frames. The application should provide it during initialization.

enumerator MFX_EXTBUFF_ENCODER_ROI
This extended buffer is used by the application to specify different Region Of Interests during encoding. The application should provide it at initialization or at runtime.

enumerator MFX_EXTBUFF_VPP_DEINTERLACING
This extended buffer is used by the application to specify different deinterlacing algorithms.

enumerator MFX_EXTBUFF_AVC_REFLISTS
This extended buffer specifies reference lists for the encoder.

enumerator MFX_EXTBUFF_DEC_VIDEO_PROCESSING
See the *mfxExtDecVideoProcessing* structure for details.

enumerator MFX_EXTBUFF_VPP_FIELD_PROCESSING
The extended buffer defines control parameters for the VPP field-processing algorithm. See the *mfxExtVPPFieldProcessing* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization or to the *mfxFrameData* structure during runtime.

enumerator MFX_EXTBUFF_CODING_OPTION3
This extended buffer defines additional encoding controls. See the *mfxExtCodingOption3* structure for details. The application can attach this buffer to the structure for encoding initialization.

enumerator MFX_EXTBUFF_CHROMA_LOC_INFO
This extended buffer defines chroma samples location information. See the *mfxExtChromaLocInfo* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator MFX_EXTBUFF_MBQP
This extended buffer defines per-macroblock QP. See the *mfxExtMBQP* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator MFX_EXTBUFF_MB_FORCE_INTRA
This extended buffer defines per-macroblock force intra flag. See the *mfxExtMBForceIntra* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator MFX_EXTBUFF_HEVC_TILES
This extended buffer defines additional encoding controls for HEVC tiles. See the *mfxExtHEVCTiles* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator MFX_EXTBUFF_MB_DISABLE_SKIP_MAP
This extended buffer defines macroblock map for current frame which forces specified macroblocks to be non skip. See the *mfxExtMBDisableSkipMap* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator MFX_EXTBUFF_HEVC_PARAM
See the *mfxExtHEVCParam* structure for details.

enumerator MFX_EXTBUFF_DECODED_FRAME_INFO
This extended buffer is used by decoders to report additional information about decoded frame. See the *mfxExtDecodedFrameInfo* structure for more details.

enumerator MFX_EXTBUFF_TIME_CODE
See the *mfxExtTimeCode* structure for more details.

enumerator MFX_EXTBUFF_HEVC_REGION
This extended buffer specifies the region to encode. The application can attach this buffer to the *mfxVideoParam* structure during HEVC encoder initialization.

enumerator MFX_EXTBUFF_PRED_WEIGHT_TABLE
See the *mfxExtPredWeightTable* structure for details.

enumerator MFX_EXTBUFF_DIRTY_RECTANGLES
See the mfxExtDitrtyRect structure for details.

enumerator MFX_EXTBUFF_MOVING_RECTANGLES
See the *mfxExtMoveRect* structure for details.

enumerator MFX_EXTBUFF_CODING_OPTION_VPS
See the *mfxExtCodingOptionVPS* structure for details.

enumerator MFX_EXTBUFF_VPP_ROTATION
See the *mfxExtVPPRotation* structure for details.

enumerator MFX_EXTBUFF_ENCODED_SLICES_INFO
See the *mfxExtEncodedSlicesInfo* structure for details.

enumerator MFX_EXTBUFF_VPP_SCALING
See the *mfxExtVPPScaling* structure for details.

enumerator MFX_EXTBUFF_HEVC_REFLIST_CTRL
This extended buffer defines additional encoding controls for reference list. See the *mfxExtAVCRefListCtrl* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding & decoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

enumerator MFX_EXTBUFF_HEVC_REFLISTS
This extended buffer specifies reference lists for the encoder.

enumerator MFX_EXTBUFF_HEVC_TEMPORAL_LAYERS
This extended buffer configures the structure of temporal layers inside the encoded H.264 bitstream. See the

mfxExtAvcTemporalLayers structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator MFX_EXTBUFF_VPP_MIRRORING
See the *mfxExtVPPMirroring* structure for details.

enumerator MFX_EXTBUFF_MV_OVER_PIC_BOUNDARIES
See the *mfxExtMVOverPicBoundaries* structure for details.

enumerator MFX_EXTBUFF_VPP_COLORFILL
See the *mfxExtVPPColorFill* structure for details.

enumerator MFX_EXTBUFF_DECODE_ERROR_REPORT
This extended buffer is used by decoders to report error information before frames get decoded. See the *mfxExtDecodeErrorReport* structure for more details.

enumerator MFX_EXTBUFF_VPP_COLOR_CONVERSION
See the *mfxExtColorConversion* structure for details.

enumerator MFX_EXTBUFF_CONTENT_LIGHT_LEVEL_INFO
This extended buffer configures HDR SEI message. See the *mfxExtContentLightLevelInfo* structure for details.

enumerator MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME
This extended buffer configures HDR SEI message. See the *mfxExtMasteringDisplayColourVolume* structure for details.

enumerator MFX_EXTBUFF_MULTI_FRAME_PARAM
This extended buffer allow to specify multi-frame submission parameters.

enumerator MFX_EXTBUFF_MULTI_FRAME_CONTROL
This extended buffer allow to manage multi-frame submission in runtime.

enumerator MFX_EXTBUFF_ENCODED_UNITS_INFO
See the *mfxExtEncodedUnitsInfo* structure for details.

enumerator MFX_EXTBUFF_VPP_MCTF
This video processing algorithm identifier is used to enable MCTF via *mfxExtVPPDoUse* and together with *mfxExtVppMctf*

enumerator MFX_EXTBUFF_VP9_SEGMENTATION
Extends *mfxVideoParam* structure with VP9 segmentation parameters. See the *mfxExtVP9Segmentation* structure for details.

enumerator MFX_EXTBUFF_VP9_TEMPORAL_LAYERS
Extends *mfxVideoParam* structure with parameters for VP9 temporal scalability. See the *mfxExtVP9TemporalLayers* structure for details.

enumerator MFX_EXTBUFF_VP9_PARAM
Extends *mfxVideoParam* structure with VP9-specific parameters. See the *mfxExtVP9Param* structure for details.

enumerator MFX_EXTBUFF_AVC_ROUNDING_OFFSET
See the *mfxExtAVCRoundingOffset* structure for details.

enumerator MFX_EXTBUFF_PARTIAL_BITSTREAM_PARAM
See the *mfxExtPartialBitstreamParam* structure for details.

enumerator MFX_EXTBUFF_BRC

enumerator MFX_EXTBUFF_VP8_CODING_OPTION
This extended buffer describes VP8 encoder configuration parameters. See the *mfxExtVP8CodingOption* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

enumerator MFX_EXTBUFF_JPEG_QT
This extended buffer defines quantization tables for JPEG encoder.

enumerator MFX_EXTBUFF_JPEG_HUFFMAN
This extended buffer defines Huffman tables for JPEG encoder.

enumerator MFX_EXTBUFF_ENCODER_IPCM_AREA
See the *mfxExtEncoderIPCMArea* structure for details.

enumerator MFX_EXTBUFF_INSERT_HEADERS
See the *mfxExtInsertHeaders* structure for details.

enumerator MFX_EXTBUFF_MVC_SEQ_DESC
This extended buffer describes the MVC stream information of view dependencies, view identifiers, and operation points. See the ITU*-T H.264 specification chapter H.7.3.2.1.4 for details.

enumerator MFX_EXTBUFF_MVC_TARGET_VIEWS
This extended buffer defines target views at the decoder output.

enumerator MFX_EXTBUFF_CENC_PARAM
This structure is used to pass decryption status report index for Common Encryption usage model. See the mfxExtCencParam structure for more details.

enumerator MFX_EXTBUFF_DEVICE_AFFINITY_MASK
See the *mfxExtDeviceAffinityMask* structure for details.

enumerator MFX_EXTBUFF_CROPS
See the *mfxExtInCrops* structure for details.

ExtMemBufferType

enumerator MFX_MEMTYPE_PERSISTENT_MEMORY
Memory page for persistent use.

ExtMemFrameType

The ExtMemFrameType enumerator specifies the memory type of frame. It is a bit-ORed value of one of the following. For information on working with video memory surfaces, see the *Working with Hardware Acceleration section*.

enumerator MFX_MEMTYPE_DXVA2_DECODER_TARGET
Frames are in video memory and belong to video decoder render targets.

enumerator MFX_MEMTYPE_DXVA2_PROCESSOR_TARGET
Frames are in video memory and belong to video processor render targets.

enumerator MFX_MEMTYPE_VIDEO_MEMORY_DECODER_TARGET
Frames are in video memory and belong to video decoder render targets.

enumerator MFX_MEMTYPE_VIDEO_MEMORY_PROCESSOR_TARGET
Frames are in video memory and belong to video processor render targets.

enumerator MFX_MEMTYPE_SYSTEM_MEMORY
The frames are in system memory.

enumerator MFX_MEMTYPE_RESERVED1

enumerator MFX_MEMTYPE_FROM_ENCODE
Allocation request comes from an ENCODE function

enumerator MFX_MEMTYPE_FROM_DECODE
Allocation request comes from a DECODE function

enumerator MFX_MEMTYPE_FROM_VPPIN
Allocation request comes from a VPP function for input frame allocation

enumerator MFX_MEMTYPE_FROM_VPPOUT
Allocation request comes from a VPP function for output frame allocation

enumerator MFX_MEMTYPE_FROM_ENC
Allocation request comes from an ENC function

enumerator MFX_MEMTYPE_INTERNAL_FRAME
Allocation request for internal frames

enumerator MFX_MEMTYPE_EXTERNAL_FRAME
Allocation request for I/O frames

enumerator MFX_MEMTYPE_EXPORT_FRAME
Application requests frame handle export to some associated object. For Linux frame handle can be considered to be exported to DRM Prime FD, DRM FLink or DRM FrameBuffer Handle. Specifics of export types and export procedure depends on external frame allocator implementation

enumerator MFX_MEMTYPE_SHARED_RESOURCE
For DX11 allocation use shared resource bind flag.

enumerator MFX_MEMTYPE_VIDEO_MEMORY_ENCODER_TARGET
Frames are in video memory and belong to video encoder render targets.

Frame Data Flags

enumerator MFX_TIMESTAMP_UNKNOWN
Indicates that time stamp is unknown for this frame/bitstream portion.

enumerator MFX_FRAMEORDER_UNKNOWN
Unused entry or API functions that generate the frame output do not use this frame.

enumerator MFX_FRAMEDATA_TIMESTAMP_UNKNOWN
Indicates the time stamp of this frame is unknown and will be calculated by SDK.

enumerator MFX_FRAMEDATA_ORIGINAL_TIMESTAMP
Indicates the time stamp of this frame is not calculated and is a pass-through of the original time stamp.

FrameType

The FrameType enumerator itemizes frame types. Use bit-ORed values to specify all that apply.

enumerator MFX_FRAMETYPE_UNKNOWN
Frame type is unspecified.

enumerator MFX_FRAMETYPE_I
This frame or the first field is encoded as an I-frame/field.

enumerator MFX_FRAMETYPE_P
This frame or the first field is encoded as an P-frame/field.

enumerator MFX_FRAMETYPE_B
This frame or the first field is encoded as an B-frame/field.

enumerator MFX_FRAMETYPE_S
This frame or the first field is either an SI- or SP-frame/field.

enumerator MFX_FRAMETYPE_REF
This frame or the first field is encoded as a reference.

enumerator MFX_FRAMETYPE_IDR
This frame or the first field is encoded as an IDR.

enumerator MFX_FRAMETYPE_xI
The second field is encoded as an I-field.

enumerator MFX_FRAMETYPE_xP
The second field is encoded as an P-field.

enumerator MFX_FRAMETYPE_xB
The second field is encoded as an S-field.

enumerator MFX_FRAMETYPE_xS
The second field is an SI- or SP-field.

enumerator MFX_FRAMETYPE_xREF
The second field is encoded as a reference.

enumerator MFX_FRAMETYPE_xIDR
The second field is encoded as an IDR.

FrcAlgm

The FrcAlgm enumerator itemizes frame rate conversion algorithms. See description of mfxExtVPPFrameRateConversion structure for more details.

enumerator MFX_FRCALGM_PRESERVE_TIMESTAMP
Frame dropping/repetition based frame rate conversion algorithm with preserved original time stamps. Any inserted frames will carry MFX_TIMESTAMP_UNKNOWN.

enumerator MFX_FRCALGM_DISTRIBUTED_TIMESTAMP
Frame dropping/repetition based frame rate conversion algorithm with distributed time stamps. The algorithm distributes output time stamps evenly according to the output frame rate.

enumerator MFX_FRCALGM_FRAME_INTERPOLATION
Frame rate conversion algorithm based on frame interpolation. This flag may be combined with MFX_FRCALGM_PRESERVE_TIMESTAMP or MFX_FRCALGM_DISTRIBUTED_TIMESTAMP flags.

GeneralConstraintFlags

The GeneralConstraintFlags enumerator uses bit-ORed values to itemize HEVC bitstream indications for specific profiles. Each value indicates for format range extensions profiles.

enumerator MFX_HEVC_CONSTR_REXT_MAX_12BIT

enumerator MFX_HEVC_CONSTR_REXT_MAX_10BIT

enumerator MFX_HEVC_CONSTR_REXT_MAX_8BIT

enumerator MFX_HEVC_CONSTR_REXT_MAX_422CHROMA

enumerator MFX_HEVC_CONSTR_REXT_MAX_420CHROMA

enumerator MFX_HEVC_CONSTR_REXT_MAX_MONOCHROME

enumerator MFX_HEVC_CONSTR_REXT_INTRA

enumerator MFX_HEVC_CONSTR_REXT_ONE_PICTURE_ONLY

enumerator MFX_HEVC_CONSTR_REXT_LOWER_BIT_RATE

GopOptFlag

The GopOptFlag enumerator itemizes special properties in the GOP (Group of Pictures) sequence.

enumerator MFX_GOP_CLOSED
The encoder generates closed GOP if this flag is set. Frames in this GOP do not use frames in previous GOP as reference.

The encoder generates open GOP if this flag is not set. In this GOP frames prior to the first frame of GOP in display order may use frames from previous GOP as reference. Frames subsequent to the first frame of GOP in display order do not use frames from previous GOP as reference.

The AVC encoder ignores this flag if IdrInterval in *mfxInfoMFX* structure is set to 0, i.e. if every GOP starts from IDR frame. In this case, GOP is encoded as closed.

This flag does not affect long-term reference frames.

enumerator MFX_GOP_STRICT
The encoder must strictly follow the given GOP structure as defined by parameter GopPicSize, GopRefDist etc in the *mfxVideoParam* structure. Otherwise, the encoder can adapt the GOP structure for better efficiency, whose range is constrained by parameter GopPicSize and GopRefDist etc. See also description of AdaptiveI and AdaptiveB fields in the *mfxExtCodingOption2* structure.

GPUCopy

enumerator MFX_GPUCOPY_DEFAULT
Use default mode for the legacy Intel(r) Media SDK implementation.

enumerator MFX_GPUCOPY_ON
Enable GPU accelerated copying.

enumerator MFX_GPUCOPY_OFF
Disable GPU accelerated copying.

HEVC Profiles

enumerator MFX_PROFILE_HEVC_MAIN

enumerator MFX_PROFILE_HEVC_MAIN10

enumerator MFX_PROFILE_HEVC_MAINSP

enumerator MFX_PROFILE_HEVC_REXT

enumerator MFX_PROFILE_HEVC_SCC

HEVC Tiers

enumerator MFX_TIER_HEVC_MAIN

enumerator MFX_TIER_HEVC_HIGH

HEVCRegionEncoding

The HEVCRegionEncoding enumerator itemizes HEVC region's encoding.

enumerator MFX_HEVC_REGION_ENCODING_ON

enumerator MFX_HEVC_REGION_ENCODING_OFF

HEVCRegionType

The HEVCRegionType enumerator itemizes type of HEVC region.

enumerator MFX_HEVC_REGION_SLICE
Slice type.

ImageStabMode

The ImageStabMode enumerator itemizes image stabilization modes. See description of mfxExtVPPImageStab structure for more details.

enumerator MFX_IMAGESTAB_MODE_UPSCALE
Upscale mode.

enumerator MFX_IMAGESTAB_MODE_BOXING
Boxing mode.

InsertHDRPayload

The InsertHDRPayload enumerator itemizes HDR payloads insertion rules.

enumerator MFX_PAYLOAD_OFF
Do not insert payload.

enumerator MFX_PAYLOAD_IDR
Insert payload on IDR frames.

InterpolationMode

The InterpolationMode enumerator specifies type of interpolation method used by VPP scaling filter.

enumerator MFX_INTERPOLATION_DEFAULT
Default interpolation mode for scaling. Library selects the most appropriate scaling method.

enumerator MFX_INTERPOLATION_NEAREST_NEIGHBOR
Nearest neighbor interpolation method.

enumerator MFX_INTERPOLATION_BILINEAR
Bilinear interpolation method.

enumerator MFX_INTERPOLATION_ADVANCED
Advanced interpolation method is defined by each implementation and usually gives best quality.

IntraPredBlockSize/InterPredBlockSize

IntraPredBlockSize/InterPredBlockSize specifies minimum block size of inter-prediction.

enumerator MFX_BLOCKSIZE_UNKNOWN
Unspecified.

enumerator MFX_BLOCKSIZE_MIN_16X16
16x16 minimum block size.

enumerator MFX_BLOCKSIZE_MIN_8X8
8x8 minimum block size. May be 16x16 or 8x8.

enumerator MFX_BLOCKSIZE_MIN_4X4
4x4 minimum block size. May be 16x16, 8x8, or 4x4.

IntraRefreshTypes

The IntraRefreshTypes enumerator itemizes types of intra refresh.

enumerator MFX_REFRESH_NO
Encode without refresh.

enumerator MFX_REFRESH_VERTICAL
Vertical refresh, by column of MBs.

enumerator MFX_REFRESH_HORIZONTAL
Horizontal refresh, by rows of MBs.

enumerator MFX_REFRESH_SLICE
Horizontal refresh by slices without overlapping.

IOPattern

The IOPattern enumerator itemizes memory access patterns for API functions. Use bit-ORed values to specify input and output access patterns.

enumerator MFX_IOPATTERN_IN_VIDEO_MEMORY
Input to functions is a video memory surface.

enumerator MFX_IOPATTERN_IN_SYSTEM_MEMORY
Input to functions is a linear buffer directly in system memory or in system memory through an external allocator.

enumerator MFX_IOPATTERN_OUT_VIDEO_MEMORY
Output to functions is a video memory surface.

enumerator MFX_IOPATTERN_OUT_SYSTEM_MEMORY
Output to functions is a linear buffer directly in system memory or in system memory through an external allocator.

JPEGColorFormat

The JPEGColorFormat enumerator itemizes the JPEG color format options.

enumerator MFX_JPEG_COLORFORMAT_UNKNOWN

enumerator MFX_JPEG_COLORFORMAT_YCbCr
Unknown color format. The decoder tries to determine color format from available in bitstream information. If such information is not present, then MFX_JPEG_COLORFORMAT_YCbCr color format is assumed.

enumerator MFX_JPEG_COLORFORMAT_RGB
Bitstream contains Y, Cb and Cr components.

JPEGScanType

The JPEGScanType enumerator itemizes the JPEG scan types.

enumerator MFX_SCANTYPE_UNKNOWN
Unknown scan type.

enumerator MFX_SCANTYPE_INTERLEAVED
Interleaved scan.

enumerator MFX_SCANTYPE_NONINTERLEAVED
Non-interleaved scan.

LongTermIdx

The LongTermIdx specifies long term index of picture control

enumerator MFX_LONGTERM_IDX_NO_IDX
Long term index of picture is undefined.

LookAheadDownSampling

The LookAheadDownSampling enumerator is used to control down sampling in look ahead bitrate control mode in AVC encoder.

enumerator MFX_LOOKAHEAD_DS_UNKNOWN
Default value, it is up to the encoder what down sampling value to use.

enumerator MFX_LOOKAHEAD_DS_OFF
Do not use down sampling, perform estimation on original size frames. This is the slowest setting that produces the best quality.

enumerator MFX_LOOKAHEAD_DS_2x
Down sample frames two times before estimation.

enumerator MFX_LOOKAHEAD_DS_4x
Down sample frames four times before estimation. This option may significantly degrade quality.

MBQPMode

The MBQPMode enumerator itemizes QP update modes.

enumerator MFX_MBQP_MODE_QP_VALUE
QP array contains QP values.

enumerator MFX_MBQP_MODE_QP_DELTA
QP array contains deltas for QP.

enumerator MFX_MBQP_MODE_QP_ADAPTIVE
QP array contains deltas for QP or absolute QP values.

mfxComponentType

enum mfxComponentType
Describes type of workload passed to MFXQueryAdapters.

Values:

enumerator MFX_COMPONENT_ENCODE
Encode workload.

enumerator MFX_COMPONENT_DECODE
Decode workload.

enumerator MFX_COMPONENT_VPP
VPP workload.

mfxHandleType

enum mfxHandleType
The mfxHandleType enumerator itemizes system handle types that implementations might use.

Values:

enumerator MFX_HANDLE_DIRECT3D_DEVICE_MANAGER9
Pointer to the IDirect3DDeviceManager9 interface. See Working with Microsoft* DirectX* Applications for more details on how to use this handle.

enumerator MFX_HANDLE_D3D9_DEVICE_MANAGER
Pointer to the IDirect3DDeviceManager9 interface. See Working with Microsoft* DirectX* Applications for more details on how to use this handle.

enumerator MFX_HANDLE_RESERVED1

enumerator MFX_HANDLE_D3D11_DEVICE
Pointer to the ID3D11Device interface. See Working with Microsoft* DirectX* Applications for more details on how to use this handle.

enumerator MFX_HANDLE_VA_DISPLAY
Pointer to VADisplay interface. See Working with VA-API Applications for more details on how to use this handle.

enumerator MFX_HANDLE_RESERVED3

enumerator MFX_HANDLE_VA_CONFIG_ID
Pointer to VAConfigID interface. It represents external VA config for Common Encryption usage model.

enumerator MFX_HANDLE_VA_CONTEXT_ID
Pointer to VAContextID interface. It represents external VA context for Common Encryption usage model.

enumerator MFX_HANDLE_CM_DEVICE
Pointer to CmDevice interface (Intel(r) C for Metal Runtime).

enumerator MFX_HANDLE_HDDLUNITE_WORKLOADCONTEXT
Pointer to HddlUnite::WorkloadContext interface.

mfxIMPL

typedef *mfxI32* **mfxIMPL**
This enumerator itemizes implementation types. The implementation type is a bit OR'ed value of the base type and any decorative flags.

Note This enumerator is for legacy dispatcher compatibility only. The new dispatcher does not use it.

enumerator MFX_IMPL_AUTO
Auto Selection/In or Not Supported/Out.

enumerator MFX_IMPL_SOFTWARE
Pure software implementation.

enumerator MFX_IMPL_HARDWARE
Hardware accelerated implementation (default device).

enumerator MFX_IMPL_AUTO_ANY
Auto selection of any hardware/software implementation.

enumerator MFX_IMPL_HARDWARE_ANY
Auto selection of any hardware implementation.

enumerator MFX_IMPL_HARDWARE2
Hardware accelerated implementation (2nd device).

enumerator MFX_IMPL_HARDWARE3
Hardware accelerated implementation (3rd device).

enumerator MFX_IMPL_HARDWARE4
Hardware accelerated implementation (4th device).

enumerator MFX_IMPL_RUNTIME
This value cannot be used for session initialization. It may be returned by the MFXQueryIMPL function to show that the session has been initialized in run-time mode.

enumerator MFX_IMPL_VIA_ANY
Hardware acceleration can go through any supported OS infrastructure. This is the default value. The default value is used by the legacy Intel(r) Media SDK if none of the MFX_IMPL_VIA_xxx flags are specified by the application.

enumerator MFX_IMPL_VIA_D3D9
Hardware acceleration goes through the Microsoft* Direct3D* 9 infrastructure.

enumerator MFX_IMPL_VIA_D3D11
Hardware acceleration goes through the Microsoft* Direct3D* 11 infrastructure.

enumerator MFX_IMPL_VIA_VAAPI
Hardware acceleration goes through the Linux* VA-API infrastructure.

enumerator MFX_IMPL_VIA_HDDLUNITE
Hardware acceleration goes through the HDDL* Unite*.

enumerator MFX_IMPL_UNSUPPORTED
One of the MFXQueryIMPL returns.

MFX_IMPL_BASETYPE(*x*)
The application can use the macro MFX_IMPL_BASETYPE(x) to obtain the base implementation type.

mfxImplCapsDeliveryFormat

enum mfxImplCapsDeliveryFormat
Values:

enumerator MFX_IMPLCAPS_IMPLDESCSTRUCTURE
Deliver capabilities as *mfxImplDescription* structure.

mfxMediaAdapterType

enum mfxMediaAdapterType
The mfxMediaAdapterType enumerator itemizes types of graphics adapters.

Values:

enumerator MFX_MEDIA_UNKNOWN
Unknown type.

enumerator MFX_MEDIA_INTEGRATED
Integrated graphics adapter.

enumerator MFX_MEDIA_DISCRETE
Discrete graphics adapter.

mfxMemoryFlags

enum mfxMemoryFlags
The mfxMemoryFlags enumerator specifies memory access mode.

Values:

enumerator MFX_MAP_READ
The surface is mapped for reading.

enumerator MFX_MAP_WRITE
The surface is mapped for writing.

enumerator MFX_MAP_READ_WRITE
The surface is mapped for reading and writing.

enumerator MFX_MAP_NOWAIT
The mapping would be done immediately without any implicit synchronizations.

Attention This flag is optional.

MfxNalUnitType

Specifies NAL unit types supported by the HEVC encoder.

enumerator MFX_HEVC_NALU_TYPE_UNKNOWN
The encoder will decide what NAL unit type to use.

enumerator MFX_HEVC_NALU_TYPE_TRAIL_N
See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator MFX_HEVC_NALU_TYPE_TRAIL_R
See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator MFX_HEVC_NALU_TYPE_RADL_N
See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator MFX_HEVC_NALU_TYPE_RADL_R
See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator MFX_HEVC_NALU_TYPE_RASL_N
See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator MFX_HEVC_NALU_TYPE_RASL_R
See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator MFX_HEVC_NALU_TYPE_IDR_W_RADL
See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator MFX_HEVC_NALU_TYPE_IDR_N_LP
See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

enumerator MFX_HEVC_NALU_TYPE_CRA_NUT
See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

mfxPriority

enum mfxPriority
The mfxPriority enumerator describes the session priority.

Values:

enumerator MFX_PRIORITY_LOW
Low priority: the session operation halts when high priority tasks are executing and more than 75% of the CPU is being used for normal priority tasks.

enumerator MFX_PRIORITY_NORMAL
Normal priority: the session operation is halted if there are high priority tasks.

enumerator MFX_PRIORITY_HIGH
High priority: the session operation blocks other lower priority session operations.

mfxResourceType

enum mfxResourceType

Values:

enumerator MFX_RESOURCE_SYSTEM_SURFACE
System memory.

enumerator MFX_RESOURCE_VA_SURFACE
VA surface.

enumerator MFX_RESOURCE_VA_BUFFER
VA buffer.

enumerator MFX_RESOURCE_DX9_SURFACE
IDirect3DSurface9.

enumerator MFX_RESOURCE_DX11_TEXTURE
ID3D11Texture2D.

enumerator MFX_RESOURCE_DX12_RESOURCE
ID3D12Resource.

enumerator MFX_RESOURCE_DMA_RESOURCE
DMA resource.

enumerator MFX_RESOURCE_HDDLUNITE_REMOTE_MEMORY
HDDL Unite Remote memory handle.

mfxSkipMode

enum mfxSkipMode

The mfxSkipMode enumerator describes the decoder skip-mode options.

Values:

enumerator MFX_SKIPMODE_NOSKIP

enumerator MFX_SKIPMODE_MORE
Do not skip any frames.

enumerator MFX_SKIPMODE_LESS
Skip more frames.

mfxStatus

enum mfxStatus

Itemizes status codes returned by API functions.

Values:

enumerator MFX_ERR_NONE
No error.

enumerator MFX_ERR_UNKNOWN
Unknown error.

enumerator MFX_ERR_NULL_PTR
Null pointer.

enumerator MFX_ERR_UNSUPPORTED
Unsupported feature.

enumerator MFX_ERR_MEMORY_ALLOC
Failed to allocate memory.

enumerator MFX_ERR_NOT_ENOUGH_BUFFER
Insufficient buffer at input/output.

enumerator MFX_ERR_INVALID_HANDLE
Invalid handle.

enumerator MFX_ERR_LOCK_MEMORY
Failed to lock the memory block.

enumerator MFX_ERR_NOT_INITIALIZED
Member function called before initialization.

enumerator MFX_ERR_NOT_FOUND
The specified object is not found.

enumerator MFX_ERR_MORE_DATA
Expect more data at input.

enumerator MFX_ERR_MORE_SURFACE
Expect more surface at output.

enumerator MFX_ERR_ABORTED
Operation aborted.

enumerator MFX_ERR_DEVICE_LOST
Lose the hardware acceleration device.

enumerator MFX_ERR_INCOMPATIBLE_VIDEO_PARAM
Incompatible video parameters.

enumerator MFX_ERR_INVALID_VIDEO_PARAM
Invalid video parameters.

enumerator MFX_ERR_UNDEFINED_BEHAVIOR
Undefined behavior.

enumerator MFX_ERR_DEVICE_FAILED
Device operation failure.

enumerator MFX_ERR_MORE_BITSTREAM
Expect more bitstream buffers at output.

enumerator MFX_ERR_GPU_HANG
Device operation failure caused by GPU hang.

enumerator MFX_ERR_REALLOC_SURFACE
Bigger output surface required.

enumerator MFX_ERR_RESOURCE_MAPPED
Write access is already acquired and user requested another write access, or read access with MFX_MEMORY_NO_WAIT flag.

enumerator MFX_ERR_NOT_IMPLEMENTED
Feature or function not implemented.

enumerator MFX_WRN_IN_EXECUTION
The previous asynchronous operation is in execution.

enumerator MFX_WRN_DEVICE_BUSY
The hardware acceleration device is busy.

enumerator MFX_WRN_VIDEO_PARAM_CHANGED
The video parameters are changed during decoding.

enumerator MFX_WRN_PARTIAL_ACCELERATION
Software acceleration is used.

enumerator MFX_WRN_INCOMPATIBLE_VIDEO_PARAM
Incompatible video parameters.

enumerator MFX_WRN_VALUE_NOT_CHANGED
The value is saturated based on its valid range.

enumerator MFX_WRN_OUT_OF_RANGE
The value is out of valid range.

enumerator MFX_WRN_FILTER_SKIPPED
One of requested filters has been skipped.

enumerator MFX_ERR_NONE_PARTIAL_OUTPUT
Frame is not ready, but bitstream contains partial output.

enumerator MFX_TASK_DONE
Task has been completed.

enumerator MFX_TASK_WORKING
There is some more work to do.

enumerator MFX_TASK_BUSY
Task is waiting for resources.

enumerator MFX_ERR_MORE_DATA_SUBMIT_TASK
Return MFX_ERR_MORE_DATA but submit internal asynchronous task.

MirroringType

The MirroringType enumerator itemizes mirroring types.

enumerator MFX_MIRRORING_DISABLED

enumerator MFX_MIRRORING_HORIZONTAL

enumerator MFX_MIRRORING_VERTICAL

MPEG-2 Profiles

enumerator MFX_PROFILE_MPEG2_SIMPLE

enumerator MFX_PROFILE_MPEG2_MAIN

enumerator MFX_PROFILE_MPEG2_HIGH

Multi-view Video Coding Extension Profiles

enumerator MFX_PROFILE_AVC_MULTIVIEW_HIGH
Multi-view high profile.

enumerator MFX_PROFILE_AVC_STEREO_HIGH
Stereo high profile.

MVPrecision

The MVPrecision enumerator specifies the motion estimation precision

enumerator MFX_MVPRECISION_UNKNOWN

enumerator MFX_MVPRECISION_INTEGER

enumerator MFX_MVPRECISION_HALFPEL

enumerator MFX_MVPRECISION_QUARTERPEL

NominalRange

The NominalRange enumerator itemizes pixel's value nominal range.

enumerator MFX_NOMINALRANGE_UNKNOWN
Range is not defined.

enumerator MFX_NOMINALRANGE_0_255
Range is from 0 to 255.

enumerator MFX_NOMINALRANGE_16_235
Range is from 16 to 235.

PartialBitstreamOutput

The PartialBitstreamOutput enumerator indicates flags of partial bitstream output type.

enumerator MFX_PARTIAL_BITSTREAM_NONE
Do not use partial output

enumerator MFX_PARTIAL_BITSTREAM_SLICE
Partial bitstream output will be aligned to slice granularity

enumerator MFX_PARTIAL_BITSTREAM_BLOCK
Partial bitstream output will be aligned to user-defined block size granularity

enumerator MFX_PARTIAL_BITSTREAM_ANY
Partial bitstream output will be return any coded data available at the end of SyncOperation timeout

PayloadCtrlFlags

The PayloadCtrlFlags enumerator itemizes additional payload properties.

enumerator MFX_PAYLOAD_CTRL_SUFFIX
Insert this payload into HEVC Suffix SEI NAL-unit.

PicStruct

The PicStruct enumerator itemizes picture structure. Use bit-OR'ed values to specify the desired picture type.

enumerator MFX_PICSTRUCT_UNKNOWN
Unspecified or mixed progressive/interlaced/field pictures.

enumerator MFX_PICSTRUCT_PROGRESSIVE
Progressive picture.

enumerator MFX_PICSTRUCT_FIELD_TFF
Top field in first interlaced picture.

enumerator MFX_PICSTRUCT_FIELD_BFF
Bottom field in first interlaced picture.

enumerator MFX_PICSTRUCT_FIELD_REPEATED
First field repeated: pic_struct=5 or 6 in H.264.

enumerator MFX_PICSTRUCT_FRAME_DOUBLING
Double the frame for display: pic_struct=7 in H.264.

enumerator MFX_PICSTRUCT_FRAME_TRIPLING
Triple the frame for display: pic_struct=8 in H.264.

enumerator MFX_PICSTRUCT_FIELD_SINGLE
Single field in a picture.

enumerator MFX_PICSTRUCT_FIELD_TOP
Top field in a picture: pic_struct = 1 in H.265.

enumerator MFX_PICSTRUCT_FIELD_BOTTOM
Bottom field in a picture: pic_struct = 2 in H.265.

enumerator MFX_PICSTRUCT_FIELD_PAIRED_PREV
Paired with previous field: pic_struct = 9 or 10 in H.265.

enumerator MFX_PICSTRUCT_FIELD_PAIRED_NEXT
Paired with next field: pic_struct = 11 or 12 in H.265

PicType

The PicType enumerator itemizes picture type.

enumerator MFX_PICTYPE_UNKNOWN
Picture type is unknown.

enumerator MFX_PICTYPE_FRAME
Picture is a frame.

enumerator MFX_PICTYPE_TOPFIELD
Picture is a top field.

enumerator MFX_PICTYPE_BOTTOMFIELD
Picture is a bottom field.

PlatformCodeName

enumerator MFX_PLATFORM_UNKNOWN
Unknown platform.

enumerator MFX_PLATFORM_SANDYBRIDGE
Intel(r) microarchitecture code name Sandy Bridge.

enumerator MFX_PLATFORM_IVYBRIDGE
Intel(r) microarchitecture code name Ivy Bridge.

enumerator MFX_PLATFORM_HASWELL
Code name Haswell.

enumerator MFX_PLATFORM_BAYTRAIL
Code name Bay Trail.

enumerator MFX_PLATFORM_BROADWELL
Intel(r) microarchitecture code name Broadwell.

enumerator MFX_PLATFORM_CHERRYTRAIL
Code name Cherry Trail.

enumerator MFX_PLATFORM_SKYLAKE
Intel(r) microarchitecture code name Skylake.

enumerator MFX_PLATFORM_APOLLOLAKE
Code name Apollo Lake.

enumerator MFX_PLATFORM_KABYLAKE
Code name Kaby Lake.

enumerator MFX_PLATFORM_GEMINILAKE
Code name Gemini Lake.

enumerator MFX_PLATFORM_COFFEELAKE
Code name Coffee Lake.

enumerator MFX_PLATFORM_CANNONLAKE
Code name Cannon Lake.

enumerator MFX_PLATFORM_ICELAKE
Code name Ice Lake.

enumerator MFX_PLATFORM_JASPERLAKE
Code name Jasper Lake.

enumerator MFX_PLATFORM_ELKHARTLAKE
Code name Elkhart Lake.

enumerator MFX_PLATFORM_TIGERLAKE
Code name Tiger Lake.

PRefType

The PRefType enumerator itemizes models of reference list construction and DPB management when GopRefDist=1.

enumerator MFX_P_REF_DEFAULT
Allow encoder to decide.

enumerator MFX_P_REF_SIMPLE
Regular sliding window used for DPB removal process.

enumerator MFX_P_REF_PYRAMID
Let N be the max reference list's size. Encoder treats each N's frame as a 'strong' reference and the others as 'weak' references. The encoder uses a 'weak' reference only for prediction of the next frame and removes it from DPB immediately after use. 'Strong' references are removed from DPB by a sliding window.

Protected

The Protected enumerator describes the protection schemes.

enumerator MFX_PROTECTION_CENC_WV_CLASSIC
The protection scheme is based on the Widevine* DRM from Google*.

enumerator MFX_PROTECTION_CENC_WV_GOOGLE_DASH
The protection scheme is based on the Widevine* Modular DRM* from Google*.

RateControlMethod

The RateControlMethod enumerator itemizes bitrate control methods.

enumerator MFX_RATECONTROL_CBR
Use the constant bitrate control algorithm.

enumerator MFX_RATECONTROL_VBR
Use the variable bitrate control algorithm.

enumerator MFX_RATECONTROL_CQP
Use the constant quantization parameter algorithm.

enumerator MFX_RATECONTROL_AVBR
Use the average variable bitrate control algorithm.

enumerator MFX_RATECONTROL_LA
Use the VBR algorithm with look ahead. It is a special bitrate control mode in the AVC encoder that has been designed to improve encoding quality. It works by performing extensive analysis of several dozen frames before the actual encoding and as a side effect significantly increases encoding delay and memory consumption.

The only available rate control parameter in this mode is *mfxInfoMFX::TargetKbps*. Two other parameters, MaxKbps and InitialDelayInKB, are ignored. To control LA depth the application can use *mfxExtCodingOption2::LookAheadDepth* parameter.

This method is not HRD compliant.

enumerator MFX_RATECONTROL_ICQ
Use the Intelligent Constant Quality algorithm. This algorithm improves subjective video quality of encoded stream. Depending on content, it may or may not decrease objective video quality. Only one control parameter is used - quality factor, specified by *mfxInfoMFX::ICQQuality*.

enumerator MFX_RATECONTROL_VCM
Use the Video Conferencing Mode algorithm. This algorithm is similar to the VBR and uses the same set of parameters *mfxInfoMFX::InitialDelayInKB*, TargetKbpsandMaxKbps. It is tuned for IPPP GOP pattern and streams with strong temporal correlation between frames. It produces better objective and subjective video quality in these conditions than other bitrate control algorithms. It does not support interlaced content, B-frames and produced stream is not HRD compliant.

enumerator MFX_RATECONTROL_LA_ICQ
Use Intelligent Constant Quality algorithm with look ahead. Quality factor is specified by *mfxInfoMFX::ICQQuality*. To control LA depth the application can use *mfxExtCodingOption2::LookAheadDepth* parameter.

This method is not HRD compliant.

enumerator MFX_RATECONTROL_LA_HRD
MFX_RATECONTROL_LA_EXT has been removed

Use HRD compliant look ahead rate control algorithm.

enumerator MFX_RATECONTROL_QVBR
Use the variable bitrate control algorithm with constant quality. This algorithm trying to achieve the target subjective quality with the minimum number of bits, while the bitrate constraint and HRD compliance are satisfied. It uses the same set of parameters as VBR and quality factor specified by *mfxExtCodingOption3::QVBRQuality*.

ROImode

The ROImode enumerator itemizes QP adjustment mode for ROIs.

enumerator MFX_ROI_MODE_PRIORITY
Priority mode.

enumerator MFX_ROI_MODE_QP_DELTA
QP mode

enumerator MFX_ROI_MODE_QP_VALUE
Absolute QP

Rotation

The Rotation enumerator itemizes the JPEG rotation options.

enumerator MFX_ROTATION_0
No rotation.

enumerator MFX_ROTATION_90
90 degree rotation.

enumerator MFX_ROTATION_180
180 degree rotation.

enumerator MFX_ROTATION_270
270 degree rotation.

SampleAdaptiveOffset

The SampleAdaptiveOffset enumerator uses bit-ORed values to itemize corresponding HEVC encoding feature.

enumerator MFX_SAO_UNKNOWN
Use default value for platform/TargetUsage.

enumerator MFX_SAO_DISABLE
Disable SAO. If set during Init leads to SPS sample_adaptive_offset_enabled_flag = 0. If set during Runtime, leads to to slice_sao_luma_flag = 0 and slice_sao_chroma_flag = 0 for current frame.

enumerator MFX_SAO_ENABLE_LUMA
Enable SAO for luma (slice_sao_luma_flag = 1).

enumerator MFX_SAO_ENABLE_CHROMA
Enable SAO for chroma (slice_sao_chroma_flag = 1).

ScalingMode

The ScalingMode enumerator itemizes variants of scaling filter implementation.

enumerator MFX_SCALING_MODE_DEFAULT
Default scaling mode. The library selects the most appropriate scaling method.

enumerator MFX_SCALING_MODE_LOWPOWER
Low power scaling mode which is applicable for library implementations. The exact scaling algorithm is defined by the library.

enumerator MFX_SCALING_MODE_QUALITY
The best quality scaling mode

ScenarioInfo

The ScenarioInfo enumerator itemizes scenarios for the encoding session.

enumerator MFX_SCENARIO_UNKNOWN

enumerator MFX_SCENARIO_DISPLAY_REMOTING

enumerator MFX_SCENARIO_VIDEO_CONFERENCE

enumerator MFX_SCENARIO_ARCHIVE

enumerator MFX_SCENARIO_LIVE_STREAMING

enumerator MFX_SCENARIO_CAMERA_CAPTURE

enumerator MFX_SCENARIO_VIDEO_SURVEILLANCE

enumerator MFX_SCENARIO_GAME_STREAMING

enumerator MFX_SCENARIO_REMOTE_GAMING

SegmentFeature

The SegmentFeature enumerator indicates features enabled for the segment. These values are used with the mfxVP9SegmentParam::FeatureEnabled parameter.

enumerator MFX_VP9_SEGMENT_FEATURE_QINDEX
Quantization index delta.

enumerator MFX_VP9_SEGMENT_FEATURE_LOOP_FILTER
Loop filter level delta.

enumerator MFX_VP9_SEGMENT_FEATURE_REFERENCE
Reference frame.

enumerator MFX_VP9_SEGMENT_FEATURE_SKIP
Skip.

SegmentIdBlockSize

The SegmentIdBlockSize enumerator indicates the block size represented by each segment_id in segmentation map. These values are used with the mfxExtVP9Segmentation::SegmentIdBlockSize parameter.

enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_UNKNOWN
Unspecified block size.

enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_8x8
8x8 block size.

enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_16x16
16x16 block size.

enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_32x32
32x32 block size.

enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_64x64
64x64 block size.

SkipFrame

The SkipFrame enumerator is used to define usage of mfxEncodeCtrl::SkipFrame parameter.

enumerator MFX_SKIPFRAME_NO_SKIP
Frame skipping is disabled, *mfxEncodeCtrl::SkipFrame* is ignored.

enumerator MFX_SKIPFRAME_INSERT_DUMMY
Skipping is allowed, when *mfxEncodeCtrl::SkipFrame* is set encoder inserts into bitstream frame where all macroblocks are encoded as skipped. Only non-reference P- and B-frames can be skipped. If GopRefDist = 1 and *mfxEncodeCtrl::SkipFrame* is set for reference P-frame, it will be encoded as non-reference.

enumerator MFX_SKIPFRAME_INSERT_NOTHING
Similar to MFX_SKIPFRAME_INSERT_DUMMY, but when *mfxEncodeCtrl::SkipFrame* is set encoder inserts nothing into bitstream.

enumerator MFX_SKIPFRAME_BRC_ONLY
mfxEncodeCtrl::SkipFrame indicates number of missed frames before the current frame. Affects only BRC, current frame will be encoded as usual.

TargetUsage

The TargetUsage enumerator itemizes a range of numbers from MFX_TARGETUSAGE_1, best quality, to MFX_TARGETUSAGE_7, best speed. It indicates trade-offs between quality and speed. The application can use any number in the range. The actual number of supported target usages depends on implementation. If the specified target usage is not supported, the encoder will use the closest supported value.

enumerator MFX_TARGETUSAGE_1
Best quality

enumerator MFX_TARGETUSAGE_2

enumerator MFX_TARGETUSAGE_3

enumerator MFX_TARGETUSAGE_4
Balanced quality and speed.

enumerator MFX_TARGETUSAGE_5

enumerator MFX_TARGETUSAGE_6

enumerator MFX_TARGETUSAGE_7
Best speed

enumerator MFX_TARGETUSAGE_UNKNOWN
Unspecified target usage.

enumerator MFX_TARGETUSAGE_BEST_QUALITY
Best quality.

enumerator MFX_TARGETUSAGE_BALANCED
Balanced quality and speed.

enumerator MFX_TARGETUSAGE_BEST_SPEED
Best speed.

TelecinePattern

The TelecinePattern enumerator itemizes telecine patterns.

enumerator MFX_TELECINE_PATTERN_32
3:2 telecine.

enumerator MFX_TELECINE_PATTERN_2332
2:3:3:2 telecine.

enumerator MFX_TELECINE_PATTERN_FRAME_REPEAT
One frame repeat telecine.

enumerator MFX_TELECINE_PATTERN_41
4:1 telecine.

enumerator MFX_TELECINE_POSITION_PROVIDED
User must provide position inside a sequence of 5 frames where the artifacts start.

TimeStampCalc

The TimeStampCalc enumerator itemizes time-stamp calculation methods.

enumerator MFX_TIMESTAMPCALC_UNKNOWN
The time stamp calculation is based on the input frame rate if time stamp is not explicitly specified.

enumerator MFX_TIMESTAMPCALC_TELECINE
Adjust time stamp to 29.97fps on 24fps progressively encoded sequences if telecine attributes are available in the bitstream and time stamp is not explicitly specified. The input frame rate must be specified.

TransferMatrix

The TransferMatrix enumerator itemizes color transfer matrices.

enumerator MFX_TRANSFERMATRIX_UNKNOWN
Transfer matrix is not specified

enumerator MFX_TRANSFERMATRIX_BT709
Transfer matrix from ITU-R BT.709 standard.

enumerator MFX_TRANSFERMATRIX_BT601
Transfer matrix from ITU-R BT.601 standard.

TrellisControl

The TrellisControl enumerator is used to control trellis quantization in AVC encoder. The application can turn it on or off for any combination of I, P, and B frames by combining different enumerator values. For example, MFX_TRELLIS_I | MFX_TRELLIS_B turns it on for I and B frames.

enumerator MFX_TRELLIS_UNKNOWN
Default value, it is up to the encoder to turn trellis quantization on or off.

enumerator MFX_TRELLIS_OFF
Turn trellis quantization off for all frame types.

enumerator MFX_TRELLIS_I
Turn trellis quantization on for I-frames.

enumerator MFX_TRELLIS_P
Turn trellis quantization on for P-frames.

enumerator MFX_TRELLIS_B
Turn trellis quantization on for B-frames.

VP9ReferenceFrame

The VP9ReferenceFrame enumerator itemizes reference frame type by the mfxVP9SegmentParam::ReferenceFrame parameter.

enumerator MFX_VP9_REF_INTRA
Intra.

enumerator MFX_VP9_REF_LAST
Last.

enumerator MFX_VP9_REF_GOLDEN
Golden.

enumerator MFX_VP9_REF_ALTREF
Alternative reference.

VPPFieldProcessingMode

The VPPFieldProcessingMode enumerator is used to control VPP field processing algorithm.

enumerator MFX_VPP_COPY_FRAME
Copy the whole frame.

enumerator MFX_VPP_COPY_FIELD
Copy only one field.

enumerator MFX_VPP_SWAP_FIELDS
Swap top and bottom fields.

WeightedPred

The WeightedPred enumerator itemizes weighted prediction modes.

enumerator MFX_WEIGHTED_PRED_UNKNOWN
Allow encoder to decide.

enumerator MFX_WEIGHTED_PRED_DEFAULT
Use default weighted prediction.

enumerator MFX_WEIGHTED_PRED_EXPLICIT
Use explicit weighted prediction.

enumerator MFX_WEIGHTED_PRED_IMPLICIT
Use implicit weighted prediction (for B-frames only).

FilmGrainFlags

The FilmGrainFlags enumerator itemizes flags in AV1 film grain parameters.

enumerator MFX_FILM_GRAIN_NO
Film grain isn't added to this frame.

enumerator MFX_FILM_GRAIN_APPLY
Film grain is added to this frame.

enumerator MFX_FILM_GRAIN_UPDATE
New set of film grain parameters is sent for this frame.

enumerator MFX_FILM_GRAIN_CHROMA_SCALING_FROM_LUMA
Chroma scaling is inferred from luma scaling.

enumerator MFX_FILM_GRAIN_OVERLAP
Overlap between film grain blocks is applied.

enumerator MFX_FILM_GRAIN_CLIP_TO_RESTRICTED_RANGE
Clipping to the restricted (studio) range is applied after adding the film grain.

11.4.4 Define Reference

API

MFX_DECODERDESCRIPTION_VERSION

MFX_DEVICEDESCRIPTION_VERSION

MFX_ENCODERDESCRIPTION_VERSION

MFX_FRAMESURFACE1_VERSION

MFX_FRAMESURFACEINTERFACE_VERSION

MFX_IMPLDESCRIPTION_VERSION

MFX_LEGACY_VERSION
The corresponding version of the Intel(r) Media SDK legacy API that is used as a basis for the current API.

MFX_STRUCT_VERSION(*MAJOR*, *MINOR*)

MFX_VARIANT_VERSION

MFX_VERSION

MFX_VERSION_MAJOR

MFX_VERSION_MINOR

MFX_VERSION_NEXT

MFX_VPPDESCRIPTION_VERSION

MFX_SURFACEARRAY_VERSION

11.4.5 Type Reference

Basic Types

typedef char **mfxChar**
UTF-8 byte.

typedef float **mfxF32**
Single-precision floating point, 32 bit type.

typedef double **mfxF64**
Double-precision floating point, 64 bit type.

typedef void ***mfxHDL**
Handle type.

typedef char **mfxI8**
Signed integer, 8 bit type.

typedef short **mfxI16**
Signed integer, 16 bit type.

typedef int **mfxI32**
Signed integer, 32 bit type.

typedef __INT64 **mfxI64**
Signed integer, 64 bit type.

typedef int **mfxL32**
Signed integer, 32 bit type.

typedef *mfxHDL* **mfxMemId**
Memory ID type.

typedef void ***mfxThreadTask**
Thread task type.

typedef unsigned char **mfxU8**
Unsigned integer, 8 bit type.

typedef unsigned short **mfxU16**
Unsigned integer, 16 bit type.

typedef unsigned int **mfxU32**
Unsigned integer, 32 bit type.

typedef __UINT64 **mfxU64**
Unsigned integer, 64 bit type.

typedef unsigned int **mfxUL32**
Unsigned integer, 32 bit type.

Typedefs

typedef struct _mfxConfig ***mfxConfig**
Config handle.

typedef struct _mfxLoader ***mfxLoader**
Loader handle.

typedef struct _mfxSession ***mfxSession**
Session handle.

typedef struct _mfxSyncPoint ***mfxSyncPoint**
Synchronization point object handle.

11.4.6 Dispatcher API

Use the Dispatcher API to load and execute the appropriate library implementation and get capabilities for the implementations available on the platform.

Dispatcher API Function Reference

API

MFXCreateConfig

mfxConfig **MFXCreateConfig**(*mfxLoader loader*)

Creates dispatcher configuration.

Creates the dispatcher internal configuration, which is used to filter out available implementations. This configuration is used to walk through selected implementations to gather more details and select the appropriate implementation to load. The loader object remembers all created mfxConfig objects and destroys them during the mfxUnload function call.

Multiple configurations per single mfxLoader object are possible.

Usage example:

```
mfxLoader loader = MFXLoad();
mfxConfig cfg = MFXCreateConfig(loader);
MFXCreateSession(loader,0,&session);
```

Return Config handle or NULL pointer is failed.

Parameters

- `[in] loader`: Loader handle.

MFXCreateSession

mfxStatus **MFXCreateSession**(*mfxLoader loader*, *mfxU32 i*, *mfxSession *session*)

Loads and initializes the implementation.

```
mfxLoader loader = MFXLoad();
int i=0;
while(1) {
   mfxImplDescription *idesc;
   MFXEnumImplementations(loader, i, MFX_IMPLCAPS_IMPLDESCSTRUCTURE, (mfxHDL*)&
↪idesc);
   if(is_good(idesc)) {
       MFXCreateSession(loader, i,&session);
```

(continues on next page)

(continued from previous page)

```
        // ...
        MFXDispReleaseImplDescription(loader, idesc);
    }
    else
    {
        MFXDispReleaseImplDescription(loader, idesc);
        break;
    }
}
```

Return MFX_ERR_NONE The function completed successfully. The session contains a pointer to the session handle. MFX_ERR_NULL_PTR If loader is NULL.

MFX_ERR_NULL_PTR If session is NULL.

MFX_ERR_NOT_FOUND Provided index is out of possible range.

Parameters

- `[in] loader`: Loader handle.
- `[in] i`: Index of the implementation.
- `[out] session`: Pointer to the session handle.

MFXDispReleaseImplDescription

mfxStatus **MFXDispReleaseImplDescription** (*mfxLoader loader*, *mfxHDL hdl*)

Destroys handle allocated by the MFXQueryImplsCapabilities function.

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR If loader is NULL.

MFX_ERR_INVALID_HANDLE Provided hdl handle is not associated with this loader.

Parameters

- `[in] loader`: Loader handle.
- `[in] hdl`: Handle to destroy. Can be equal to NULL.

MFXEnumImplementations

mfxStatus **MFXEnumImplementations** (*mfxLoader loader*, *mfxU32 i*, *mfxImplCapsDeliveryFormat format*, *mfxHDL *idesc*)

Iterates over filtered out implementations to gather their details. This function allocates memory to store *mfxImplDescription* structure instance. Use the MFXDispReleaseImplDescription function to free memory allocated to the *mfxImplDescription* structure.

Return MFX_ERR_NONE The function completed successfully. The idesc contains valid information. MFX_ERR_NULL_PTR If loader is NULL.

MFX_ERR_NULL_PTR If idesc is NULL.

MFX_ERR_NOT_FOUND Provided index is out of possible range.

MFX_ERR_UNSUPPORTED If requested format is not supported.

Parameters

- `[in] loader`: Loader handle.
- `[in] i`: Index of the implementation.
- `[in] format`: Format in which capabilities need to be delivered. See the mfxImplCapsDeliveryFormat enumerator for more details.
- `[out] idesc`: Pointer to the *mfxImplDescription* structure.

MFXLoad

mfxLoader **MFXLoad**()
Creates the loader.

Return Loader Loader handle or NULL if failed.

MFXSetConfigFilterProperty

mfxStatus **MFXSetConfigFilterProperty**(*mfxConfig config*, **const** *mfxU8 *name*, *mfxVariant value*)
Adds additional filter properties (any fields of the *mfxImplDescription* structure) to the configuration of the loader object. One mfxConfig properties can hold only single filter property.

Simple usage example:

```
mfxLoader loader = MFXLoad();
mfxConfig cfg = MFXCreateConfig(loader);
mfxVariant ImplValue;
ImplValue.Type = MFX_VARIANT_TYPE_U32;
ImplValue.Data.U32 = MFX_IMPL_TYPE_HARDWARE;
MFXSetConfigFilterProperty(cfg,"mfxImplDescription.Impl",ImplValue);
MFXCreateSession(loader,0,&session);
```

Note Each new call with the same parameter name will overwrite the previously set value. This may invalidate other properties.

Note Each new call with another parameter name will delete the previous property and create a new property based on new name's value.

Usage example with two sessions (multiple loaders):

```
// Create session with software based implementation
mfxLoader loader1 = MFXLoad();
mfxConfig cfg1 = MFXCreateConfig(loader1);
mfxVariant ImplValueSW;
ImplValueSW.Type = MFX_VARIANT_TYPE_U32;
ImplValueSW.Data.U32 = MFX_IMPL_TYPE_SOFTWARE;
MFXSetConfigFilterProperty(cfg1,"mfxImplDescription.Impl",ImplValueSW);
MFXCreateSession(loader1,0,&sessionSW);

// Create session with hardware based implementation
mfxLoader loader2 = MFXLoad();
mfxConfig cfg2 = MFXCreateConfig(loader2);
mfxVariant ImplValueHW;
ImplValueHW.Type = MFX_VARIANT_TYPE_U32;
ImplValueHW.Data.U32 = MFX_IMPL_TYPE_HARDWARE;
```

(continues on next page)

(continued from previous page)

```
MFXSetConfigFilterProperty(cfg2,"mfxImplDescription.Impl",ImplValueHW);
MFXCreateSession(loader2,0,&sessionHW);

// use both sessionSW and sessionHW
// ...
// Close everything
MFXClose(sessionSW);
MFXClose(sessionHW);
MFXUnload(loader1); // cfg1 will be destroyed here.
MFXUnload(loader2); // cfg2 will be destroyed here.
```

Usage example with two decoders (multiple config objects):

```
mfxLoader loader = MFXLoad();

mfxConfig cfg1 = MFXCreateConfig(loader);
mfxVariant ImplValue;
val.Type = MFX_VARIANT_TYPE_U32;
val.Data.U32 = MFX_CODEC_AVC;
MFXSetConfigFilterProperty(cfg1,"mfxImplDescription.mfxDecoderDescription.decoder.
↪CodecID",ImplValue);

mfxConfig cfg2 = MFXCreateConfig(loader);
mfxVariant ImplValue;
val.Type = MFX_VARIANT_TYPE_U32;
val.Data.U32 = MFX_CODEC_HEVC;
MFXSetConfigFilterProperty(cfg2,"mfxImplDescription.mfxDecoderDescription.decoder.
↪CodecID",ImplValue);

MFXCreateSession(loader,0,&sessionAVC);
MFXCreateSession(loader,0,&sessionHEVC);
```

Return MFX_ERR_NONE The function completed successfully. MFX_ERR_NULL_PTR If config is NULL. MFX_ERR_NULL_PTR If name is NULL.

MFX_ERR_NOT_FOUND If name contains unknown parameter name. MFX_ERR_UNSUPPORTED If value data type does not equal the parameter with provided name.

Parameters

- `[in] config`: Config handle.
- `[in] name`: Name of the parameter (see *mfxImplDescription* structure and example).
- `[in] value`: Value of the parameter.

MFXUnload

void **`MFXUnload`**(*mfxLoader loader*)

Destroys the dispatcher.

Parameters

- `[in] loader`: Loader handle.

Dispatcher API Structure Reference

API

mfxDecoderDescription

struct mfxDecoderDescription
The *mfxDecoderDescription* structure represents the description of a decoder.

Public Members

mfxStructVersion **Version**
Version of the structure.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumCodecs**
Number of supported decoders.

struct *mfxDecoderDescription*::*decoder* ***Codecs**
Pointer to the array of decoders.

struct decoder
This structure represents the decoder description.

Public Members

mfxU32 **CodecID**
Decoder ID in FourCC format.

mfxU16 **reserved**[8]
Reserved for future use.

mfxU16 **MaxcodecLevel**
Maximum supported codec level. See the CodecProfile enumerator for possible values.

mfxU16 **NumProfiles**
Number of supported profiles.

struct *mfxDecoderDescription*::*decoder*::*decprofile* ***Profiles**
Pointer to the array of profiles supported by the codec.

struct decprofile
This structure represents the codec profile description.

Public Members

mfxU32 **Profile**
Profile ID. See the CodecProfile enumerator for possible values.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumMemTypes**
Number of supported memory types.

struct *mfxDecoderDescription*::*decoder*::*decprofile*::*decmemdesc* ***MemDesc**
Pointer to the array of memory types.

struct decmemdesc
This structure represents the underlying details of the memory type.

Public Members

mfxResourceType **MemHandleType**
Memory handle type.

mfxRange32U **Width**
Range of supported image widths.

mfxRange32U **Height**
Range of supported image heights.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumColorFormats**
Number of supported output color formats.

mfxU32 ***ColorFormats**
Pointer to the array of supported output color formats (in FOURCC).

mfxDeviceDescription

struct mfxDeviceDescription
This structure represents device description.

Public Members

mfxStructVersion **Version**
Version of the structure.

mfxU16 **reserved**[7]
reserved for future use.

mfxChar **DeviceID**[**MFX_STRFIELD_LEN**]
Null terminated string with device ID.

mfxU16 **NumSubDevices**
Number of available uniform sub-devices. Pure software implementation can report 0.

struct *mfxDeviceDescription*::*subdevices* ***SubDevices**
Pointer to the array of available sub-devices.

struct subdevices
This structure represents sub-device description.

Public Members

mfxU32 **Index**
Index of the sub-device, started from 0 and increased by 1.

mfxChar **SubDeviceID**[**MFX_STRFIELD_LEN**]
Null terminated string with unique sub-device ID, mapped to the system ID.

mfxU32 **reserved**[7]
reserved for future use.

mfxEncoderDescription

struct mfxEncoderDescription
This structure represents an encoder description.

Public Members

mfxStructVersion **Version**
Version of the structure.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumCodecs**
Number of supported encoders.

struct *mfxEncoderDescription*::*encoder* ***Codecs**
Pointer to the array of encoders.

struct encoder
This structure represents encoder description.

Public Members

mfxU32 **CodecID**
Encoder ID in FourCC format.

mfxU16 **MaxcodecLevel**
Maximum supported codec level. See the CodecProfile enumerator for possible values.

mfxU16 **BiDirectionalPrediction**
Indicates B-frames support.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumProfiles**
Number of supported profiles.

struct *mfxEncoderDescription*::*encoder*::*encprofile* ***Profiles**
Pointer to the array of profiles supported by the codec.

struct encprofile
This structure represents the codec profile description.

Public Members

mfxU32 **Profile**
Profile ID. See the CodecProfile enumerator for possible values.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumMemTypes**
Number of supported memory types.

struct *mfxEncoderDescription*::*encoder*::*encprofile*::*encmemdesc* ***MemDesc**
Pointer to the array of memory types.

struct encmemdesc
This structure represents the underlying details of the memory type.

Public Members

mfxResourceType **MemHandleType**
Memory handle type.

mfxRange32U **Width**
Range of supported image widths.

mfxRange32U **Height**
Range of supported image heights.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumColorFormats**
Number of supported input color formats.

mfxU32 ***ColorFormats**
Pointer to the array of supported input color formats (in FOURCC).

mfxImplDescription

struct mfxImplDescription
This structure represents the implementation description.

Public Members

mfxStructVersion **Version**
Version of the structure.

mfxImplType **Impl**
Impl type: software/hardware.

mfxAccelerationMode **AccelerationMode**
Default Hardware acceleration stack to use. OS dependent parameter. Use VA for Linux* and DX* for Windows*.

mfxVersion **ApiVersion**
Supported API version.

mfxChar **ImplName**[**MFX_IMPL_NAME_LEN**]
Null-terminated string with implementation name given by vendor.

mfxChar **License**[**MFX_STRFIELD_LEN**]
Null-terminated string with license name of the implementation.

mfxChar **Keywords**[**MFX_STRFIELD_LEN**]
Null-terminated string with comma-separated list of keywords specific to this implementation that dispatcher can search for.

mfxU32 **VendorID**
Standard vendor ID 0x8086 - Intel.

mfxU32 **VendorImplID**
Vendor specific number with given implementation ID.

mfxDeviceDescription **Dev**
Supported device.

mfxDecoderDescription **Dec**
Decoder configuration.

mfxEncoderDescription **Enc**
Encoder configuration.

mfxVPPDescription **VPP**
VPP configuration.

mfxAccelerationModeDescription **AccelerationModeDescription**
Supported acceleration modes.

mfxU32 **reserved**[12]
Reserved for future use.

mfxU32 **NumExtParam**
Number of extension buffers. Reserved for future use. Must be 0.

mfxExtBuffer ****ExtParam**
Array of extension buffers.

mfxU64 **Reserved2**
Reserved for future use.

union *mfxImplDescription*::**[anonymous] ExtParams**
Extension buffers. Reserved for future.

mfxVariant

struct mfxVariant
The mfxVariantType enumerator data types for mfxVarianf type.

Public Members

mfxStructVersion **Version**
Version of the structure.

mfxVariantType **Type**
Value type.

union *mfxVariant*::*data* **Data**
Value data member.

union data
Value data holder.

Public Members

mfxU8 **U8**
mfxU8 data.

mfxI8 **I8**
mfxI8 data.

mfxU16 **U16**
mfxU16 data.

mfxI16 **I16**
mfxI16 data.

mfxU32 **U32**
mfxU32 data.

mfxI32 **I32**
mfxI32 data.

mfxU64 **U64**
mfxU64 data.

mfxI64 **I64**
mfxI64 data.

mfxF32 **F32**
mfxF32 data.

mfxF64 **F64**
mfxF64 data.

mfxHDL **Ptr**
Pointer.

enum mfxVariantType
The mfxVariantType enumerator data types for mfxVarianf type.

Values:

enumerator MFX_VARIANT_TYPE_UNSET
Undefined type.

enumerator MFX_VARIANT_TYPE_U8
8-bit unsigned integer.

enumerator MFX_VARIANT_TYPE_I8
8-bit signed integer.

enumerator MFX_VARIANT_TYPE_U16
16-bit unsigned integer.

enumerator MFX_VARIANT_TYPE_I16
16-bit signed integer.

enumerator MFX_VARIANT_TYPE_U32
32-bit unsigned integer.

enumerator MFX_VARIANT_TYPE_I32
32-bit signed integer.

enumerator MFX_VARIANT_TYPE_U64
64-bit unsigned integer.

enumerator MFX_VARIANT_TYPE_I64
64-bit signed integer.

enumerator MFX_VARIANT_TYPE_F32
32-bit single precision floating point.

enumerator MFX_VARIANT_TYPE_F64
64-bit double precision floating point.

enumerator MFX_VARIANT_TYPE_PTR
Generic type pointer.

mfxVPPDescription

struct mfxVPPDescription
This structure represents VPP description.

Public Members

mfxStructVersion **Version**
Version of the structure.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumFilters**
Number of supported VPP filters.

struct *mfxVPPDescription*::*filter* ***Filters**
Pointer to the array of supported filters.

struct filter
This structure represents the VPP filters description.

Public Members

mfxU32 **FilterFourCC**
Filter ID in FourCC format.

mfxU16 **MaxDelayInFrames**
Introduced output delay in frames.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumMemTypes**
Number of supported memory types.

struct *mfxVPPDescription*::*filter*::*memdesc* ***MemDesc**
Pointer to the array of memory types.

struct memdesc
This structure represents the underlying details of the memory type.

Public Members

mfxResourceType **MemHandleType**
Memory handle type.

mfxRange32U **Width**
Range of supported image widths.

mfxRange32U **Height**
Range of supported image heights.

mfxU16 **reserved**[7]
Reserved for future use.

mfxU16 **NumInFormats**
Number of supported input color formats.

struct *mfxVPPDescription*::*filter*::*memdesc*::*format* ***Formats**
Pointer to the array of supported formats.

struct format
This structure represents the input color format description.

Public Members

mfxU32 **InFormat**
Input color in FourCC format.

mfxU16 **reserved**[5]
Reserved for future use.

mfxU16 **NumOutFormat**
Number of supported output color formats.

mfxU32 ***OutFormats**
Pointer to the array of supported output color formats (in FOURCC).

mfxAccelerationModeDescription

struct mfxAccelerationModeDescription
This structure represents acceleration modes description.

Public Members

mfxStructVersion **Version**
Version of the structure.

mfxU16 **reserved**[2]
reserved for future use.

mfxU16 **NumAccelerationModes**
Number of supported acceleration modes.

mfxAccelerationMode ***Mode**
Pointer to the array of supported acceleration modes.

Dispatcher API Enumeration Reference

API

- *mfxAccelerationMode*
- *mfxImplType*

mfxAccelerationMode

enum mfxAccelerationMode
This enum itemizes hardware acceleration stack to use.

Values:

enumerator MFX_ACCEL_MODE_NA
Hardware acceleration is not applicable.

enumerator MFX_ACCEL_MODE_VIA_D3D9
Hardware acceleration goes through the Microsoft* Direct3D9* infrastructure.

enumerator MFX_ACCEL_MODE_VIA_D3D11
Hardware acceleration goes through the Microsoft* Direct3D11* infrastructure.

enumerator MFX_ACCEL_MODE_VIA_VAAPI
Hardware acceleration goes through the Linux* VA-API infrastructure.

enumerator MFX_ACCEL_MODE_VIA_HDDLUNITE
Hardware acceleration goes through the HDDL* Unite*.

mfxImplType

enum mfxImplType
This enum itemizes implementation type.

Values:

enumerator MFX_IMPL_TYPE_SOFTWARE
Pure Software Implementation.

enumerator MFX_IMPL_TYPE_HARDWARE
Hardware Accelerated Implementation.

Dispatcher API Define Reference

API

- *MFX_IMPL_NAME_LEN*
- *MFX_STRFIELD_LEN*

MFX_IMPL_NAME_LEN

MFX_IMPL_NAME_LEN
Maximum allowed length of the implementation name.

MFX_STRFIELD_LEN

MFX_STRFIELD_LEN
Maximum allowed length of the implementation name.

11.5 oneVPL API Versioning

oneVPL is the successor to Intel® Media Software Development Kit. oneVPL API versioning starts from 2.0. There is a correspondent version of Intel® Media Software Development Kit API which is used as a basis for oneVPL and defined as the `MFX_LEGACY_VERSION` macro.

Experimental APIs in oneVPL are protected with the following macro:

```
#if (MFX_VERSION >= MFX_VERSION_NEXT)
```

To use the API, define the `MFX_VERSION_USE_LATEST` macro.

11.6 Appendices

11.6.1 oneVPL for Intel® Media Software Development Kit Users

oneVPL is source compatible with Intel® Media Software Development Kit. Applications can use Intel® Media Software Development Kit to target older hardware and oneVPL to target everything else. Some obsolete features of Intel® Media Software Development Kit have been omitted from oneVPL.

oneVPL Ease of Use Enhancements

oneVPL provides improved ease of use compared to Intel® Media Software Development Kit. Ease of use enhancements include the following:

- Smart dispatcher with discovery of implementation capabilities. See *oneVPL Session* for more details.
- Simplified decoder initialization. See *Decoding Procedures* for more details.
- New memory management and components (session) interoperability. See *Internal Memory Management* and *Decoding Procedures* for more details.

New APIs in oneVPL

oneVPL introduces new functions that are not available in Intel® Media Software Development Kit.

New oneVPL dispatcher functions:

- *MFXLoad()*
- *MFXUnload()*
- *MFXCreateConfig()*
- *MFXSetConfigFilterProperty()*
- *MFXEnumImplementations()*
- *MFXCreateSession()*
- *MFXDispReleaseImplDescription()*

New oneVPL memory management functions:

- *MFXMemory_GetSurfaceForVPP()*
- *MFXMemory_GetSurfaceForVPPOut()*
- *MFXMemory_GetSurfaceForEncode()*
- *MFXMemory_GetSurfaceForDecode()*

New oneVPL implementation capabilities retrieval functions:

- *MFXQueryImplsDescription()*
- *MFXReleaseImplDescription()*

New oneVPL session initialization:

- *MFXInitialize()*

Intel® Media Software Development Kit Feature Removals

The following Intel® Media Software Development Kit features are considered obsolete and are not included in oneVPL:

- **Audio support.** oneVPL is intended for video processing. Audio APIs that duplicate functionality from other audio libraries such as Sound Open Firmware have been removed.
- **ENC and PAK interfaces.** Part of the Flexible Encode Infrastructure (FEI) and plugin interfaces which provide additional control over the encoding process for AVC and HEVC encoders. This feature was removed because it is not widely used by customers.
- **User plugins architecture.** oneVPL enables robust video acceleration through API implementations of many different video processing frameworks. Support of a SDK user plugin framework is obsolete.
- **External buffer memory management.** A set of callback functions to replace internal memory allocation is obsolete.
- **Video Processing extended runtime functionality.** Video processing function MFXVideoVPP_RunFrameVPPAsyncEx is used for plugins only and is obsolete.
- **External threading.** The new threading model makes the MFXDoWork function obsolete.
- **Multi-frame encode.** A set of external buffers to combine several frames into one encoding call. This feature was removed because it is device specific and not commonly used.
- **Surface Type Neutral Transcoding.** Opaque memory support is removed and replaced with internal memory allocation concept.

Intel® Media Software Development Kit API Removals

The following Intel® Media Software Development Kit functions are not included in oneVPL:

- **Audio related functions**
  - MFXAudioCORE_SyncOperation()
  - MFXAudioDECODE_Close()
  - MFXAudioDECODE_DecodeFrameAsync()
  - MFXAudioDECODE_DecodeHeader()
  - MFXAudioDECODE_GetAudioParam()
  - MFXAudioDECODE_Init()
  - MFXAudioDECODE_Query()
  - MFXAudioDECODE_QueryIOSize()
  - MFXAudioDECODE_Reset()
  - MFXAudioENCODE_Close()
  - MFXAudioENCODE_EncodeFrameAsync()
  - MFXAudioENCODE_GetAudioParam()
  - MFXAudioENCODE_Init()
  - MFXAudioENCODE_Query()
  - MFXAudioENCODE_QueryIOSize()
  - MFXAudioENCODE_Reset()

- **Flexible encode infrastructure functions**
  - MFXVideoENC_Close()
  - MFXVideoENC_GetVideoParam()
  - MFXVideoENC_Init()
  - MFXVideoENC_ProcessFrameAsync()
  - MFXVideoENC_Query()
  - MFXVideoENC_QueryIOSurf()
  - MFXVideoENC_Reset()
  - MFXVideoPAK_Close()
  - MFXVideoPAK_GetVideoParam()
  - MFXVideoPAK_Init()
  - MFXVideoPAK_ProcessFrameAsync()
  - MFXVideoPAK_Query()
  - MFXVideoPAK_QueryIOSurf()
  - MFXVideoPAK_Reset()
- **User plugin functions**
  - MFXAudioUSER_ProcessFrameAsync()
  - MFXAudioUSER_Register()
  - MFXAudioUSER_Unregister()
  - MFXVideoUSER_GetPlugin()
  - MFXVideoUSER_ProcessFrameAsync()
  - MFXVideoUSER_Register()
  - MFXVideoUSER_Unregister()
  - MFXVideoUSER_Load()
  - MFXVideoUSER_LoadByPath()
  - MFXVideoUSER_UnLoad()
  - MFXDoWork()
- **Memory functions**
  - MFXVideoCORE_SetBufferAllocator()
- **Video processing functions**
  - MFXVideoVPP_RunFrameVPPAsyncEx()
- **Memory type and IOPattern enumerations**
  - MFX_IOPATTERN_IN_OPAQUE_MEMORY
  - MFX_IOPATTERN_OUT_OPAQUE_MEMORY
  - MFX_MEMTYPE_OPAQUE_FRAME

Important: Corresponding extension buffers are also removed.

The following behaviors occur when attempting to use a Intel® Media Software Development Kit API that is not supported by oneVPL:

- Code compiled with the oneVPL API headers will generate a compile and/or link error when attempting to use a removed API.
- Code previously compiled with Intel® Media Software Development Kit and executed using a oneVPL runtime will generate an *MFX_ERR_NOT_IMPLEMENTED* error when calling a removed function.

11.6.2 Configuration Parameter Constraints

The *mfxFrameInfo* structure is used by both the *mfxVideoParam* structure during oneVPL class initialization and the *mfxFrameSurface1* structure during the actual oneVPL class operation. The parameter constraints described in the following tables apply.

DECODE, ENCODE, and VPP Constraints

The *DECODE, ENCODE, and VPP Constraints table* lists parameter constraints common to *DECODE*, *ENCODE*, and *VPP*.

Table 11: DECODE, ENCODE, and VPP Constraints

Parameters	Use During Initialization	Use During Operation
FourCC	Any valid value.	The value must be the same as the initialization value. The only exception is *VPP* in composition mode, where in some cases it is allowed to mix RGB and NV12 surfaces. See *mfxExtVPPComposite* for more details.
ChromaFormat	Any valid value.	The value must be the same as the initialization value.

DECODE Constraints

The *DECODE Constraints table* lists *DECODE* parameter constraints.

Table 12: DECODE Constraints

Parameters	Use During Initialization	Use During Operation
Width, Height	Aligned frame size.	The values must be the equal to or larger than the initialization values.
CropX, CropY CropW, CropH	Ignored.	*DECODE* output. The cropping values are per-frame based.
AspectRatioW, AspectRatioH	Any valid values or unspecified (zero); if unspecified, values from the input bitstream will be used. See note below the table.	DECODE output.
FrameRateExtN, FrameRateExtD	If unspecified, values from the input bitstream will be used. See note below the table.	DECODE output.
PicStruct	Ignored.	DECODE output.

Note: If the application explicitly sets FrameRateExtN/FrameRateExtD or AspectRatioW/AspectRatioH during initialization, then the decoder will use these values during decoding regardless of the values from bitstream and does not update them on new SPS. If the application sets them to 0, then the decoder uses values from the stream and updates them on each SPS.

ENCODE Constraints

The *ENCODE Constraints table* lists *ENCODE* parameter constraints.

Table 13: ENCODE Constraints

Parameters	Use During Initialization	Use During Operation
Width, Height	Encoded frame size.	The values must be the equal to or larger than the initialization values.
CropX, CropY CropW, CropH	H.264: Cropped frame size MPEG-2: CropW and CropH Specify the real width and height (may be unaligned) of the coded frames. CropX and CropY must be zero.	Ignored.
AspectRatioW, AspectRatioH	Any valid values.	Ignored.
FrameRateExtN, FrameRateExtD	Any valid values.	Ignored.
PicStruct	*MFX_PICSTRUCT_UNKNOWN* *MFX_PICSTRUCT_PROGRESSIVE* *MFX_PICSTRUCT_FIELD_TFF* *MFX_PICSTRUCT_FIELD_BFF*	The base value must be the same as the initialization value unless *MFX_PICSTRUCT_UNKNOWN* is specified during initialization. Add other decorative picture structure flags to indicate additional display attributes. Use *MFX_PICSTRUCT_UNKNOWN* during initialization for field attributes and *MFX_PICSTRUCT_PROGRESSIVE* for frame attributes. See the *PicStruct* enumerator for details.

VPP Constraints

The *VPP Constraints table* lists *VPP* parameter constraints.

Table 14: VPP Constraints

Parameters	During Initialization	During Operation
Width, Height	Any valid values	The values must be the equal to or larger than the initialization values.
CropX, CropY, CropW, CropH	Ignored	These parameters specify the region of interest from input to output.
AspectRatioW, AspectRatioH	Ignored	Aspect ratio values will be passed through from input to output.
FrameRateExtN, FrameRateExtD	Any valid values	Frame rate values will be updated with the initialization value at output.
PicStruct	*MFX_PICSTRUCT_UNKNOWN* *MFX_PICSTRUCT_PROGRESSIVE* *MFX_PICSTRUCT_FIELD_TFF* *MFX_PICSTRUCT_FIELD_BFF* *MFX_PICSTRUCT_FIELD_SINGLE* *MFX_PICSTRUCT_FIELD_TOP* *MFX_PICSTRUCT_FIELD_BOTTOM*	The base value must be the same as the initialization value unless *MFX_PICSTRUCT_UNKNOWN* is specified during initialization. Other decorative picture structure flags are passed through or added as needed. See the *PicStruct* enumerator for details.

Specifying Configuration Parameters

The following *Configuration Parameters tables* summarize how to specify the configuration parameters during initialization, encoding, decoding, and video processing.

Table 15: mfxVideoParam Configuration Parameters

Structure (param)	ENCODE Init	ENCODE Encoding	DECODE Init	DECODE Decoding	VPP Init	VPP Processing
Protected	R	•	R	•	R	•
IOPattern	M	•	M	•	M	•
ExtParam	O	•	O	•	O	•
NumExtParam	O	•	O	•	O	•

Table 16: mfxInfoMFX Configuration Parameters

Structure (param)	ENCODE Init	ENCODE Encoding	DECODE Init	DECODE Decoding	VPP Init	VPP Processing
CodecId	M	•	M	•	•	•
CodecProfile	O	•	O/M*	•	•	•
CodecLevel	O	•	O	•	•	•
NumThread	O	•	O	•	•	•
TargetUsage	O	•	•	•	•	•
GopPicSize	O	•	•	•	•	•
GopRefDist	O	•	•	•	•	•
GopOptFlag	O	•	•	•	•	•
IdrInterval	O	•	•	•	•	•
RateControlMethod	O	•	•	•	•	•
InitialDelayInKB	O	•	•	•	•	•
BufferSizeInKB	O	•	•	•	•	•
TargetKbps	M	•	•	•	•	•
MaxKbps	O	•	•	•	•	•
NumSlice	O	•	•	•	•	•
NumRefFrame	O	•	•	•	•	•
EncodedOrder	M	•	•	•	•	•

Table 17: mfxFrameInfo Configuration Parameters

Structure (param)	ENCODE Init	ENCODE Encoding	DECODE Init	DECODE Decoding	VPP Init	VPP Processing
FourCC	M	M	M	M	M	M
Width	M	M	M	M	M	M
Height	M	M	M	M	M	M
CropX	M	Ign	Ign	U	Ign	M
CropY	M	Ign	Ign	U	Ign	M
CropW	M	Ign	Ign	U	Ign	M
CropH	M	Ign	Ign	U	Ign	M
FrameRateExtN	M	Ign	O	U	M	U
FrameRateExtD	M	Ign	O	U	M	U
AspectRatioW	O	Ign	O	U	Ign	PT
AspectRatioH	O	Ign	O	U	Ign	PT
PicStruct	O	M	Ign	U	M	M/U
ChromaFormat	M	M	M	M	Ign	Ign

Table 18: Abbreviations used in configuration parameter tables

Abbreviation	Meaning
Ign	Ignored
PT	Pass Through
•	Does Not Apply
M	Mandated
R	Reserved
O	Optional
U	Updated at output

Note: *CodecProfile* is mandated for HEVC REXT and SCC profiles and optional for other cases. If the application does not explicitly set CodecProfile during initialization, the HEVC decoder will use a profile up to Main10.

11.6.3 Multiple-segment Encoding

Multiple-segment encoding is useful in video editing applications during production, for example when the encoder encodes multiple video clips according to their time line. In general, one can define multiple-segment encoding as dividing an input sequence of frames into segments and encoding them in different encoding sessions with the same or different parameter sets. For example:

Segment Already Encoded	Segment in Encoding	Segment to be Encoded
0s	200s	500s

Note: Different encoders can also be used.

The application must be able to:

- Extract encoding parameters from the bitstream of previously encoded segment.
- Import these encoding parameters to configure the encoder.

Encoding can then continue on the current segment using either the same or similar encoding parameters.

Extracting the header that contains the encoding parameter set from the encoded bitstream is usually the task of a format splitter (de-multiplexer). Alternatively, the *MFXVideoDECODE_DecodeHeader()* function can export the raw header if the application attaches the *mfxExtCodingOptionSPSPPS* structure as part of the parameters.

The encoder can use the *mfxExtCodingOptionSPSPPS* structure to import the encoding parameters during *MFXVideoENCODE_Init()*. The encoding parameters are in the encoded bitstream format. Upon a successful import of the header parameters, the encoder will generate bitstreams with a compatible (not necessarily bit-exact) header. The *Header Import Functions table* shows all functions that can import a header and their error codes if there are unsupported parameters in the header or the encoder is unable to achieve compatibility with the imported header.

Table 19: Header Import Functions

Function Name	Error Code if Import Fails
MFXVideoENCODE_Init()	*MFX_ERR_INCOMPATIBLE_VIDEO_PARAM*
MFXVideoENCODE_QueryIOSurf()	*MFX_ERR_INCOMPATIBLE_VIDEO_PARAM*
MFXVideoENCODE_Reset()	*MFX_ERR_INCOMPATIBLE_VIDEO_PARAM*
MFXVideoENCODE_Query()	*MFX_ERR_UNSUPPORTED*

The encoder must encode frames to a GOP sequence starting with an IDR frame for H.264 (or I frame for MPEG-2) to ensure that the current segment encoding does not refer to any frames in the previous segment. This ensures that the encoded segment is self-contained, allowing the application to insert the segment anywhere in the final bitstream. After encoding, each encoded segment is HRD compliant. Concatenated segments may not be HRD compliant.

The following example shows the encoder initialization procedure that imports H.264 sequence and picture parameter sets:

```
mfxStatus init_encoder() {
   mfxExtCodingOptionSPSPPS option, *option_array;

   /* configure mfxExtCodingOptionSPSPPS */
   memset(&option,0,sizeof(option));
   option.Header.BufferId=MFX_EXTBUFF_CODING_OPTION_SPSPPS;
   option.Header.BufferSz=sizeof(option);
   option.SPSBuffer=sps_buffer;
   option.SPSBufSize=sps_buffer_length;
   option.PPSBuffer=pps_buffer;
   option.PPSBufSize=pps_buffer_length;

   /* configure mfxVideoParam */
   mfxVideoParam param;
   //...
   param.NumExtParam=1;
   option_array=&option;
   param.ExtParam=(mfxExtBuffer**)&option_array;

   /* encoder initialization */
   mfxStatus status;
   status=MFXVideoENCODE_Init(session, &param);
   if (status==MFX_ERR_INCOMPATIBLE_VIDEO_PARAM) {
      printf("Initialization failed.\n");
   } else {
```

(continues on next page)

(continued from previous page)

```
        printf("Initialized.\n");
    }
    return status;
}
```

11.6.4 Streaming and Video Conferencing Features

The following sections address some aspects of additional requirements that streaming or video conferencing applications may use in the encoding or transcoding process. See the *Configuration Change* section for additional information.

Dynamic Bitrate Change

The oneVPL encoder supports dynamic bitrate change according to bitrate control mode and HRD conformance requirements. If HRD conformance is required, for example if the application sets the `NalHrdConformance` option in the *`mfxExtCodingOption`* structure to ON, the only allowed bitrate control mode is VBR. In this mode, the application can change the `TargetKbps` and `MaxKbps` values of the *`mfxInfoMFX`* structure by calling the *`MFXVideoENCODE_Reset()`* function. This sort of change in bitrate usually results in the generation of a new keyframe and sequence header. There are exceptions, such as if HRD information is absent in the stream. In this scenario, the change of `TargetKbps` does not require a change in the sequence header and as a result the encoder does not insert a keyframe.

If HRD conformance is not required, for example if the application turns off the `NalHrdConformance` option in the *`mfxExtCodingOption`* structure, all bitrate control modes are available. In CBR and AVBR modes the application can change `TargetKbps`. In VBR mode the application can change `TargetKbps` and `MaxKbps` values. This sort of change in bitrate will not result in the generation of a new keyframe or sequence header.

The oneVPL encoder may change some initialization parameters provided by the application during initialization. That in turn may lead to incompatibility between the parameters provided by the application during reset and the working set of parameters used by the encoder. For this reason, it is strongly recommended to retrieve the actual working parameters using the *`MFXVideoENCODE_GetVideoParam()`* function before making any changes to bitrate settings.

In all modes, oneVPL encoders will respond to the bitrate changes as quickly as the underlying algorithm allows, without breaking other encoding restrictions such as HRD compliance if it is enabled. How quickly the actual bitrate can catch up with the specified bitrate is implementation dependent.

Alternatively, the application may use the *CQP* encoding mode to perform customized bitrate adjustment on a per-frame base. The application may use any of the encoded or display order modes to use per-frame CQP.

Dynamic Resolution Change

The oneVPL encoder supports dynamic resolution change in all bitrate control modes. The application may change resolution by calling the *`MFXVideoENCODE_Reset()`* function. The application may decrease or increase resolution up to the size specified during encoder initialization.

Resolution change always results in the insertion of a key IDR frame and a new sequence parameter set in the header. The only exception is the oneVPL VP9 encoder (see section for *Dynamic reference frame scaling*). The oneVPL encoder does not guarantee HRD conformance across the resolution change point.

The oneVPL encoder may change some initialization parameters provided by the application during initialization. That in turn may lead to incompatibility of parameters provide by the application during reset and working set of parameters used by the encoder. Due to this potential incompatibility, it is strongly recommended to retrieve the actual working parameters set by *`MFXVideoENCODE_GetVideoParam()`* function before making any resolution change.

Dynamic Reference Frame Scaling

The VP9 standard allows changing the resolution without the insertion of a keyframe. This is possible because the VP9 encoder has the built-in capability to upscale and downscale reference frames to match the resolution of the frame being encoded. By default the oneVPL VP9 encoder inserts a keyframe when the application does *Dynamic Resolution Change*. In this case, the first frame with a new resolution is encoded using inter prediction from the scaled reference frame of the previous resolution. Dynamic scaling has the following limitations, described in the VP9 specification:

- The resolution of any active reference frame cannot exceed 2x the resolution of the current frame.
- The resolution of any active reference frame cannot be smaller than 1/16 of the current frame resolution.

In the case of dynamic scaling, the oneVPL VP9 encoder always uses a single active reference frame for the first frame after a resolution change. The VP9 encoder has the following limitations for dynamic resolution change:

- The new resolution should not exceed 16x the resolution of the current frame.
- The new resolution should be less than 1/2 of current frame resolution.

The application may force insertion of a keyframe at the point of resolution change by invoking encoder reset with *mfxExtEncoderResetOption::StartNewSequence* set to *MFX_CODINGOPTION_ON*. If a keyframe is inserted, the dynamic resolution limitations are not enforced.

Note that resolution change with dynamic reference scaling is compatible with multiref (*mfxInfoMFX::NumRefFrame* > 1). For multiref configuration, the oneVPL VP9 encoder uses multiple references within stream pieces of the same resolution and uses a single reference at the place of resolution change.

Forced Keyframe Generation

oneVPL supports forced keyframe generation during encoding. The application can set the FrameType parameter of the *mfxEncodeCtrl* structure to control how the current frame is encoded, as follows:

- If the oneVPL encoder works in the display order, the application can enforce any current frame to be a keyframe. The application cannot change the frame type of already buffered frames inside the encoder.
- If the oneVPL encoder works in the encoded order, the application must specify exact frame type for every frame. In this way, the application can enforce the current frame to have any frame type that the particular coding standard allows.

Reference List Selection

During streaming or video conferencing, if the application can obtain feedback about how well the client receives certain frames, the application may need to adjust the encoding process to use or not use certain frames as reference. This section describes how to fine-tune the encoding process based on client feedback.

The application can specify the reference window size by specifying the *mfxInfoMFX::NumRefFrame* parameter during encoding initialization. Certain platforms may have limits on the the size of the reference window. Use the *MFXVideoENCODE_GetVideoParam()* function to retrieve the current working set of parameters.

During encoding, the application can specify the actual reference list lengths by attaching the *mfxExtAVCRefListCtrl* structure to the *MFXVideoENCODE_EncodeFrameAsync()* function. `NumRefIdxL0Active` specifies the length of the reference list L0 and `NumRefIdxL1Active` specifies the length of the reference list L1. These two numbers must be less than or equal to the *mfxInfoMFX::NumRefFrame* parameter during encoding initialization.

The application can instruct the oneVPL encoder to use or not use certain reference frames. To do this, there is a pre-requisite that the application uniquely identify each input frame by setting the *mfxFrameData::FrameOrder* parameter. The application then specifies the preferred reference frame list `PreferredRefList` and/or

the rejected frame list `RejectedRefList`, and attaches the *mfxExtAVCRefListCtrl* structure to the *MFXVideoENCODE_EncodeFrameAsync()* function. The two lists fine-tune how the encoder chooses the reference frames for the current frame. The encoder does not keep `PreferredRefList` and the application must send it for each frame if necessary. There are limitations as follows:

- The frames in the lists are ignored if they are out of the reference window.
- If by going through the lists, the oneVPL encoder cannot find a reference frame for the current frame, the encoder will encode the current frame without using any reference frames.
- If the GOP pattern contains B-frames, the oneVPL encoder may not be able to follow the *mfxExtAVCRefListCtrl* instructions.

Low Latency Encoding and Decoding

The application can set *mfxVideoParam::AsyncDepth* = 1 to disable any decoder buffering of output frames, which is aimed to improve the transcoding throughput. With *mfxVideoParam::AsyncDepth* = 1, the application must synchronize after the decoding or transcoding operation of each frame.

The application can adjust *mfxExtCodingOption::MaxDecFrameBuffering* during encoding initialization to improve decoding latency. It is recommended to set this value equal to the number of reference frames.

Reference Picture Marking Repetition SEI Message

The application can request writing the reference picture marking repetition SEI message during encoding initialization by setting `RefPicMarkRep` of the *mfxExtCodingOption* structure. The reference picture marking repetition SEI message repeats certain reference frame information in the output bitstream for robust streaming.

The oneVPL decoder will respond to the reference picture marking repetition SEI message if the message exists in the bitstream and compare it to the reference list information specified in the sequence/picture headers. The decoder will report any mismatch of the SEI message with the reference list information in the *mfxFrameData::Corrupted* field.

Long Term Reference Frame

The application may use long term reference frames to improve coding efficiency or robustness for video conferencing applications. The application controls the long term frame marking process by attaching the *mfxExtAVCRefListCtrl* extended buffer during encoding. The oneVPL encoder itself never marks a frame as long term.

There are two control lists in the *mfxExtAVCRefListCtrl* extended buffer. The `LongTermRefList` list contains the frame orders (the `FrameOrder` value in the *mfxFrameData* structure) of the frames that should be marked as long term frames. The `RejectedRefList` list contains the frame order of the frames that should be unmarked as long term frames. The application can only mark or unmark the frames that are buffered inside the encoder. Because of this, it is recommended that the application marks a frame when it is submitted for encoding. The application can either explicitly unmark long term reference frames or wait for the IDR frame. When the IDR frame is reached, all long term reference frames will be unmarked.

The oneVPL encoder puts all long term reference frames at the end of a reference frame list. If the number of active reference frames (the `NumRefIdxL0Active` and `NumRefIdxL1Active` values in the *mfxExtAVCRefListCtrl* extended buffer) is less than than the total reference frame number (the `NumRefFrame` value in the *mfxInfoMFX* structure during the encoding initialization), the encoder may ignore some or all long term reference frames. The application may avoid this by providing a list of preferred reference frames in the `PreferredRefList` list in the *mfxExtAVCRefListCtrl* extended buffer. In this case, the encoder reorders the reference list based on the specified list.

Temporal Scalability

The application may specify the temporal hierarchy of frames by using the *mfxExtAvcTemporalLayers* extended buffer during the encoder initialization in the display order encoding mode. oneVPL inserts the prefix NAL unit before each slice with a unique temporal and priority ID. The temporal ID starts from zero and the priority ID starts from the `BaseLayerPID` value. oneVPL increases the temporal ID and priority ID value by one for each consecutive layer.

If the application needs to specify a unique sequence or picture parameter set ID, the application must use the *mfxExtCodingOptionSPSPPS* extended buffer, with all pointers and sizes set to zero and valid `SPSId` and `PPSId` fields. The same SPS and PPS ID will be used for all temporal layers.

Each temporal layer is a set of frames with the same temporal ID. Each layer is defined by the `Scale` value. The scale for layer N is equal to the ratio between the frame rate of subsequent temporal layers with a temporal ID less than or equal to N and the frame rate of the base temporal layer. The application may skip some temporal layers by specifying the `Scale` value as zero. The application should use an integer ratio of the frame rates for two consecutive temporal layers.

For example, a video sequence with 30 frames/second is typically separated by three temporal layers that can be decoded as 7.5 fps (base layer), 15 fps (base and first temporal layer) and 30 fps (all three layers). In this scenario, `Scale` should have the values {1,2,4,0,0,0,0,0}.

11.6.5 Switchable Graphics and Multiple Monitors

The following sections discuss support for switchable graphics and multiple monitor configurations.

Switchable Graphics

Switchable Graphics refers to the machine configuration that multiple graphic devices are available (integrated device for power saving and discrete devices for performance.) Usually at one time or instance, one of the graphic devices drives display and becomes the active device, and others become inactive. There are different variations of software or hardware mechanisms to switch between the graphic devices. In one of the switchable graphics variations, it is possible to register an application in an affinity list to certain graphic device so that the launch of the application automatically triggers a switch. The actual techniques to enable such a switch are outside the scope of this document. This section discusses the implication of switchable graphics to Intel® Media Software Development Kit and Intel® Media Software Development Kit applications.

As Intel® Media Software Development Kit performs hardware acceleration through graphic devices, it is critical that Intel® Media Software Development Kit can access the graphic device in the switchable graphics setting. It is recommended to add the application to the graphic device affinity list. If this is not possible, the application should handle the following cases:

- By design, during legacy Intel® Media Software Development Kit library initialization, the *MFXInit()* function searches for graphic devices. If a Intel® Media Software Development Kit implementation is successfully loaded, the *MFXInit()* function returns *mfxStatus::MFX_ERR_NONE* and the *MFXQueryIMPL()* function returns the actual implementation type. If no Intel® Media Software Development Kit implementation is loaded, the *MFXInit()* function returns *mfxStatus::MFX_ERR_UNSUPPORTED*. In the switchable graphics environment, if the application is not in the graphic device affinity list, it is possible that the graphic device will not be accessible during the library initialization. The fact that the *MFXInit()* function returns *mfxStatus::MFX_ERR_UNSUPPORTED* does not mean that hardware acceleration is permanently impossible. The user may switch the graphics later and the graphic device will become accessible. It is recommended that the application initialize the library right before the actual decoding, video processing, and encoding operations to determine the hardware acceleration capability.
- During decoding, video processing, and encoding operations, if the application is not in the graphic device affinity list, the previously accessible graphic device may become inaccessible due to a switch event. The

Intel® Media Software Development Kit functions will return *mfxStatus::MFX_ERR_DEVICE_LOST* or *mfxStatus::MFX_ERR_DEVICE_FAILED*, depending on when the switch occurs and what stage the Intel® Media Software Development Kit functions operate. The application should handle these errors and exit gracefully.

Multiple Monitors

Multiple monitors refer to the machine configuration that multiple graphic devices are available. Some graphic devices connect to a display and become active and accessible under the Microsoft* DirectX* infrastructure. Graphic devices that are not connected to a display are inactive. Using the Microsoft Direct3D* 9 infrastructure, devices that are not connected to a display are not accessible.

The legacy Intel® Media Software Development Kit uses the adapter number to access a specific graphic device. Usually, the graphic device driving the main desktop becomes the primary adapter. Other graphic devices take subsequent adapter numbers after the primary adapter. Under the Microsoft Direct3D 9 infrastructure, only active adapters are accessible and have an adapter number.

Intel® Media Software Development Kit extends the *mfxIMPL* implementation type as shown in the *Intel® Media SDK mfxIMPL Implementation Type Definitions table*:

Table 20: Intel® Media SDK mfxIMPL Implementation Type Definitions

Implementation Type	Definition
MFX_IMPL_HARDWARE	Intel® Media Software Development Kit should initialize on the primary adapter
MFX_IMPL_HARDWARE2	Intel® Media Software Development Kit should initialize on the 2nd graphic adapter
MFX_IMPL_HARDWARE3	Intel® Media Software Development Kit should initialize on the 3rd graphic adapter
MFX_IMPL_HARDWARE4	Intel® Media Software Development Kit should initialize on the 4th graphic adapter
MFX_IMPL_HARDWARE_ANY	Intel® Media Software Development Kit should initialize on any graphic adapter.
MFX_IMPL_AUTO_ANY	Intel® Media Software Development Kit should initialize on any graphic adapter. If not successful, load the software implementation.

The application can use the first four definitions shown in the *Intel® Media SDK mfxIMPL Implementation Type Definitions table* to instruct the legacy Intel® Media Software Development Kit library to initialize on a specific graphic device. The application can use the definitions for *MFX_IMPL_HARDWARE_ANY* and *MFX_IMPL_AUTO_ANY* for automatic detection.

If the application uses the Microsoft DirectX surfaces for I/O, it is critical that the application and Intel® Media Software Development Kit work on the same graphic device. It is recommended that the application use the following procedure:

1. The application uses the *MFXInit()* function to initialize the legacy Intel® Media Software Development Kit, with option *MFX_IMPL_HARDWARE_ANY* or *MFX_IMPL_AUTO_ANY*. The *MFXInit()* function returns *mfxStatus::MFX_ERR_NONE* if successful.
2. The application uses the *MFXQueryIMPL()* function to check the actual implementation type. The implementation type *MFX_IMPL_HARDWARE*, *MFX_IMPL_HARDWARE2*, *MFX_IMPL_HARDWARE3*, or *MFX_IMPL_HARDWARE4* indicates the graphic adapter the Intel® Media Software Development Kit works on.

3. The application creates the Direct3D device on the respective graphic adapter and passes it to Intel® Media Software Development Kit through the *MFXVideoCORE_SetHandle()* function.

Similar to the switchable graphics cases, interruption may result if the user disconnects monitors from the graphic devices or remaps the primary adapter. If the interruption occurs during the Intel® Media Software Development Kit library initialization, the *MFXInit()* function may return *mfxStatus::MFX_ERR_UNSUPPORTED*. This means hardware acceleration is currently not available. It is recommended that the application initialize Intel® Media Software Development Kit right before the actual decoding, video processing, and encoding operations to determine the hardware acceleration capability.

If the interruption occurs during decoding, video processing, or encoding operations, oneVPL functions will return *mfxStatus::MFX_ERR_DEVICE_LOST* or *mfxStatus::MFX_ERR_DEVICE_FAILED*. The application should handle these errors and exit gracefully.

11.6.6 Working Directly with VA API for Linux*

Intel® Media Software Development Kit takes care of all memory and synchronization related operations in the VA API. The application may need to extend Intel® Media Software Development Kit functionality by working directly with the VA API for Linux*, for example to implement a customized external allocator. This section describes basic memory management and synchronization techniques.

To create the VA surface pool, the application should call the vaCreateSurfaces function:

```
VASurfaceAttrib attrib;
attrib.type = VASurfaceAttribPixelFormat;
attrib.value.type = VAGenericValueTypeInteger;
attrib.value.value.i = VA_FOURCC_NV12;
attrib.flags = VA_SURFACE_ATTRIB_SETTABLE;

#define NUM_SURFACES 5;
VASurfaceID surfaces[NUMSURFACES];

vaCreateSurfaces(va_display, VA_RT_FORMAT_YUV420, width, height, surfaces, NUM_
↪SURFACES, &attrib, 1);
```

To destroy the surface pool, the application should call the vaDestroySurfaces function:

```
vaDestroySurfaces(va_display, surfaces, NUM_SURFACES);
```

If the application works with hardware acceleration through Intel® Media Software Development Kit, then it can access surface data immediately after successful completion of the *MFXVideoCORE_SyncOperation()* call. If the application works with hardware acceleration directly, then it must check surface status before accessing data in video memory. This check can be done asynchronously by calling the vaQuerySurfaceStatus function or synchronously by calling the vaSyncSurface function.

After successful synchronization, the application can access surface data. Accessing surface data is performed in two steps:

1. Create VAImage from surface.
2. Map image buffer to system memory.

After mapping, the VAImage.offsets[3] array holds offsets to each color plain in a mapped buffer and the VAImage.pitches[3] array holds color plain pitches in bytes. For packed data formats, only first entries in these arrays are valid. The following example shows how to access data in a NV12 surface:

```
VAImage image;
unsigned char *buffer, Y, U, V;

vaDeriveImage(va_display, surface_id, &image);
vaMapBuffer(va_display, image.buf, &buffer);

/* NV12 */
Y = buffer + image.offsets[0];
U = buffer + image.offsets[1];
V = U + 1;
```

After processing data in a VA surface, the application should release resources allocated for the mapped buffer and VAImage object:

```
vaUnmapBuffer(va_display, image.buf);
vaDestroyImage(va_display, image.image_id);
```

In some cases, in order to retrieve encoded bitstream data from video memory, the application must use the VABuffer to store data. The following example shows how to create, use, and destroy the VABuffer:

```
/* create buffer */
VABufferID buf_id;
vaCreateBuffer(va_display, va_context, VAEncCodedBufferType, buf_size, 1, NULL, & buf_
↪id);

/* encode frame */
// ...

/* map buffer */
VACodedBufferSegment *coded_buffer_segment;

vaMapBuffer(va_display, buf_id, (void **)(& coded_buffer_segment));

size   = coded_buffer_segment->size;
offset = coded_buffer_segment->bit_offset;
buf    = coded_buffer_segment->buf;

/* retrieve encoded data*/
// ...

/* unmap and destroy buffer */
vaUnmapBuffer(va_display, buf_id);
vaDestroyBuffer(va_display, buf_id);
```

Note that the vaMapBuffer function returns pointers to different objects depending on the mapped buffer type. The VAImage is a plain data buffer and the encoded bitstream is a VACodedBufferSegment structure. The application cannot use VABuffer for synchronization. If encoding, it is recommended to synchronize using the VA surface as described above.

11.6.7 CQP HRD Mode Encoding

The application can configure an AVC encoder to work in CQP rate control mode with HRD model parameters. oneVPL will place HRD information to SPS/VUI and choose the appropriate profile/level. It's the responsibility of the application to provide per-frame QP, track HRD conformance, and insert required SEI messages to the bitstream.

The following example shows how to enable CQP HRD mode. The application should set *RateControlMethod* to CQP, *mfxExtCodingOption::VuiNalHrdParameters* to ON, *mfxExtCodingOption::NalHrdConformance* to OFF, and set rate control parameters similar to CBR or VBR modes (instead of QPI, QPP, and QPB). oneVPL will choose CBR or VBR HRD mode based on the `MaxKbps` parameter. If `MaxKbps` is set to zero, oneVPL will use CBR HRD model (write cbr_flag = 1 to VUI), otherwise the VBR model will be used (and cbr_flag = 0 is written to VUI).

Note: For CQP, if implementation does not support individual QPI, QPP and QPB parameters, then QPI parameter should be used as a QP parameter across all frames.

```
mfxExtCodingOption option, *option_array;

/* configure mfxExtCodingOption */
memset(&option,0,sizeof(option));
option.Header.BufferId          = MFX_EXTBUFF_CODING_OPTION;
option.Header.BufferSz          = sizeof(option);
option.VuiNalHrdParameters      = MFX_CODINGOPTION_ON;
option.NalHrdConformance        = MFX_CODINGOPTION_OFF;

/* configure mfxVideoParam */
mfxVideoParam param;

// ...

param.mfx.RateControlMethod          = MFX_RATECONTROL_CQP;
param.mfx.FrameInfo.FrameRateExtN    = valid_non_zero_value;
param.mfx.FrameInfo.FrameRateExtD    = valid_non_zero_value;
param.mfx.BufferSizeInKB             = valid_non_zero_value;
param.mfx.InitialDelayInKB           = valid_non_zero_value;
param.mfx.TargetKbps                 = valid_non_zero_value;

if (write_cbr_flag == 1)
   param.mfx.MaxKbps = 0;
else /* write_cbr_flag = 0 */
   param.mfx.MaxKbps = valid_non_zero_value;

param.NumExtParam = 1;
option_array     = &option;
param.ExtParam     = (mfxExtBuffer **)&option_array;

/* encoder initialization */
mfxStatus sts;
sts = MFXVideoENCODE_Init(session, &param);

// ...

/* encoding */
mfxEncodeCtrl ctrl;
memset(&ctrl,0,sizeof(ctrl));
ctrl.QP = frame_qp;
```

(continues on next page)

(continued from previous page)

```
41
42 sts=MFXVideoENCODE_EncodeFrameAsync(session,&ctrl,surface2,bits,&syncp);
```

11.7 Glossary

The oneVPL API and documentation uses a standard set of acronyms and terms. This section describes these conventions.

11.7.1 Acronyms and Terms

AVC Advanced video codec (same as H.264 and MPEG-4, part 10).

BRC Bit rate control.

CQP Constant quantization parameter.

DRM Digital rights management.

DXVA2 Microsoft DirectX* Video Acceleration standard 2.0.

GOP Group of pictures. In video coding, a group of frames in a specific order. In the H.264 standard, a group of I-frames, B-frames and P-frames.

GPB Generalized P/B picture. B-picture, containing only forward references in both L0 and L1.

H.264 Video coding standard. See ISO*/IEC* 14496-10 and ITU-T* H.264, MPEG-4 Part 10, Advanced Video Coding, May 2005.

HDR High dynamic range.

HRD Hypothetical reference decoder, a term used in the H.264 specification.

IDR Instantaneous decoding fresh picture, a term used in the H.264 specification.

LA Look ahead. Special encoding mode where encoder performs pre-analysis of several frames before actual encoding starts.

MCTF Motion compensated temporal filter. Special type of noise reduction filter which utilizes motion to improve efficiency of video denoising.

NAL Network abstraction layer.

PPS Picture parameter set.

QP Quantization parameter.

SEI Supplemental enhancement information.

SPS Sequence parameter set.

VA API Video acceleration API.

VBR Variable bit rate.

VBV Video buffering verifier.

Video memory Memory used by a hardware acceleration device, also known as GPU, to hold frame and other types of video data.

VUI Video usability information.

11.7.2 Video Formats

MPEG Moving Picture Experts Group video file.

MPEG-2 Moving Picture Experts Group video file. See ISO/IEC 13818-2 and ITU-T H.262, MPEG-2 Part 2, Information Technology- Generic Coding of Moving Pictures and Associate Audio Information: Video, 2000.

NV12 YUV 4:2:0 video format, 12 bits per pixel.

NV16 YUV 4:2:2 video format, 16 bits per pixel.

P010 YUV 4:2:0 video format, extends NV12, 10 bits per pixel.

P210 YUV 4:2:2 video format, 10 bits per pixel.

UYVY YUV 4:2:2 video format, 16 bits per pixel.

VC-1 Video coding format. See SMPTE* 421M, SMPTE Standard for Television: VC-1 Compressed Video Bitstream Format and Decoding Process, August 2005.

11.7.3 Color Formats

I010 Color format for raw video frames, extends IYUV/I420 for 10 bit.

IYUV A color format for raw video frames, also known as I420.

RGB32 Thirty-two-bit RGB color format.

RGB4 Thirty-two-bit RGB color format. Also known as RGB32.

YUY2 A color format for raw video frames.

YV12 A color format for raw video frames, similar to IYUV with U and V reversed.

CHAPTER

TWELVE

ONEMKL

The oneAPI Math Kernel Library (oneMKL) defines a set of fundamental mathematical routines for use in high-performance computing and other applications. As part of oneAPI, oneMKL is designed to allow execution on a wide variety of computational devices: CPUs, GPUs, FPGAs, and other accelerators. The functionality is subdivided into several domains: dense linear algebra, sparse linear algebra, discrete Fourier transforms, random number generators and vector math.

The general assumptions, design features and requirements for the oneMKL library and host-to-device computational routines will be described in *oneMKL Architecture*. The individual domains and their APIs are described in *oneMKL Domains*. Other design considerations that are not necessarily part of the oneMKL specification but that are worth mentioning will be discussed in *oneMKL Appendix*.

12.1 oneMKL Architecture

The oneMKL element of oneAPI has several general assumptions, requirements and recommendations for all domains contained therein. These will be addressed in this architecture section. In particular, DPC++ allows for a great control over the execution of kernels on the various devices. We discuss the supported execution models of oneMKL APIs in *Execution Model*. A discussion of how data is stored and passed in and out of the APIs is addressed in *Memory Model*. The general structure and design of oneMKL APIs including namespaces and common data types are expressed in *API Design*. The exceptions and error handling are described in *Exceptions and Error Handling*. Finally all the other necessary aspects related to oneMKL architecture can be found in *Other Features* including versioning and discussion of pre and post conditions. Other nonessential, but useful aspects of the oneMKL architecture and design may also be found in the *oneMKL Appendix*.

12.1.1 Execution Model

This section describes the execution environment common to all oneMKL functionality. The execution environment includes how data is provided to computational routines in *Use of Queues*, support for several devices in *Device Usage*, synchronous and asynchronous execution models in *Asynchronous Execution* and *Host Thread Safety*.

Use of Queues

The `sycl::queue` defined in the oneAPI DPC++ specification is used to specify the device and features enabled on that device on which a task will be enqueued. There are two forms of computational routines in oneMKL: class based *Member Functions* and standalone *Non-Member Functions*. As these may interact with the `sycl::queue` in different ways, we provide a section for each one to describe assumptions.

Non-Member Functions

Each oneMKL non-member computational routine takes a `sycl::queue` reference as its first parameter:

```
mkl::domain::routine(sycl::queue &q, ...);
```

All computation performed by the routine shall be done on the hardware device(s) associated with this queue, with possible aid from the host, unless otherwise specified. In the case of an ordered queue, all computation shall also be ordered with respect to other kernels as if enqueued on that queue.

A particular oneMKL implementation may not support the execution of a given oneMKL routine on the specified device(s). In this case, the implementation may either perform the computation on the host or throw an exception. See *Exceptions and Error Handling* for the possible exceptions.

Member Functions

oneMKL class-based APIs, such as those in the RNG and DFT domains, require a `sycl::queue` as an argument to the constructor or another setup routine. The execution requirements for computational routines from the previous section also apply to computational class methods.

Device Usage

oneMKL itself does not currently provide any interfaces for controlling device usage: for instance, controlling the number of cores used on the CPU, or the number of execution units on a GPU. However, such functionality may be available by partitioning a `sycl::device` instance into subdevices, when supported by the device.

When given a queue associated with such a subdevice, a oneMKL implementation shall only perform computation on that subdevice.

Asynchronous Execution

The oneMKL API is designed to allow asynchronous execution of computational routines, to facilitate concurrent usage of multiple devices in the system. Each computational routine enqueues work to be performed on the selected device, and may (but is not required to) return before execution completes.

Hence, it is the calling application's responsibility to ensure that any inputs are valid until computation is complete, and likewise to wait for computation completion before reading any outputs. This can be done automatically when using DPC++ buffers, or manually when using Unified Shared Memory (USM) pointers, as described in the sections below.

Unless otherwise specified, asynchronous execution is *allowed*, but not *guaranteed*, by any oneMKL computational routine, and may vary between implementations and/or versions. oneMKL implementations must clearly document whether execution is guaranteed to be asynchronous for each supported routine. Regardless, calling applications shall not launch any oneMKL computational routine with a dependency on a future oneMKL API call, even if this computational routine executes asynchronously (i.e. a oneMKL implementation may assume no antidependencies are present). This guarantee allows oneMKL implementations to reserve resources for execution without risking deadlock.

Synchronization When Using Buffers

`sycl::buffer` objects automatically manage synchronization between kernel launches linked by a data dependency (either read-after-write, write-after-write, or write-after-read).

oneMKL routines are not required to perform any additional synchronization of `sycl::buffer` arguments.

Synchronization When Using USM APIs

When USM pointers are used as input to, or output from, a oneMKL routine, it becomes the calling application's responsibility to manage possible asynchronicity.

To help the calling application, all oneMKL routines with at least one USM pointer argument also take an optional reference to a list of *input events*, of type `sycl::vector_class<sycl::event>`, and have a return value of type `sycl::event` representing computation completion:

```
sycl::event mkl::domain::routine(..., sycl::vector_class<sycl::event> &in_events = {}
↪);
```

The routine shall ensure that all input events (if the list is present and non-empty) have occurred before any USM pointers are accessed. Likewise, the routine's output event shall not be complete until the routine has finished accessing all USM pointer arguments.

For class methods, "argument" includes any USM pointers previously provided to the object via the class constructor or other class methods.

Host Thread Safety

All oneMKL member and non-member functions shall be *host thread safe*. That is, they may be safely called simultaneously from concurrent host threads. However, oneMKL objects in class-based APIs may not be shared between concurrent host threads unless otherwise specified.

12.1.2 Memory Model

The oneMKL memory model shall follow directly from the oneAPI memory model. Mainly, oneMKL shall support two modes of encapsulating data for consumption on the device: the buffer memory abstraction model and the pointer-based memory model using Unified Shared Memory (USM). These two paradigms shall also support both synchronous and asynchronous execution models as described in *Asynchronous Execution*.

The Buffer Memory Model

The SYCL 1.2.1 specification defines the buffer container templated on the provided data type which encapsulates the data in a SYCL application across both host and devices. It provides the concept of accessors as the mechanism to access the buffer data with different modes to read and or write into that data. These accessors allow SYCL to create and manage the data dependencies in the SYCL graph that order the kernel executions. With the buffer model, all data movement is handled by the SYCL runtime supporting both synchronous and asynchronous execution.

oneMKL provides APIs where buffers (in particular 1D buffers, `sycl::buffer<T,1>`) contain the memory for all non scalar input and output data arguments. See *Synchronization When Using Buffers* for details on how oneMKL routines manage any data dependencies with buffer arguments. Any higher dimensional buffer must be converted to a 1D buffer prior to use in oneMKL APIs, e.g., via `buffer::reinterpret`.

Unified Shared Memory Model

While the buffer model is powerful and elegantly expresses data dependencies, it can be a burden for programmers to replace all pointers and arrays by buffers in their C++ applications. DPC++ also provides pointer-based addressing for device-accessible data, using the Unified Shared Memory (USM) model. Correspondingly, oneMKL provides USM APIs in which non-scalar input and output data arguments are passed by USM pointer.

USM devices and system configurations vary in their ability to share data between devices and between a device and the host. oneMKL implementations may only assume that user-provided USM pointers are accessible by the device associated with the user-provided queue. In particular, an implementation must not assume that USM pointers can be accessed by any other device, or by the host, without querying the DPC++ runtime. An implementation must accept any device-accessible USM pointer regardless of how it was created (*sycl::malloc_device*, *sycl::malloc_shared*, etc.).

Unlike buffers, USM pointers cannot automatically manage data dependencies between kernels. Users may use in-order queues to ensure ordered execution, or explicitly manage dependencies with `sycl::event` objects. To support the second use case, oneMKL USM APIs accept input events (prerequisites before computation can begin) and return an output event (indicating computation is complete). See *Synchronization When Using USM APIs* for details.

12.1.3 API Design

This section discusses the general features of oneMKL API design. In particular, it covers the use of namespaces and data types from C++, from DPC++ and new ones introduced for oneMKL APIs.

oneMKL namespaces

The oneMKL library uses C++ namespaces to organize routines by mathematical domain. All oneMKL objects and routines shall be contained within the `oneapi::mkl` base namespace. The individual oneMKL domains use a secondary namespace layer as follows:

namespace	oneMKL domain or content
`oneapi::mkl`	oneMKL base namespace, contains general oneMKL data types, objects, exceptions and routines
`oneapi::mkl::blas`	Dense linear algebra routines from BLAS and BLAS like extensions. The oneapi::mkl::blas namespace should contain two namespaces column_major and row_major to support both matrix layouts. See *BLAS Routines*
`oneapi::mkl::lapack`	Dense linear algebra routines from LAPACK and LAPACK like extensions. See *LAPACK Routines*
`oneapi::mkl::sparse`	Sparse linear algebra routines from Sparse BLAS and Sparse Solvers. See *Sparse Linear Algebra*
`oneapi::mkl::dft`	Discrete and fast Fourier transformations. See *Discrete Fourier Transform Functions*
`oneapi::mkl::rng`	Random number generator routines. See *Random Number Generators*
`oneapi::mkl::vm`	Vector mathematics routines, e.g. trigonometric, exponential functions acting on elements of a vector. See *Vector Math*

Note: Inside each oneMKL domain, there are many routines, classes, enums and objects defined which constitute the breadth and scope of that oneMKL domain. It is permitted for a library implementation of the oneMKL specification to implement either all, one or more than one of the domains in oneMKL. However, within an implementation of a specific domain, all relevant routines, classes, enums and objects (including those relevant enums and objects which live outside a particular domain in the general `oneapi::mkl` namespace must be both declared and defined in the library so that an application that uses that domain could build and link against that library implementation successfully.

It is however acceptable to throw the runtime exception *oneapi::mkl::unimplemented* inside of the routines or class member functions in that domain that have not been fully implemented. For instance, a library may choose to implement the oneMKL BLAS functionality and in particular may choose to implement only the *gemm* api for their library, in which case they must also include all the other blas namespaced routines and throw the *oneapi::mkl::unimplemented* exception inside all the others.

In such a case, the implemented routines in such a library should be communicated clearly and easily understood by users of that library.

Standard C++ datatype usage

oneMKL uses C++ STL data types for scalars where applicable:

- Integer scalars are C++ fixed-size integer types (`std::intN_t`, `std::uintN_t`).
- Complex numbers are represented by C++ `std::complex` types.

In general, scalar integer arguments to oneMKL routines are 64-bit integers (`std::int64_t` or `std::uint64_t`). Integer vectors and matrices may have varying bit widths, defined on a per-routine basis.

DPC++ datatype usage

oneMKL uses the following DPC++ data types:

- SYCL queue `sycl::queue` for scheduling kernels on a SYCL device. See *Use of Queues* for more details.
- SYCL buffer `sycl::buffer` for buffer-based memory access. See *The Buffer Memory Model* for more details.
- Unified Shared Memory (USM) for pointer-based memory access. See *Unified Shared Memory Model* for more details.
- SYCL event `sycl::event` for output event synchronization in oneMKL routines with USM pointers. See *Synchronization When Using USM APIs* for more details.
- Vector of SYCL events `sycl::vector_class<sycl::event>` for input events synchronization in oneMKL routines with USM pointers. See *Synchronization When Using USM APIs* for more details.

oneMKL defined datatypes

oneMKL dense and sparse linear algebra routines use scoped enum types as type-safe replacements for the traditional character arguments used in C/Fortran implementations of BLAS and LAPACK. These types all belong to the `oneapi::mkl` namespace.

Each enumeration value comes with two names: A single-character name (the traditional BLAS/LAPACK character) and a longer, more descriptive name. The two names are exactly equivalent and may be used interchangeably.

transpose

The `transpose` type specifies whether an input matrix should be transposed and/or conjugated. It can take the following values:

Short Name	Long Name	Description
`transpose::N`	`transpose::nontrans`	Do not transpose or conjugate the matrix.
`transpose::T`	`transpose::trans`	Transpose the matrix.
`transpose::C`	`transpose::conjtrans`	Perform Hermitian transpose (transpose and conjugate). Only applicable to complex matrices.

uplo

The `uplo` type specifies whether the lower or upper triangle of a triangular, symmetric, or Hermitian matrix should be accessed. It can take the following values:

Short Name	Long Name	Description
`uplo::U`	`uplo::upper`	Access the upper triangle of the matrix.
`uplo::L`	`uplo::lower`	Access the lower triangle of the matrix.

In both cases, elements that are not in the selected triangle are not accessed or updated.

diag

The `diag` type specifies the values on the diagonal of a triangular matrix. It can take the following values:

Short Name	Long Name	Description
`diag::N`	`diag::nonunit`	The matrix is not unit triangular. The diagonal entries are stored with the matrix data.
`diag::U`	`diag::unit`	The matrix is unit triangular (the diagonal entries are all 1's). The diagonal entries in the matrix data are not accessed.

side

The `side` type specifies the order of matrix multiplication when one matrix has a special form (triangular, symmetric, or Hermitian):

Short Name	Long Name	Description
`side::L`	`side::left`	The special form matrix is on the left in the multiplication.
`side::R`	`side::right`	The special form matrix is on the right in the multiplication.

offset

The `offset` type specifies whether the offset to apply to an output matrix is a fix offset, column offset or row offset. It can take the following values

Short Name	Long Name	Description
`offset::F`	`offset::fix`	The offset to apply to the output matrix is fix, all the inputs in the `C_offset` matrix has the same value given by the first element in the `co` array.
`offset::C`	`offset::column`	The offset to apply to the output matrix is a column offset, that is to say all the columns in the `C_offset` matrix are the same and given by the elements in the `co` array.
`offset::R`	`offset::row`	The offset to apply to the output matrix is a row offset, that is to say all the rows in the `C_offset` matrix are the same and given by the elements in the `co` array.

index_base

The `index_base` type specifies how values in index arrays are interpreted. For instance, a sparse matrix stores nonzero values and the indices that they correspond to. The indices are traditionally provided in one of two forms: C/C++-style using zero-based indices, or Fortran-style using one-based indices. The `index_base` type can take the following values:

Name	Description
`index_base::zero`	Index arrays for an input matrix are provided using zero-based (C/C++ style) index values. That is, indices start at 0.
`index_base::one`	Index arrays for an input matrix are provided using one-based (Fortran style) index values. That is, indices start at 1.

Note: *oneMKL Appendix* may contain other API design decisions or recommendations that may be of use to the general developer of oneMKL, but which may not necessarily be part of the oneMKL specification.

12.1.4 Exceptions and Error Handling

oneMKL error handling relies on the mechanism of C++ exceptions. Should error occur, it will be propagated at the point of a function call where it is caught using standard C++ error handling mechanism.

Exception classification

Exception classification in oneMKL is aligned with C++ Standard Library classification. oneMKL introduces class that defines the base class in the hierarchy of oneMKL exception classes. All oneMKL routines throw exceptions inherited from this base class. In the hierarchy of oneMKL exceptions, `oneapi::mkl::exception` is the base class inherited from `std::exception` class. All other oneMKL exception classes are derived from this base class.

This specification does not require implementations to perform error-checking. However, if an implementation does provide error-checking, it shall use the following exception classes. Additional implementation-specific exception classes can be used for exceptional conditions not fitting any of these classes.

Common exceptions

Exception class	Description
`oneapi::mkl::exception`	Reports general unspecified problem
`oneapi::mkl::unsupported_device`	Reports a problem when the routine is not supported on a specific device
`oneapi::mkl::host_bad_alloc`	Reports a problem that occurred during memory allocation on the host
`oneapi::mkl::device_bad_alloc`	Reports a problem that occurred during memory allocation on a specific device
`oneapi::mkl::unimplemented`	Reports a problem when a specific routine has not been implemented for the specified parameters
`oneapi::mkl::invalid_argument`	Reports problem when arguments to the routine were rejected
`oneapi::mkl::uninitialized`	Reports problem when a handle (descriptor) has not been initialized
`oneapi::mkl::computation_error`	Reports any computation errors that have occurred inside a oneMKL routine
`oneapi::mkl::batch_error`	Reports errors that have occurred inside a batch oneMKL routine

LAPACK specific exceptions

Exception class	Description
`oneapi::mkl::lapack::exception`	Base class for all LAPACK exceptions providing access to info code familiar to users of conventional LAPACK API. All LAPACK related exceptions can be handled with catch block for this class.
`oneapi::mkl::lapack::invalid_argument`	Reports errors when arguments provided to the LAPACK subroutine are inconsistent or do not match expected values. Class extends base `oneapi::mkl::invalid_argument` with ability to access conventional status info code.
`oneapi::mkl::lapack::computation_error`	Reports computation errors that have occurred during call to LAPACK subroutine. Class extends base `oneapi::mkl::computation_error` with ability to access conventional status info code familiar to LAPACK users.
`oneapi::mkl::lapack::batch_error`	Reports errors that have occurred during batch LAPACK computations. Class extends base `oneapi::mkl::batch_error` with ability to access individual exception objects for each of the issues observed in a batch and an info code. The info code contains the number of errors that occurred in a batch. Positions of problems in a supplied batch that experienced issues during computations can be retrieved with *ids()* method, and list of particular exceptions can be obtained with *exceptions()* method of the exception object. Possible exceptions for a batch are documented for corresponding non-batch API.

12.1.5 Other Features

This section covers all other features in the design of oneMKL architecture.

Current Version of this oneMKL Specification

This is the oneMKL specification which is part of the oneAPI specification version 1.0.0.

Pre/Post Condition Checking

The individual oneMKL computational routines will define any preconditions and postconditions and will define in this specification any specific checks or verifications that should be enabled for all implementations.

12.2 oneMKL Domains

This section describes the Data Parallel C++ (DPC++) interface.

12.2.1 Dense Linear Algebra

This section contains information about dense linear algebra routines:

Matrix Storage provides information about dense matrix and vector storage formats that are used by oneMKL *BLAS Routines* and *LAPACK Routines*.

BLAS Routines provides vector, matrix-vector, and matrix-matrix routines for dense matrices and vector operations.

LAPACK Routines provides more complex dense linear algebra routines, e.g., matrix factorization, solving dense systems of linear equations, least square problems, eigenvalue and singular value problems, and performing a number of related computational tasks.

Matrix Storage

The oneMKL BLAS and LAPACK routines for DPC++ use several matrix and vector storage formats. These are the same formats used in traditional Fortran BLAS/LAPACK.

General Matrix

A general matrix `A` of `m` rows and `n` columns with leading dimension `lda` is represented as a one dimensional array `a` of size of at least `lda * n` if column major layout is used and at least `lda * m` if row major layout is used. Before entry in any BLAS function using a general matrix, the leading `m` by `n` part of the array `a` must contain the matrix `A`. For column (respectively row) major layout, the elements of each column (respectively row) are contiguous in memory while the elements of each row (respectively column) are at distance `lda` from the element in the same row (respectively column) and the previous column (respectively row).

Visually, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \dots & A_{1n} \\ A_{21} & A_{22} & A_{23} & \dots & A_{2n} \\ A_{31} & A_{32} & A_{33} & \dots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & A_{m3} & \dots & A_{mn} \end{bmatrix}$$

is stored in memory as an array

- For column major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{21}, A_{31}, ..., A_{m1}, *, ..., *}_{\text{lda}}, \underbrace{A_{12}, A_{22}, A_{32}, ..., A_{m2}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{1n}, A_{2n}, A_{3n}, ..., A_{mn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

- For row major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{12}, A_{13}, ..., A_{1n}, *, ..., *}_{\text{lda}}, \underbrace{A_{21}, A_{22}, A_{23}, ..., A_{2n}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{m1}, A_{m2}, A_{m3}, ..., A_{mn}, *, ..., *}_{\text{lda}}}_{\text{m x lda}}]$$

Triangular Matrix

A triangular matrix `A` of `n` rows and `n` columns with leading dimension `lda` is represented as a one dimensional array `a`, of a size of at least `lda * n`. When column (respectively row) major layout is used, the elements of each column (respectively row) are contiguous in memory while the elements of each row (respectively column) are at distance `lda` from the element in the same row (respectively column) and the previous column (respectively row).

Before entry in any BLAS function using a triangular matrix,

- If `upper_lower = uplo::upper`, the leading `n` by `n` upper triangular part of the array `a` must contain the upper triangular part of the matrix `A`. The strictly lower triangular part of the array `a` is not referenced. In other words, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \dots & A_{1n} \\ * & A_{22} & A_{23} & \dots & A_{2n} \\ * & * & A_{33} & \dots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ * & * & * & \dots & A_{nn} \end{bmatrix}$$

  is stored in memory as the array

  - For column major layout,

$$a=[\underbrace{\underbrace{A_{11}, *, ..., *}_{\text{lda}}, \underbrace{A_{12}, A_{22}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{1n}, A_{2n}, A_{3n}, ..., A_{nn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

  - For row major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{12}, A_{13}, ..., A_{1n}, *, ..., *}_{\text{lda}}, \underbrace{*, A_{22}, A_{23}, ..., A_{2n}, *, ..., *}_{\text{lda}}, ..., \underbrace{*, ..., *, A_{nn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

- If `upper_lower = uplo::lower`, the leading `n` by `n` lower triangular part of the array `a` must contain the lower triangular part of the matrix `A`. The strictly upper triangular part of the array `a` is not referenced. That is, the matrix

$$A = \begin{bmatrix} A_{11} & * & * & \dots & * \\ A_{21} & A_{22} & * & \dots & * \\ A_{31} & A_{32} & A_{33} & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & A_{n3} & \dots & A_{nn} \end{bmatrix}$$

  is stored in memory as the array

  - For column major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{21}, A_{31}, .., A_{n1}, *, ..., *}_{\text{lda}}, \underbrace{*, A_{22}, A_{32}, ..., A_{n2}, *, ..., *}_{\text{lda}}, ..., \underbrace{*, ..., *, A_{nn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

  - For row major layout,

$$a=[\underbrace{\underbrace{A_{11}, *, ..., *}_{\text{lda}}, \underbrace{A_{21}, A_{22}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{n1}, A_{n2}, A_{n3}, ..., A_{nn}, *, ..., *}_{\text{lda}}}_{\text{lda x n}}]$$

Band Matrix

A general band matrix `A` of `m` rows and `n` columns with `kl` sub-diagonals, `ku` super-diagonals, and leading dimension `lda` is represented as a one dimensional array `a` of a size of at least `lda * n` (respectively `lda * m`) if column (respectively row) major layout is used.

Before entry in any BLAS function using a general band matrix, the leading (`kl + ku + 1`) by `n` (respectively `m`) part of the array `a` must contain the matrix `A`. This matrix must be supplied column-by-column (respectively row-by-row), with the main diagonal of the matrix in row `ku` (respectively `kl`) of the array (0-based indexing), the first super-diagonal starting at position 1 (respectively 0) in row (`ku - 1`) (respectively column (`kl + 1`)), the first sub-diagonal starting at position 0 (respectively 1) in row (`ku + 1`) (respectively column (`kl - 1`)), and so on. Elements in the array `a` that do not correspond to elements in the band matrix (such as the top left `ku` by `ku` triangle) are not referenced.

Visually, the matrix `A`

$$A = \begin{bmatrix}
A_{11} & A_{12} & A_{13} & \dots & A_{1,ku+1} & * & \dots & \dots & \dots & \dots & \dots & * \\
A_{21} & A_{22} & A_{23} & A_{24} & \dots & A_{2,ku+2} & * & \dots & \dots & \dots & \dots & * \\
A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & \dots & A_{3,ku+3} & * & \dots & \dots & \dots & * \\
\vdots & A_{42} & A_{43} & \ddots & \ddots & \ddots & \ddots & \ddots & * & \dots & \dots & \vdots \\
A_{kl+1,1} & \vdots & A_{53} & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * & \dots & \vdots \\
* & A_{kl+2,2} & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & * & A_{kl+3,3} & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * \\
\vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{n-ku,n} \\
\vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{m-2,n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{m-1,n} \\
* & * & * & \dots & \dots & \dots & * & A_{m,m-kl} & \dots & A_{m,n-2} & A_{m,n-1} & A_{m,n}
\end{bmatrix}$$

is stored in memory as an array

- For column major layout,

$$a=[\underbrace{*, \dots, *}_{\text{ku}}, A_{11}, A_{12}, \dots, A_{min(kl+1,m),1}, *, \dots, *, \underbrace{*, \dots, *}_{\text{ku-1}}, A_{max(1,2-ku),2}, \dots, A_{min(kl+2,m),2}, *, \dots *, \dots, \underbrace{*, \dots, *}_{\text{max(0,ku-n+1)}}, A_{max(1,n-ku),n}, \dots,$$

lda, lda, lda, lda x n

- For row major layout,

$$a=[\underbrace{*, \dots, *}_{\text{kl}}, A_{11}, A_{12}, \dots, A_{1,min(ku+1,n)}, *, \dots, *, \underbrace{*, \dots, *}_{\text{kl-1}}, A_{2,max(1,2-kl)}, \dots, A_{2,min(ku+2,n)}, *, \dots *, \dots, \underbrace{*, \dots, *}_{\text{max(0,kl-m+1)}}, A_{m,max(1,m-kl)}, \dots,$$

lda, lda, lda, lda x m

The following program segment transfers a band matrix from conventional full matrix storage (variable `matrix`, with leading dimension `ldm`) to band storage (variable `a`, with leading dimension `lda`):

- Using matrices stored with column major layout,

```
for (j = 0; j < n; j++) {
    k = ku - j;
    for (i = max(0, j - ku); i < min(m, j + kl + 1); i++) {
        a[(k + i) + j * lda] = matrix[i + j * ldm];
    }
}
```

- Using matrices stored with row major layout,

```
for (i = 0; i < n; i++) {
    k = kl - i;
    for (j = max(0, i - kl); j < min(n, i + ku + 1); j++) {
        a[(k + j) + i * lda] = matrix[j + i * ldm];
    }
}
```

Triangular Band Matrix

A triangular band matrix `A` of `n` rows and `n` columns with `k` sub/super-diagonals and leading dimension `lda` is represented as a one dimensional array `a` of size at least `lda * n`.

Before entry in any BLAS function using a triangular band matrix,

- If `upper_lower = uplo::upper`, the leading (`k + 1`) by `n` part of the array `a` must contain the upper triangular band part of the matrix `A`. When using column major layout, this matrix must be supplied column-by-column (respectively row-by-row) with the main diagonal of the matrix in row (`k`) (respectively column 0) of the array, the first super-diagonal starting at position 1 (respectively 0) in row (`k - 1`) (respectively column 1), and so on. Elements in the array `a` that do not correspond to elements in the triangular band matrix (such as the top left `k` by `k` triangle) are not referenced.

  Visually, the matrix

$$A = \begin{bmatrix}
A_{11} & A_{12} & A_{13} & \dots & A_{1,k+1} & * & \dots & \dots & \dots & \dots & \dots & * \\
* & A_{22} & A_{23} & A_{24} & \dots & A_{2,k+2} & * & \dots & \dots & \dots & \dots & * \\
\vdots & * & A_{33} & A_{34} & A_{35} & \dots & A_{3,k+3} & * & \dots & \dots & \dots & * \\
\vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & * & \dots & \dots & \vdots \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * & \dots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{n-k,n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & A_{n-2,n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & A_{n-1,n} \\
* & * & * & \dots & \dots & \dots & \dots & \dots & \dots & \dots & * & A_{n,n}
\end{bmatrix}$$

  is stored as an array

- For column major layout,

$$a=[\underbrace{\underbrace{*, \dots, *}_{\text{ku}}}_{\text{lda}}, \underbrace{\underbrace{A_{11}, *, \dots, *, *, \dots, *}_{\text{ku-1}}, \underbrace{A_{max(1,2-k),2}, \dots, A_{2,2}, *, \dots *, \dots,}_{\text{lda}} \underbrace{\underbrace{*, \dots, *}_{\text{max(0,k-n+1)}}}_{\text{lda}}, A_{max(1,n-k),n}, \dots, A_{n,n}, *, \dots *]}_{\text{lda x n}}$$

- For row major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{21}, \dots, A_{min(k+1,n),1}, *, \dots, *}_{\text{lda}}, \underbrace{A_{2,2}, \dots, A_{min(k+2,n),2}, *, \dots, *}_{\text{lda}}, \dots, \underbrace{A_{n,n}, *, \dots *}_{\text{lda}}]}_{\text{lda x n}}$$

The following program segment transfers a band matrix from conventional full matrix storage (variable `matrix`, with leading dimension `ldm`) to band storage (variable `a`, with leading dimension `lda`):

- Using matrices stored with column major layout,

```
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max(0, j - k); i <= j; i++) {
        a[(m + i) + j * lda] = matrix[i + j * ldm];
    }
}
```

- Using matrices stored with column major layout,

```
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < min(n, i + k + 1); j++) {
        a[(m + j) + i * lda] = matrix[j + i * ldm];
    }
}
```

- If `upper_lower = uplo::lower`, the leading (`k` + 1) by `n` part of the array `a` must contain the upper triangular band part of the matrix `A`. This matrix must be supplied column-by-column with the main diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. Elements in the array `a` that do not correspond to elements in the triangular band matrix (such as the bottom right `k` by `k` triangle) are not referenced.

  That is, the matrix

$$A = \begin{bmatrix} A_{11} & * & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & * \\ A_{21} & A_{22} & * & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & * \\ A_{31} & A_{32} & A_{33} & * & \dots & \dots & \dots & \dots & \dots & \dots & \dots & * \\ \vdots & A_{42} & A_{43} & \ddots & \ddots & \dots & \dots & \dots & \dots & \dots & \dots & \vdots \\ A_{k+1,1} & \vdots & A_{53} & \ddots & \ddots & \ddots & \dots & \dots & \dots & \dots & \dots & \vdots \\ * & A_{k+2,2} & \vdots & \ddots & \ddots & \ddots & \ddots & \dots & \dots & \dots & \dots & \vdots \\ \vdots & * & A_{k+3,3} & \ddots & \ddots & \ddots & \ddots & \ddots & \dots & \dots & \dots & \vdots \\ \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \dots & \dots & \vdots \\ \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \dots & \vdots \\ \vdots & \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * \\ * & * & * & \dots & \dots & \dots & * & A_{n,n-k} & \dots & A_{n,n-2} & A_{n,n-1} & A_{n,n} \end{bmatrix}$$

  is stored as the array

- For column major layout,

$$a=[\underbrace{\underbrace{A_{11}, A_{21}, ..., A_{min(k+1,n),1}, *, ..., *}_{\text{lda}}, \underbrace{A_{2,2}, ..., A_{min(k+2,n),2}, *, ..., *}_{\text{lda}}, ..., \underbrace{A_{n,n}, *, ...*}_{\text{lda}}}_{\text{lda x n}}]$$

- For row major layout,

$$a=[\underbrace{\underbrace{\underbrace{*, ..., *}_{\text{k}}, A_{11}, *, ..., *}_{\text{lda}}, \underbrace{\underbrace{*, ..., *}_{\text{k-1}}, A_{max(1,2-k),2}, ..., A_{2,2}, *, ...*}_{\text{lda}}, ..., \underbrace{\underbrace{*, ..., *}_{\text{max(0,k-n+1)}}, A_{max(1,n-k),n}, ..., A_{n,n}, *, ...*}_{\text{lda}}}_{\text{lda x n}}]$$

The following program segment transfers a band matrix from conventional full matrix storage (variable `matrix`, with leading dimension `ldm`) to band storage (variable `a`, with leading dimension `lda`):

- Using matrices stored with column major layout,

```
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m + i) + j * lda] = matrix[i + j * ldm];
    }
}
```

- Using matrices stored with row major layout,

```
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i - k); j <= i; j++) {
        a[(m + j) + i * lda] = matrix[j + i * ldm];
    }
}
```

Packed Triangular Matrix

A triangular matrix `A` of `n` rows and `n` columns is represented in packed format as a one dimensional array `a` of size at least (`n`*(`n` + 1))/2. All elements in the upper or lower part of the matrix `A` are stored contiguously in the array `a`.

Before entry in any BLAS function using a triangular packed matrix,

- If `upper_lower = uplo::upper`, if column (respectively row) major layout is used, the first (`n`*(`n` + 1))/2 elements in the array `a` must contain the upper triangular part of the matrix `A` packed sequentially, column by column (respectively row by row) so that `a`[0] contains A_{11}, `a`[1] and `a`[2] contain A_{12} and A_{22} (respectively A_{13}) respectively, and so on. Hence, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \dots & A_{1n} \\ * & A_{22} & A_{23} & \dots & A_{2n} \\ * & * & A_{33} & \dots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ * & * & * & \dots & A_{nn} \end{bmatrix}$$

  is stored as the array

  - For column major layout,

$$a=[A_{11},A_{12},A_{22},A_{13},A_{23},A_{33},\dots,A_{(n-1),n},A_{nn}]$$

  - For row major layout,

$$a=[A_{11},A_{12},A_{13},\dots,A_{1n},A_{22},A_{23},\dots,A_{2n},\dots,A_{(n-1),(n-1)},A_{(n-1),n},A_{nn}]$$

- If `upper_lower = uplo::lower`, if column (respectively row) major layout is used, the first (`n`*(`n` + 1))/2 elements in the array `a` must contain the lower triangular part of the matrix `A` packed sequentially, column by column (row by row) so that `a`[0] contains A_{11}, `a`[1] and `a`[2] contain A_{21} and A_{31} (respectively A_{22}) respectively, and so on. The matrix

$$A = \begin{bmatrix} A_{11} & * & * & \dots & * \\ A_{21} & A_{22} & * & \dots & * \\ A_{31} & A_{32} & A_{33} & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & A_{n3} & \dots & A_{nn} \end{bmatrix}$$

is stored as the array

- For column major layout,

$$a=[A_{11},A_{21},A_{31},...,A_{n1},A_{22},A_{32},...,A_{n2},...,A_{(n-1),(n-1)},A_{n,(n-1)},A_{nn}]$$

- For row major layout,

$$a=[A_{11},A_{21},A_{22},A_{31},A_{32},A_{33},...,A_{n,(n-1)},A_{nn}]$$

Vector

A vector `X` of `n` elements with increment `incx` is represented as a one dimensional array `x` of size at least (1 + (`n` - 1) * abs(`incx`)).

Visually, the vector

$$X = (X_1, X_2, X_3, ..., X_n)$$

is stored in memory as an array

$$x=[\underbrace{\underbrace{X_1, *, ..., *}_{\text{incx}}, \underbrace{X_2, *, ..., *}_{\text{incx}}, ..., \underbrace{X_{n-1}, *, ..., *}_{\text{incx}}, X_n}_{\text{1 + (n-1) x incx}}] \quad if\ incx > 0$$

$$x=[\underbrace{\underbrace{X_n, *, ..., *}_{\text{|incx|}}, \underbrace{X_{n-1}, *, ..., *}_{\text{|incx|}}, ..., \underbrace{X_2, *, ..., *}_{\text{|incx|}}, X_1}_{\text{1 + (1-n) x incx}}] \quad if\ incx < 0$$

Parent topic: *Dense Linear Algebra*

BLAS Routines

oneMKL provides DPC++ interfaces to the Basic Linear Algebra Subprograms (BLAS) routines (Level1, Level2, Level3), as well as several BLAS-like extension routines.

BLAS Level 1 Routines

BLAS Level 1 includes routines which perform vector-vector operations as described in the following table.

Routines	Description
asum	Sum of vector magnitudes
axpy	Scalar-vector product
copy	Copy vector
dot	Dot product
sdsdot	Dot product with double precision
dotc	Dot product conjugated
dotu	Dot product unconjugated
nrm2	Vector 2-norm (Euclidean norm)
rot	Plane rotation of points
rotg	Generate Givens rotation of points
rotm	Modified Givens plane rotation of points
rotmg	Generate modified Givens plane rotation of points
scal	Vector-scalar product
swap	Vector-vector swap
iamax	Index of the maximum absolute value element of a vector
iamin	Index of the minimum absolute value element of a vector

asum

Computes the sum of magnitudes of the vector elements.

Description

The `asum` routine computes the sum of the magnitudes of elements of a real vector, or the sum of magnitudes of the real and imaginary parts of elements of a complex vector:

$$result = \sum_{i=1}^{n}(|Re(x_i)| + |Im(x_i)|)$$

where `x` is a vector with `n` elements.

`asum` supports the following precisions for data:

T	T_res
`float`	`float`
`double`	`double`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

asum (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void asum(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
```

(continues on next page)

(continued from previous page)

```
                sycl::buffer<T_res,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void asum(sycl::queue &queue,
                std::int64_t n,
                sycl::buffer<T,1> &x,
                std::int64_t incx,
                sycl::buffer<T_res,1> &result)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

result Buffer where the scalar result is stored (the sum of magnitudes of the real and imaginary parts of all elements of the vector).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

asum (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event asum(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     T_res *result,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event asum(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     T_res *result,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Pointer to input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result Pointer to the output matrix where the scalar result is stored (the sum of magnitudes of the real and imaginary parts of all elements of the vector).

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

axpy

Computes a vector-scalar product and adds the result to a vector.

Description

The `axpy` routines compute a scalar-vector product and add the result to a vector:

$$y \leftarrow alpha * x + y$$

where:

`x` and `y` are vectors of `n` elements,

`alpha` is a scalar.

`axpy` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

axpy (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void axpy(sycl::queue &queue,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void axpy(sycl::queue &queue,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

alpha Specifies the scalar alpha.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

y Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

axpy (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event axpy(sycl::queue &queue,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event axpy(sycl::queue &queue,
                     std::int64_t n,
                     T alpha,
                     const T *x,
```

(continues on next page)

(continued from previous page)

```
                        std::int64_t incx,
                        T *y,
                        std::int64_t incy,
                        const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

alpha Specifies the scalar alpha.

x Pointer to the input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to the input vector `y`. The array holding the vector `y` must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

copy

Copies a vector to another vector.

Description

The `copy` routines copy one vector to another:

$$y \leftarrow x$$

where `x` and `y` are vectors of n elements.

`copy` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

copy (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void copy(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void copy(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

incy Stride of vector `y`.

Output Parameters

y Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

copy (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event copy(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event copy(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Pointer to the input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

dot

Computes the dot product of two real vectors.

Description

The `dot` routines perform a dot product between two vectors:

$$result = \sum_{i=1}^{n} X_i Y_i$$

`dot` supports the following precisions for data.

T	T_res
`float`	`float`
`double`	`double`
`float`	`double`

Note

For the mixed precision version (inputs are float while result is double), the dot product is computed with double precision.

dot (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void dot(sycl::queue &queue,
             std::int64_t n,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             sycl::buffer<T_res,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void dot(sycl::queue &queue,
             std::int64_t n,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             sycl::buffer<T_res,1> &result)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vectors `x` and `y`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

result Buffer where the result (a scalar) will be stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

dot (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event dot(sycl::queue &queue,
                    std::int64_t n,
                    const T *x,
                    std::int64_t incx,
                    const T *y,
                    std::int64_t incy,
                    T_res *result,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event dot(sycl::queue &queue,
                    std::int64_t n,
                    const T *x,
                    std::int64_t incx,
                    const T *y,
                    std::int64_t incy,
                    T_res *result,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vectors `x` and `y`.

x Pointer to the input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` – 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to the input vector `y`. The array holding the vector `y` must be of size at least (1 + (`n` – 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result Pointer to where the result (a scalar) will be stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

sdsdot

Computes a vector-vector dot product with double precision.

Description

The `sdsdot` routines perform a dot product between two vectors with double precision:

$$result = sb + \sum_{i=1}^{n} X_i Y_i$$

sdsdot (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void sdsdot(sycl::queue &queue,
                std::int64_t n,
                float sb,
                sycl::buffer<float,1> &x,
                std::int64_t incx,
                sycl::buffer<float,1> &y,
                std::int64_t incy,
                sycl::buffer<float,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void sdsdot(sycl::queue &queue,
                std::int64_t n,
                float sb,
                sycl::buffer<float,1> &x,
                std::int64_t incx,
                sycl::buffer<float,1> &y,
                std::int64_t incy,
                sycl::buffer<float,1> &result)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vectors `x` and `y`.

sb Single precision scalar to be added to the dot product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incxy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

result Buffer where the result (a scalar) will be stored. If `n` < 0 the result is `sb`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

sdsdot (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event sdsdot(sycl::queue &queue,
                       std::int64_t n,
                       float sb,
                       const float *x,
                       std::int64_t incx,
                       const float *y,
                       std::int64_t incy,
                       float *result,
                       const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event sdsdot(sycl::queue &queue,
                       std::int64_t n,
                       float sb,
                       const float *x,
                       std::int64_t incx,
                       const float *y,
                       std::int64_t incy,
                       float *result,
                       const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vectors `x` and `y`.

sb Single precision scalar to be added to the dot product.

x Pointer to the input vector `x`. The array must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to the input vector `y`. The array must be of size at least (1 + (`n` - 1)*abs(`incxy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result Pointer to where the result (a scalar) will be stored. If `n` < 0 the result is `sb`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

dotc

Computes the dot product of two complex vectors, conjugating the first vector.

Description

The `dotc` routines perform a dot product between two complex vectors, conjugating the first of them:

$$result = \sum_{i=1}^{n} \overline{X_i} Y_i$$

`dotc` supports the following precisions for data.

T
`std::complex<float>`
`std::complex<double>`

dotc (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void dotc(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
```

(continues on next page)

(continued from previous page)

```
                std::int64_t incy,
                sycl::buffer<T,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void dotc(sycl::queue &queue,
                std::int64_t n,
                sycl::buffer<T,1> &x,
                std::int64_t incx,
                sycl::buffer<T,1> &y,
                std::int64_t incy,
                sycl::buffer<T,1> &result)
}
```

Input Parameters

queue The queue where the routine should be executed.

n The number of elements in vectors `x` and `y`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx The stride of vector `x`.

y Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details..

incy The stride of vector `y`.

Output Parameters

result The buffer where the result (a scalar) is stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

dotc (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void dotc(sycl::queue &queue,
              std::int64_t n,
              const T *x,
              std::int64_t incx,
              const T *y,
              std::int64_t incy,
              T *result,
              const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    void dotc(sycl::queue &queue,
              std::int64_t n,
              const T *x,
              std::int64_t incx,
              const T *y,
              std::int64_t incy,
              T *result,
              const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n The number of elements in vectors `x` and `y`.

x Pointer to input vector `x`. The array holding the input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx The stride of vector `x`.

y Pointer to input vector `y`. The array holding the input vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details..

incy The stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result The pointer to where the result (a scalar) is stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

dotu

Computes the dot product of two complex vectors.

Description

The `dotu` routines perform a dot product between two complex vectors:

$$result = \sum_{i=1}^{n} X_i Y_i$$

`dotu` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

dotu (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void dotu(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void dotu(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &result)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vectors `x` and `y`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

result Buffer where the result (a scalar) is stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

dotu (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event dotu(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
```

(continues on next page)

(continued from previous page)

```
                      std::int64_t incy,
                      T *result,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event dotu(sycl::queue &queue,
                      std::int64_t n,
                      const T *x,
                      std::int64_t incx,
                      const T *y,
                      std::int64_t incy,
                      T *result,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vectors `x` and `y`.

x Pointer to the input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to input vector `y`. The array holding input vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result Pointer to where the result (a scalar) is stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

nrm2

Computes the Euclidean norm of a vector.

Description

The `nrm2` routines computes Euclidean norm of a vector

$$result = \|x\|$$

where:

`x` is a vector of `n` elements.

`nrm2` supports the following precisions.

T	T_res
`float`	`float`
`double`	`double`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

nrm2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void nrm2(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T_res,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void nrm2(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T_res,1> &result)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

result Buffer where the Euclidean norm of the vector `x` will be stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

nrm2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event nrm2(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     T_res *result,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event nrm2(sycl::queue &queue,
                     std::int64_t n,
                     const T *x,
                     std::int64_t incx,
                     T_res *result,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result Pointer to where the Euclidean norm of the vector `x` will be stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

rot

Performs rotation of points in the plane.

Description

Given two vectors `x` and `y` of `n` elements, the `rot` routines compute four scalar-vector products and update the input vectors with the sum of two of these scalar-vector products as follow:

$$\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} c*x + s*y \\ -s*x + c*y \end{bmatrix}$$

`rot` supports the following precisions.

T	T_scalar
`float`	`float`
`double`	`double`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

rot (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void rot(sycl::queue &queue,
             std::int64_t n,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             T_scalar c,
             T_scalar s)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void rot(sycl::queue &queue,
             std::int64_t n,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             T_scalar c,
             T_scalar s)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

c Scaling factor.

s Scaling factor.

Output Parameters

x Buffer holding updated buffer `x`.

y Buffer holding updated buffer `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

rot (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event rot(sycl::queue &queue,
                    std::int64_t n,
                    T *x,
                    std::int64_t incx,
                    T *y,
                    std::int64_t incy,
                    T_scalar c,
                    T_scalar s,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event rot(sycl::queue &queue,
                    std::int64_t n,
                    T *x,
                    std::int64_t incx,
                    T *y,
                    std::int64_t incy,
                    T_scalar c,
                    T_scalar s,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to input vector `y`. The array holding input vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

c Scaling factor.

s Scaling factor.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x Pointer to the updated matrix `x`.

y Pointer to the updated matrix `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

rotg

Computes the parameters for a Givens rotation.

Description

Given the Cartesian coordinates `(a, b)` of a point, the `rotg` routines return the parameters `c`, `s`, `r`, and `z` associated with the Givens rotation. The parameters `c` and `s` define a unitary matrix such that:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

The parameter `z` is defined such that if |a| > |b|, `z` is `s`; otherwise if `c` is not 0 `z` is 1/`c`; otherwise `z` is 1.

`rotg` supports the following precisions.

T	T_res
`float`	`float`
`double`	`double`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

rotg (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void rotg(sycl::queue &queue,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &b,
              sycl::buffer<T_real,1> &c,
              sycl::buffer<T,1> &s)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void rotg(sycl::queue &queue,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &b,
              sycl::buffer<T_real,1> &c,
              sycl::buffer<T,1> &s)
}
```

Input Parameters

queue The queue where the routine should be executed

a Buffer holding the `x`-coordinate of the point.

b Buffer holding the `y`-coordinate of the point.

Output Parameters

a Buffer holding the parameter `r` associated with the Givens rotation.

b Buffer holding the parameter `z` associated with the Givens rotation.

c Buffer holding the parameter `c` associated with the Givens rotation.

s Buffer holding the parameter `s` associated with the Givens rotation.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

rotg (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event rotg(sycl::queue &queue,
                     T *a,
                     T *b,
                     T_real *c,
                     T *s,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event rotg(sycl::queue &queue,
                     T *a,
                     T *b,
                     T_real *c,
                     T *s,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed

a Pointer to the `x`-coordinate of the point.

b Pointer to the `y`-coordinate of the point.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the parameter `r` associated with the Givens rotation.

b Pointer to the parameter `z` associated with the Givens rotation.

c Pointer to the parameter `c` associated with the Givens rotation.

s Pointer to the parameter `s` associated with the Givens rotation.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

rotm

Performs modified Givens rotation of points in the plane.

Description

Given two vectors `x` and `y`, each vector element of these vectors is replaced as follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = H \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

for `i` from 1 to `n`, where `H` is a modified Givens transformation matrix.

`rotm` supports the following precisions.

T
float
double

rotm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void rotm(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &param)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void rotm(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &param)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

param Buffer holding an array of size 5.

The elements of the `param` array are:

`param[0]` contains a switch, `flag`. The other array elements `param[1-4]` contain the components of the modified Givens transformation matrix `H`: h_{11}, h_{21}, h_{12}, and h_{22}, respectively.

Depending on the values of `flag`, the components of `H` are set as follows:

`flag = -1.0`:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag = 0.0`:

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag = 1.0`:

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0`:

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

Output Parameters

x Buffer holding updated buffer `x`.

y Buffer holding updated buffer `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

rotm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event rotm(sycl::queue &queue,
                     std::int64_t n,
                     T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     T *param,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event rotm(sycl::queue &queue,
                     std::int64_t n,
                     T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     T *param,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Pointer to the input vector `x`. The array holding the vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

yparam Pointer to the input vector `y`. The array holding the vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

param Buffer holding an array of size 5.

The elements of the `param` array are:

`param[0]` contains a switch, `flag`. The other array elements `param[1-4]` contain the components of the modified Givens transformation matrix `H`: h_{11}, h_{21}, h_{12}, and h_{22}, respectively.

Depending on the values of `flag`, the components of `H` are set as follows:

`flag = -1.0`:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag = 0.0`:

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag = 1.0`:

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0`:

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x Pointer to the updated array `x`.

y Pointer to the updated array `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

rotmg

Computes the parameters for a modified Givens rotation.

Description

Given Cartesian coordinates (`x1`, `y1`) of an input vector, the `rotmg` routines compute the components of a modified Givens transformation matrix `H` that zeros the `y`-component of the resulting vector:

$$\begin{bmatrix} x1 \\ 0 \end{bmatrix} = H \begin{bmatrix} x1\sqrt{d1} \\ y1\sqrt{d2} \end{bmatrix}$$

`rotmg` supports the following precisions.

T
`float`
`double`

rotmg (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void rotmg(sycl::queue &queue,
               sycl::buffer<T,1> &d1,
               sycl::buffer<T,1> &d2,
               sycl::buffer<T,1> &x1,
               sycl::buffer<T,1> &y1,
               sycl::buffer<T,1> &param)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void rotmg(sycl::queue &queue,
               sycl::buffer<T,1> &d1,
               sycl::buffer<T,1> &d2,
               sycl::buffer<T,1> &x1,
               sycl::buffer<T,1> &y1,
               sycl::buffer<T,1> &param)
}
```

Input Parameters

queue The queue where the routine should be executed.

d1 Buffer holding the scaling factor for the `x`-coordinate of the input vector.

d2 Buffer holding the scaling factor for the `y`-coordinate of the input vector.

x1 Buffer holding the `x`-coordinate of the input vector.

y1 Scalar specifying the `y`-coordinate of the input vector.

Output Parameters

d1 Buffer holding the first diagonal element of the updated matrix.

d2 Buffer holding the second diagonal element of the updated matrix.

x1 Buffer holding the `x`-coordinate of the rotated vector before scaling

param Buffer holding an array of size 5.

The elements of the `param` array are:

`param[0]` contains a switch, `flag`. The other array elements `param[1-4]` contain the components of the modified Givens transformation matrix `H`: h_{11}, h_{21}, h_{12}, and h_{22}, respectively.

Depending on the values of `flag`, the components of `H` are set as follows:

`flag = -1.0`:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag = 0.0`:

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag = 1.0`:

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0`:

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

rotmg (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event rotmg(sycl::queue &queue,
                      T *d1,
                      T *d2,
                      T *x1,
                      T *y1,
                      T *param,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event rotmg(sycl::queue &queue,
                      T *d1,
                      T *d2,
                      T *x1,
                      T *y1,
                      T *param,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

d1 Pointer to the scaling factor for the `x`-coordinate of the input vector.

d2 Pointer to the scaling factor for the `y`-coordinate of the input vector.

x1 Pointer to the `x`-coordinate of the input vector.

y1 Scalar specifying the `y`-coordinate of the input vector.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

d1 Pointer to the first diagonal element of the updated matrix.

d2 Pointer to the second diagonal element of the updated matrix.

x1 Pointer to the `x`-coordinate of the rotated vector before scaling

param Buffer holding an array of size 5.

The elements of the `param` array are:

`param[0]` contains a switch, `flag`. The other array elements `param[1-4]` contain the components of the modified Givens transformation matrix `H`: h_{11}, h_{21}, h_{12}, and h_{22}, respectively.

Depending on the values of `flag`, the components of `H` are set as follows:

`flag = -1.0`:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag = 0.0`:

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag = 1.0`:

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0`:

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

scal

Computes the product of a vector by a scalar.

Description

The `scal` routines computes a scalar-vector product:

$$x \leftarrow alpha * x$$

where:

`x` is a vector of `n` elements,

`alpha` is a scalar.

`scal` supports the following precisions.

T	T_scalar
`float`	`float`
`double`	`double`
`std::complex<float>`	`std::complex<float>`
`std::complex<double>`	`std::complex<double>`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

scal (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void scal(sycl::queue &queue,
              std::int64_t n,
              T_scalar alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void scal(sycl::queue &queue,
              std::int64_t n,
              T_scalar alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

alpha Specifies the scalar `alpha`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

x Buffer holding updated buffer `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

scal (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event scal(sycl::queue &queue,
                     std::int64_t n,
                     T_scalar alpha,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event scal(sycl::queue &queue,
                     std::int64_t n,
                     T_scalar alpha,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

alpha Specifies the scalar `alpha`.

x Pointer to the input vector `x`. The array must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

x Pointer to the updated array `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

swap

Swaps a vector with another vector.

Description

Given two vectors of `n` elements, `x` and `y`, the `swap` routines return vectors `y` and `x` swapped, each replacing the other.

$$\begin{bmatrix} y \\ x \end{bmatrix} \leftarrow \begin{bmatrix} x \\ y \end{bmatrix}$$

`swap` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

swap (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void swap(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void swap(sycl::queue &queue,
              std::int64_t n,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

x Buffer holding updated buffer `x`, that is, the input vector `y`.

y Buffer holding updated buffer `y`, that is, the input vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

swap (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event swap(sycl::queue &queue,
                     std::int64_t n,
                     T *x,
                     std::int64_t incx,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event swap(sycl::queue &queue,
                     std::int64_t n,
                     T *x,
                     std::int64_t incx,
                     T *y,
```

(continues on next page)

(continued from previous page)

```
                        std::int64_t incy,
                        const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Pointer to the input vector `x`. The array must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to the input vector `y`. The array must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x Pointer to the updated array `x`, that is, the input vector `y`.

y Pointer to the updated array `y`, that is, the input vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

iamax

Finds the index of the element with the largest absolute value in a vector.

Description

The `iamax` routines return an index `i` such that `x[i]` has the maximum absolute value of all elements in vector `x` (real variants), or such that (|Re(`x[i]`)| + |Im(`x[i]`)|) is maximal (complex variants).

If either `n` or `incx` are not positive, the routine returns `0`.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

If the vector contains `NaN` values, then the routine returns the index of the first `NaN`.

`iamax` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std:complex<double>`

Note

The index is zero-based.

iamax (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void iamax(sycl::queue &queue,
               std::int64_t n,
               sycl::buffer<T,
               1> &x,
               std::int64_t incx,
               sycl::buffer<std::int64_t,
               1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void iamax(sycl::queue &queue,
               std::int64_t n,
               sycl::buffer<T,
               1> &x,
               std::int64_t incx,
               sycl::buffer<std::int64_t,
               1> &result)
}
```

Input Parameters

queue The queue where the routine should be executed.

n The number of elements in vector `x`.

x The buffer that holds the input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx The stride of vector `x`.

Output Parameters

result The buffer where the zero-based index `i` of the maximal element is stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

iamax (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event iamax(sycl::queue &queue,
                      std::int64_t n,
                      const T *x,
                      std::int64_t incx,
                      T_res *result,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event iamax(sycl::queue &queue,
                      std::int64_t n,
                      const T *x,
                      std::int64_t incx,
                      T_res *result,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n The number of elements in vector `x`.

x The pointer to the input vector `x`. The array holding the input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx The stride of vector `x`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

result The pointer to where the zero-based index `i` of the maximal element is stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

iamin

Finds the index of the element with the smallest absolute value.

Description

The `iamin` routines return an index `i` such that `x[i]` has the minimum absolute value of all elements in vector `x` (real variants), or such that (|Re(`x[i]`)| + |Im(`x[i]`)|) is minimal (complex variants).

If either `n` or `incx` are not positive, the routine returns `0`.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

If the vector contains `NaN` values, then the routine returns the index of the first `NaN`.

`iamin` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Note

The index is zero-based.

iamin (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void iamin(sycl::queue &queue,
               std::int64_t n,
               sycl::buffer<T,1> &x,
               std::int64_t incx,
               sycl::buffer<std::int64_t,1> &result)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void iamin(sycl::queue &queue,
               std::int64_t n,
               sycl::buffer<T,1> &x,
               std::int64_t incx,
               sycl::buffer<std::int64_t,1> &result)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector x.

Output Parameters

result Buffer where the zero-based index `i` of the minimum element will be stored.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

iamin (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event iamin(sycl::queue &queue,
                      std::int64_t n,
                      const T *x,
                      std::int64_t incx,
                      T_res *result,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event iamin(sycl::queue &queue,
                      std::int64_t n,
                      const T *x,
                      std::int64_t incx,
                      T_res *result,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in vector `x`.

x The pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector x.

Output Parameters

result Pointer to where the zero-based index `i` of the minimum element will be stored.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 1 Routines*

Parent topic: *BLAS Routines*

BLAS Level 2 Routines

BLAS Level 2 includes routines which perform matrix-vector operations as described in the following table.

Routines	Description
gbmv	Matrix-vector product using a general band matrix
gemv	Matrix-vector product using a general matrix
ger	Rank-1 update of a general matrix
gerc	Rank-1 update of a conjugated general matrix
geru	Rank-1 update of a general matrix, unconjugated
hbmv	Matrix-vector product using a Hermitian band matrix
hemv	Matrix-vector product using a Hermitian matrix
her	Rank-1 update of a Hermitian matrix
her2	Rank-2 update of a Hermitian matrix
hpmv	Matrix-vector product using a Hermitian packed matrix
hpr	Rank-1 update of a Hermitian packed matrix
hpr2	Rank-2 update of a Hermitian packed matrix
sbmv	Matrix-vector product using symmetric band matrix
spmv	Matrix-vector product using a symmetric packed matrix
spr	Rank-1 update of a symmetric packed matrix
spr2	Rank-2 update of a symmetric packed matrix
symv	Matrix-vector product using a symmetric matrix
syr	Rank-1 update of a symmetric matrix
syr2	Rank-2 update of a symmetric matrix
tbmv	Matrix-vector product using a triangular band matrix
tbsv	Solution of a linear system of equations with a triangular band matrix
tpmv	Matrix-vector product using a triangular packed matrix
tpsv	Solution of a linear system of equations with a triangular packed matrix
trmv	Matrix-vector product using a triangular matrix
trsv	Solution of a linear system of equations with a triangular matrix

gbmv

Computes a matrix-vector product with a general band matrix.

Description

The `gbmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a general band matrix. The operation is defined as

$$y \leftarrow alpha * op(A) * x + beta * y$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = A^T, or op(`A`) = A^H,

`alpha` and `beta` are scalars,

`A` is an `m`-by-`n` matrix with `kl` sub-diagonals and `ku` super-diagonals,

`x` and `y` are vectors.

`gbmv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

gbmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gbmv(sycl::queue &queue,
              onemkl::transpose trans,
              std::int64_t m,
              std::int64_t n,
              std::int64_t kl,
              std::int64_t ku,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gbmv(sycl::queue &queue,
              onemkl::transpose trans,
              std::int64_t m,
              std::int64_t n,
              std::int64_t kl,
              std::int64_t ku,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

m Number of rows of `A`. Must be at least zero.

n Number of columns of `A`. Must be at least zero.

kl Number of sub-diagonals of the matrix `A`. Must be at least zero.

ku Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Buffer holding input matrix `A`. Must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`kl` + `ku` + 1), and positive.

x Buffer holding input vector `x`. The length `len` of vector `x` is `n` if `A` is not transposed, and `m` if `A` is transposed. The buffer must be of size at least (1 + (`len` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Buffer holding input/output vector `y`. The length `len` of vector `y` is `m`, if `A` is not transposed, and `n` if `A` is transposed. The buffer must be of size at least (1 + (`len` - 1)*abs(`incy`)) where `len` is this length. See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

y Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gbmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gbmv(sycl::queue &queue,
                     onemkl::transpose trans,
                     std::int64_t m,
                     std::int64_t n,
                     std::int64_t kl,
                     std::int64_t ku,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gbmv(sycl::queue &queue,
                     onemkl::transpose trans,
                     std::int64_t m,
                     std::int64_t n,
                     std::int64_t kl,
                     std::int64_t ku,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

m Number of rows of `A`. Must be at least zero.

n Number of columns of `A`. Must be at least zero.

kl Number of sub-diagonals of the matrix `A`. Must be at least zero.

ku Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`kl` + `ku` + 1), and positive.

x Pointer to input vector `x`. The length `len` of vector `x` is `n` if `A` is not transposed, and `m` if `A` is transposed. The array holding input vector `x` must be of size at least (1 + (`len` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Pointer to input/output vector `y`. The length `len` of vector `y` is `m`, if `A` is not transposed, and `n` if `A` is transposed. The array holding input/output vector `y` must be of size at least (1 + (`len` - 1)*abs(`incy`)) where `len` is this length. See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

gemv

Computes a matrix-vector product using a general matrix.

Description

The `gemv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a general matrix. The operation is defined as:

$$y \leftarrow alpha * op(A) * x + beta * y$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = A^T, or op(`A`) = A^H,

alpha and beta are scalars,

A is an m-by-n matrix, and x, y are vectors.

gemv supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

gemv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemv(sycl::queue &queue,
              onemkl::transpose trans,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemv(sycl::queue &queue,
              onemkl::transpose trans,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

trans Specifies `op(A)`, the transposition operation applied to `A`.

m Specifies the number of rows of the matrix `A`. The value of `m` must be at least zero.

n Specifies the number of columns of the matrix `A`. The value of `n` must be at least zero.

alpha Scaling factor for the matrix-vector product.

a The buffer holding the input matrix `A`. Must have a size of at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

x Buffer holding input vector `x`. The length `len` of vector `x` is `n` if `A` is not transposed, and `m` if `A` is transposed. The buffer must be of size at least (1 + (`len` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx The stride of vector `x`.

beta The scaling factor for vector `y`.

y Buffer holding input/output vector `y`. The length `len` of vector `y` is `m`, if `A` is not transposed, and `n` if `A` is transposed. The buffer must be of size at least (1 + (`len` - 1)*abs(`incy`)) where `len` is this length. See *Matrix Storage* for more details.

incy The stride of vector `y`.

Output Parameters

y The buffer holding updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemv(sycl::queue &queue,
                     onemkl::transpose trans,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemv(sycl::queue &queue,
                     onemkl::transpose trans,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

trans Specifies `op(A)`, the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

m Specifies the number of rows of the matrix `A`. The value of `m` must be at least zero.

n Specifies the number of columns of the matrix `A`. The value of `n` must be at least zero.

alpha Scaling factor for the matrix-vector product.

a The pointer to the input matrix `A`. Must have a size of at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

x Pointer to the input vector `x`. The length `len` of vector `x` is `n` if `A` is not transposed, and `m` if `A` is transposed. The array holding vector `x` must be of size at least (1 + (`len` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx The stride of vector `x`.

beta The scaling factor for vector `y`.

y Pointer to input/output vector `y`. The length `len` of vector `y` is `m`, if `A` is not transposed, and `n` if `A` is transposed. The array holding input/output vector `y` must be of size at least (1 + (`len` - 1)*abs(`incy`)) where `len` is this length. See *Matrix Storage* for more details.

incy The stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y The pointer to updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

ger

Computes a rank-1 update of a general matrix.

Description

The `ger` routines compute a scalar-vector-vector product and add the result to a general matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^T + A$$

where:

`alpha` is scalar,

`A` is an `m`-by-`n` matrix,

`x` is a vector of length `m`,

`y` is a vector of length `n`.

`ger` supports the following precisions.

T
float
double

ger (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void ger(sycl::queue &queue,
             std::int64_t m,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void ger(sycl::queue &queue,
             std::int64_t m,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &y,
             std::int64_t incy,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

Input Parameters

queue The queue where the routine should be executed.

m Number of rows of `A`. Must be at least zero.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Buffer holding input matrix `A`. Must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

Output Parameters

a Buffer holding the updated matrix `A`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

ger (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event ger(sycl::queue &queue,
                    std::int64_t m,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    const T *y,
                    std::int64_t incy,
                    T *a,
                    std::int64_t lda,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event ger(sycl::queue &queue,
                    std::int64_t m,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    const T *y,
                    std::int64_t incy,
                    T *a,
                    std::int64_t lda,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

m Number of rows of `A`. Must be at least zero.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Pointer to input matrix `A`. Must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated matrix `A`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

gerc

Computes a rank-1 update (conjugated) of a general complex matrix.

Description

The `gerc` routines compute a scalar-vector-vector product and add the result to a general matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^H + A$$

where:

`alpha` is a scalar,

`A` is an `m`-by-`n` matrix,

`x` is a vector of length `m`,

`y` is vector of length `n`.

`gerc` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

gerc (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gerc(sycl::queue &queue,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gerc(sycl::queue &queue,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

Input Parameters

queue The queue where the routine should be executed.

m Number of rows of `A`. Must be at least zero.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Buffer holding input matrix `A`. Must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

Output Parameters

a Buffer holding the updated matrix `A`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gerc (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gerc(sycl::queue &queue,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
```

(continues on next page)

(continued from previous page)

```
                      T *a,
                      std::int64_t lda,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gerc(sycl::queue &queue,
                      std::int64_t m,
                      std::int64_t n,
                      T alpha,
                      const T *x,
                      std::int64_t incx,
                      const T *y,
                      std::int64_t incy,
                      T *a,
                      std::int64_t lda,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

m Number of rows of `A`. Must be at least zero.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to the input vector `x`. The array holding input vector `x` must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to the input/output vector `y`. The array holding the input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Pointer to input matrix `A`. The array holding input matrix `A`must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated matrix `A`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

geru

Computes a rank-1 update (unconjugated) of a general complex matrix.

Description

The `geru` routines routines compute a scalar-vector-vector product and add the result to a general matrix. The operation is defined as

$$A \leftarrow alpha * x * y^T + A$$

where:

`alpha` is a scalar,

`A` is an `m`-by-`n` matrix,

`x` is a vector of length `m`,

`y` is a vector of length `n`.

`geru` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

geru (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void geru(sycl::queue &queue,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void geru(sycl::queue &queue,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

Input Parameters

queue The queue where the routine should be executed.

m Number of rows of `A`. Must be at least zero.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Buffer holding input matrix `A`. Must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

Output Parameters

a Buffer holding the updated matrix `A`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

geru (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event geru(sycl::queue &queue,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event geru(sycl::queue &queue,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

m Number of rows of `A`. Must be at least zero.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to the input vector `x`. The array holding input vector `x` must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n` if column major layout is used or at least `lda*m` if row major layout is used. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be positive and at least `m` if column major layout is used or at least `n` if row major layout is used.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated matrix `A`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hbmv

Computes a matrix-vector product using a Hermitian band matrix.

Description

The `hbmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a Hermitian band matrix. The operation is defined as

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` Hermitian band matrix, with `k` super-diagonals,

`x` and `y` are vectors of length `n`.

`hbmv` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

hbmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hbmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              std::int64_t k,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hbmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              std::int64_t k,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
```

(continues on next page)

(continued from previous page)

```
                T beta,
                sycl::buffer<T,1> &y,
                std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

k Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

y Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hbmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hbmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     std::int64_t k,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hbmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     std::int64_t k,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

k Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Pointer to the input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`m` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hemv

Computes a matrix-vector product using a Hermitian matrix.

Description

The `hemv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a Hermitian matrix. The operation is defined as

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` Hermitian matrix,

`x` and `y` are vectors of length `n`.

`hemv` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

hemv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hemv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hemv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether *A* is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `m`, and positive.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

y Buffer holding the updated vector y.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hemv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hemv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hemv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether *A* is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `m`, and positive.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

her

Computes a rank-1 update of a Hermitian matrix.

Description

The `her` routines compute a scalar-vector-vector product and add the result to a Hermitian matrix. The operation is defined as:

$$A \leftarrow alpha * x * x^H + A$$

where:

`alpha` is scalar,

`A` is an `n`-by-`n` Hermitian matrix,

`x` is a vector of length `n`.

`her` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

her (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void her(sycl::queue &queue,
             onemkl::uplo upper_lower,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void her(sycl::queue &queue,
             onemkl::uplo upper_lower,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

Output Parameters

a Buffer holding the updated upper triangular part of the Hermitian matrix `A` if `upper_lower = upper` or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

The imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

her (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event her(sycl::queue &queue,
                    onemkl::uplo upper_lower,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    std::int64_t lda,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event her(sycl::queue &queue,
                    onemkl::uplo upper_lower,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    std::int64_t lda,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether *A* is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper` or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

The imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

her2

Computes a rank-2 update of a Hermitian matrix.

Description

The `her2` routines compute two scalar-vector-vector products and add them to a Hermitian matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^H + conjg(alpha) * y * x^H + A$$

where:

`alpha` is a scalar,

`A` is an `n`-by-`n` Hermitian matrix,

`x` and `y` are vectors or length `n`.

`her2` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

her2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void her2(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void her2(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
```

(continues on next page)

(continued from previous page)

```
                std::int64_t incy,
                sycl::buffer<T,1> &a,
                std::int64_t lda)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

Output Parameters

a Buffer holding the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

The imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

her2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event her2(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event her2(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

The imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hpmv

Computes a matrix-vector product using a Hermitian packed matrix.

Description

The `hpmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a Hermitian packed matrix. The operation is defined as

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` Hermitian matrix supplied in packed form,

`x` and `y` are vectors of length `n`.

`hpmv` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

hpmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hpmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hpmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Buffer holding input matrix `A`. Must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

y Buffer holding the updated vector y.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hpmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hpmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hpmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `(n*(n+1))/2`. See *Matrix Storage* for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least `(1 + (n - 1)*abs(incx))`. See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least `(1 + (n - 1)*abs(incy))`. See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hpr

Computes a rank-1 update of a Hermitian packed matrix.

Description

The `hpr` routines compute a scalar-vector-vector product and add the result to a Hermitian packed matrix. The operation is defined as

$$A \leftarrow alpha * x * x^H + A$$

where:

`alpha` is scalar,

`A` is an `n`-by-`n` Hermitian matrix, supplied in packed form,

`x` is a vector of length `n`.

`hpr` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

hpr (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hpr(sycl::queue &queue,
             onemkl::uplo upper_lower,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hpr(sycl::queue &queue,
             onemkl::uplo upper_lower,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

a Buffer holding input matrix `A`. Must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

The imaginary part of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

a Buffer holding the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

The imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hpr (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hpr(sycl::queue &queue,
                    onemkl::uplo upper_lower,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hpr(sycl::queue &queue,
                    onemkl::uplo upper_lower,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

The imaginary part of the diagonal elements need not be set and are assumed to be zero.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

The imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

hpr2

Performs a rank-2 update of a Hermitian packed matrix.

Description

The `hpr2` routines compute two scalar-vector-vector products and add them to a Hermitian packed matrix. The operation is defined as

$$A \leftarrow alpha * x * y^H + conjg(alpha) * y * x^H + A$$

where:

`alpha` is a scalar,

`A` is an `n`-by-`n` Hermitian matrix, supplied in packed form,

`x` and `y` are vectors of length `n`.

`hpr2` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

hpr2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hpr2(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hpr2(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Buffer holding input matrix `A`. Must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

a Buffer holding the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

The imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hpr2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hpr2(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
```

(continues on next page)

(continued from previous page)

```
                    T *a,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hpr2(sycl::queue &queue,
                    onemkl::uplo upper_lower,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    const T *y,
                    std::int64_t incy,
                    T *a,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated upper triangular part of the Hermitian matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the Hermitian matrix `A` if `upper_lower=lower`.

The imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

sbmv

Computes a matrix-vector product with a symmetric band matrix.

Description

The `sbmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a symmetric band matrix. The operation is defined as:

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` symmetric matrix with `k` super-diagonals,

`x` and `y` are vectors of length `n`.

`sbmv` supports the following precisions.

T
`float`
`double`

sbmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void sbmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
                std::int64_t k,
                T alpha,
                sycl::buffer<T,1> &a,
                std::int64_t lda,
                sycl::buffer<T,1> &x,
                std::int64_t incx,
                T beta,
                sycl::buffer<T,1> &y,
                std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void sbmv(sycl::queue &queue,
                onemkl::uplo upper_lower,
                std::int64_t n,
                std::int64_t k,
                T alpha,
                sycl::buffer<T,1> &a,
                std::int64_t lda,
                sycl::buffer<T,1> &x,
                std::int64_t incx,
                T beta,
                sycl::buffer<T,1> &y,
                std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

k Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

y Buffer holding the updated vector y.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

sbmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event sbmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     std::int64_t k,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event sbmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     std::int64_t k,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

k Number of super-diagonals of the matrix `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

spmv

Computes a matrix-vector product with a symmetric packed matrix.

Description

The `spmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a symmetric packed matrix. The operation is defined as:

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` symmetric matrix, supplied in packed form,

`x` and `y` are vectors of length `n`.

`spmv` supports the following precisions.

T
float
double

spmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void spmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void spmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Buffer holding input matrix `A`. Must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

y Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

spmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event spmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
```

(continues on next page)

(continued from previous page)

```
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event spmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

beta Scaling factor for vector `y`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

spr

Performs a rank-1 update of a symmetric packed matrix.

Description

The `spr` routines compute a scalar-vector-vector product and add the result to a symmetric packed matrix. The operation is defined as:

$$A \leftarrow alpha * x * x^T + A$$

where:

`alpha` is scalar,

`A` is an `n`-by-`n` symmetric matrix, supplied in packed form,

`x` is a vector of length `n`.

`spr` supports the following precisions.

T
`float`
`double`

spr (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void spr(sycl::queue &queue,
             onemkl::uplo upper_lower,
             std::std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
            T alpha,
            sycl::buffer<T,1> &x,
            std::int64_t incx,
            sycl::buffer<T,1> &a)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void spr(sycl::queue &queue,
            onemkl::uplo upper_lower,
            std::std::int64_t n,
            T alpha,
            sycl::buffer<T,1> &x,
            std::int64_t incx,
            sycl::buffer<T,1> &a)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

a Buffer holding input matrix `A`. Must have size at least (`n`*(`n` + 1))/2. See *Matrix Storage* for more details.

Output Parameters

a Buffer holding the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

spr (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event spr(sycl::queue &queue,
                    onemkl::uplo upper_lower,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event spr(sycl::queue &queue,
                    onemkl::uplo upper_lower,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n` + 1))/2. See *Matrix Storage* for more details.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

spr2

Computes a rank-2 update of a symmetric packed matrix.

Description

The `spr2` routines compute two scalar-vector-vector products and add them to a symmetric packed matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^T + alpha * y * x^T + A$$

where:

`alpha` is scalar,

`A` is an `n`-by-`n` symmetric matrix, supplied in packed form,

`x` and `y` are vectors of length `n`.

`spr` supports the following precisions.

T
`float`
`double`

spr2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void spr2(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
                T alpha,
                sycl::buffer<T,1> &x,
                std::int64_t incx,
                sycl::buffer<T,1> &y,
                std::int64_t incy,
                sycl::buffer<T,1> &a)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void spr2(sycl::queue &queue,
                onemkl::uplo upper_lower,
                std::int64_t n,
                T alpha,
                sycl::buffer<T,1> &x,
                std::int64_t incx,
                sycl::buffer<T,1> &y,
                std::int64_t incy,
                sycl::buffer<T,1> &a)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Buffer holding input matrix `A`. Must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

Output Parameters

a Buffer holding the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper` or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

spr2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event spr2(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a)
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event spr2(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`-1))/2. See *Matrix Storage* for more details.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper` or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

symv

Computes a matrix-vector product for a symmetric matrix.

Description

The `symv` routines routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a symmetric matrix. The operation is defined as:

$$y \leftarrow alpha * A * x + beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an `n`-by-`n` symmetric matrix,

`x` and `y` are vectors of length `n`.

`symv` supports the following precisions.

T
float
double

symv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void symv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void symv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              T beta,
              sycl::buffer<T,1> &y,
              std::int64_t incy)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `m`, and positive.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

Output Parameters

y Buffer holding the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

symv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event symv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event symv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *a,
                     std::int64_t lda,
                     const T *x,
                     std::int64_t incx,
                     T beta,
                     T *y,
                     std::int64_t incy,
```

(continues on next page)

(continued from previous page)

```
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `m`, and positive.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Pointer to the updated vector `y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

syr

Computes a rank-1 update of a symmetric matrix.

Description

The `syr` routines compute a scalar-vector-vector product add them and add the result to a matrix, with a symmetric matrix. The operation is defined as:

$$A \leftarrow alpha * x * x^T + A$$

where:

`alpha` is scalar,

`A` is an `n`-by-`n` symmetric matrix,

`x` is a vector of length `n`.

`syr` supports the following precisions.

T
float
double

syr (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void syr(sycl::queue &queue,
             onemkl::uplo upper_lower,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void syr(sycl::queue &queue,
             onemkl::uplo upper_lower,
             std::int64_t n,
             T alpha,
             sycl::buffer<T,1> &x,
             std::int64_t incx,
             sycl::buffer<T,1> &a,
             std::int64_t lda)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

Output Parameters

a Buffer holding the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper` or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

syr (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syr(sycl::queue &queue,
                    onemkl::uplo upper_lower,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    std::int64_t lda,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syr(sycl::queue &queue,
                    onemkl::uplo upper_lower,
                    std::int64_t n,
                    T alpha,
                    const T *x,
                    std::int64_t incx,
                    T *a,
                    std::int64_t lda,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper` or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

syr2

Computes a rank-2 update of a symmetric matrix.

Description

The `syr2` routines compute two scalar-vector-vector product add them and add the result to a matrix, with a symmetric matrix. The operation is defined as:

$$A \leftarrow alpha * x * y^T + alpha * y * x^T + A$$

where:

`alpha` is a scalar,

`A` is an `n`-by-`n` symmetric matrix,

`x` and `y` are vectors of length `n`.

`syr2` supports the following precisions.

T
float
double

syr2 (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void syr2(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
              sycl::buffer<T,1> &a,
              std::int64_t lda)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void syr2(sycl::queue &queue,
              onemkl::uplo upper_lower,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &x,
              std::int64_t incx,
              sycl::buffer<T,1> &y,
              std::int64_t incy,
```

(continues on next page)

(continued from previous page)

```
                sycl::buffer<T,1> &a,
                std::int64_t lda)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Buffer holding input/output vector `y`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Buffer holding input matrix `A`. Must have size at least `lda`*`n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

Output Parameters

a Buffer holding the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

syr2 (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syr2(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syr2(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     std::int64_t n,
                     T alpha,
                     const T *x,
                     std::int64_t incx,
                     const T *y,
                     std::int64_t incy,
                     T *a,
                     std::int64_t lda,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

n Number of columns of `A`. Must be at least zero.

alpha Scaling factor for the matrix-vector product.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

y Pointer to input/output vector `y`. The array holding input/output vector `y` must be of size at least (1 + (`n` - 1)*abs(`incy`)). See *Matrix Storage* for more details.

incy Stride of vector `y`.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

a Pointer to the updated upper triangular part of the symmetric matrix `A` if `upper_lower=upper`, or the updated lower triangular part of the symmetric matrix `A` if `upper_lower=lower`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

tbmv

Computes a matrix-vector product using a triangular band matrix.

Description

The `tbmv` routines compute a matrix-vector product with a triangular band matrix. The operation is defined as:

$$x \leftarrow op(A) * x$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = A^T, or op(`A`) = A^H,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix, with (`k` + 1) diagonals,

`x` is a vector of length `n`.

`tbmv` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

tbmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void tbmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void tbmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

k Number of sub/super-diagonals of the matrix `A`. Must be at least zero.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

x Buffer holding the updated vector x.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

tbmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event tbmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event tbmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

k Number of sub/super-diagonals of the matrix `A`. Must be at least zero.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x Pointer to the updated vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

tbsv

Solves a system of linear equations whose coefficients are in a triangular band matrix.

Description

The `tbsv` routines solve a system of linear equations whose coefficients are in a triangular band matrix. The operation is defined as:

$$op(A) * x = b$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = A^T, or op(`A`) = A^H,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix, with (`k` + 1) diagonals,

`b` and `x` are vectors of length `n`.

`tbsv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

tbsv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void tbsv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void tbsv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
```

(continues on next page)

(continued from previous page)

```
                std::int64_t incx)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

k Number of sub/super-diagonals of the matrix `A`. Must be at least zero.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

x Buffer holding the solution vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

tbsv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event tbsv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
```

(continues on next page)

(continued from previous page)

```
                      const T *a,
                      std::int64_t lda,
                      T *x,
                      std::int64_t incx,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event tbsv(sycl::queue &queue,
                      onemkl::uplo upper_lower,
                      onemkl::transpose trans,
                      onemkl::diag unit_nonunit,
                      std::int64_t n,
                      std::int64_t k,
                      const T *a,
                      std::int64_t lda,
                      T *x,
                      std::int64_t incx,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Number of rows and columns of `A`. Must be at least zero.

k Number of sub/super-diagonals of the matrix `A`. Must be at least zero.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least (`k` + 1), and positive.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x Pointer to the solution vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

tpmv

Computes a matrix-vector product using a triangular packed matrix.

Description

The `tpmv` routines compute a matrix-vector product with a triangular packed matrix. The operation is defined as:

$$x \leftarrow op(A) * x$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = A^T, or op(`A`) = A^H,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix, supplied in packed form,

`x` is a vector of length `n`.

`tpmv` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

tpmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void tpmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void tpmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

a Buffer holding input matrix `A`. Must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

x Buffer holding the updated vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

tpmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event tpmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     const T *a,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event tpmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     const T *a,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x Pointer to the updated vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

tpsv

Solves a system of linear equations whose coefficients are in a triangular packed matrix.

Description

The `tpsv` routines solve a system of linear equations whose coefficients are in a triangular packed matrix. The operation is defined as:

$$op(A) * x = b$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = A^T, or op(`A`) = A^H,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix, supplied in packed form,

`b` and `x` are vectors of length `n`.

`tpsv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

tpsv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void tpsv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void tpsv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

a Buffer holding input matrix `A`. Must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x Buffer holding the `n`-element right-hand side vector `b`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

x Buffer holding the solution vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

tpsv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event tpsv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event tpsv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least (`n`*(`n`+1))/2. See *Matrix Storage* for more details.

x Pointer to the `n`-element right-hand side vector `b`. The array holding the `n`-element right-hand side vector `b` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x Pointer to the solution vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

trmv

Computes a matrix-vector product using a triangular matrix.

Description

The `trmv` routines compute a matrix-vector product with a triangular matrix. The operation is defined as:

$$x \leftarrow op(A) * x$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = A^T, or op(`A`) = A^H,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular band matrix,

`x` is a vector of length `n`.

`trmv` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

trmv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trmv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

x Buffer holding input vector `x`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

x Buffer holding the updated vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

trmv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trmv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

x Pointer to input vector `x`. The array holding input vector `x` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x Pointer to the updated vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

trsv

Solves a system of linear equations whose coefficients are in a triangular matrix.

Description

The `trsv` routines compute a matrix-vector product with a triangular band matrix. The operation is defined as:

$$op(A) * x = b$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = A^T, or op(`A`) = A^H,

`A` is an `n`-by-`n` unit or non-unit, upper or lower triangular matrix,

`b` and `x` are vectors of length `n`.

`trsv` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

trsv (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trsv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
              std::int64_t k,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &x,
              std::int64_t incx)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trsv(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              onemkl::diag unit_nonunit,
              std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
                std::int64_t k,
                sycl::buffer<T,1> &a,
                std::int64_t lda,
                sycl::buffer<T,1> &x,
                std::int64_t incx)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

a Buffer holding input matrix `A`. Must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

x Buffer holding the `n`-element right-hand side vector `b`. The buffer must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

Output Parameters

x Buffer holding the solution vector `x`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

trsv (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trsv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trsv(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     onemkl::diag unit_nonunit,
                     std::int64_t n,
                     std::int64_t k,
                     const T *a,
                     std::int64_t lda,
                     T *x,
                     std::int64_t incx,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_nonunit Specifies whether the matrix `A` is unit triangular or not. See *oneMKL defined datatypes* for more details.

n Numbers of rows and columns of `A`. Must be at least zero.

a Pointer to input matrix `A`. The array holding input matrix `A` must have size at least `lda*n`. See *Matrix Storage* for more details.

lda Leading dimension of matrix `A`. Must be at least `n`, and positive.

x Pointer to the `n`-element right-hand side vector `b`. The array holding the `n`-element right-hand side vector `b` must be of size at least (1 + (`n` - 1)*abs(`incx`)). See *Matrix Storage* for more details.

incx Stride of vector `x`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

x Pointer to the solution vector `x`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 2 Routines*

Parent topic: *BLAS Routines*

BLAS Level 3 Routines

BLAS Level 3 includes routines which perform matrix-matrix operations as described in the following table.

Routines	Description
gemm	Computes a matrix-matrix product with general matrices.
hemm	Computes a matrix-matrix product where one input matrix is Hermitian and one is general.
herk	Performs a Hermitian rank-k update.
her2k	Performs a Hermitian rank-2k update.
symm	Computes a matrix-matrix product where one input matrix is symmetric and one matrix is general.
syrk	Performs a symmetric rank-k update.
syr2k	Performs a symmetric rank-2k update.
trmm	Computes a matrix-matrix product where one input matrix is triangular and one input matrix is general.
trsm	Solves a triangular matrix equation (forward or backward solve).

gemm

Computes a matrix-matrix product with general matrices.

Description

The gemm routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, with general matrices. The operation is defined as:

$$C \leftarrow alpha * op(A) * op(B) + beta * C$$

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

alpha and beta are scalars,

A, B and C are matrices,

op(A) is an m-by-k matrix,

op(B) is a k-by-n matrix,

C is an m-by-n matrix.

gemm supports the following precisions.

Ts	Ta	Tb	Tc
float	half	half	float
half	half	half	half
float	float	float	float
double	double	double	double
std::complex<float>	std::complex<float>	std::complex<float>	std::complex<float>
std::complex<double>	std::complex<double>	std::complex<double>	std::complex<double>

gemm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemm(sycl::queue &queue,
              onemkl::transpose transa,
              onemkl::transpose transb,
              std::int64_t m,
              std::int64_t n,
              std::int64_t k,
              Ts alpha,
              sycl::buffer<Ta,1> &a,
              std::int64_t lda,
              sycl::buffer<Tb,1> &b,
              std::int64_t ldb,
              Ts beta,
              sycl::buffer<Tc,1> &c,
              std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemm(sycl::queue &queue,
              onemkl::transpose transa,
```

(continues on next page)

(continued from previous page)

```
                onemkl::transpose transb,
                std::int64_t m,
                std::int64_t n,
                std::int64_t k,
                Ts alpha,
                sycl::buffer<Ta,1> &a,
                std::int64_t lda,
                sycl::buffer<Tb,1> &b,
                std::int64_t ldb,
                Ts beta,
                sycl::buffer<Tc,1> &c,
                std::int64_t ldc)
}
```

Input Parameters

queue The queue where the routine should be executed.

transa Specifies the form of op(`A`), the transposition operation applied to `A`.

transb Specifies the form of op(`B`), the transposition operation applied to `B`.

m Specifies the number of rows of the matrix op(`A`) and of the matrix `C`. The value of m must be at least zero.

n Specifies the number of columns of the matrix op(`B`) and the number of columns of the matrix `C`. The value of n must be at least zero.

k Specifies the number of columns of the matrix op(`A`) and the number of rows of the matrix op(`B`). The value of k must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a The buffer holding the input matrix `A`.

	`A` not transposed	`A` transposed
Column major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*m`
Row major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*m`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*k`

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `m`.

b The buffer holding the input matrix `B`.

	`B` not transposed	`B` transposed
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`

See *Matrix Storage* for more details.

ldb The leading dimension of `B`. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

beta Scaling factor for matrix `C`.

c The buffer holding the input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

Output Parameters

c The buffer, which is overwritten by `alpha`*op(`A`)*op(`B`) + `beta`*`C`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm(sycl::queue &queue,
                     onemkl::transpose transa,
                     onemkl::transpose transb,
                     std::int64_t m,
                     std::int64_t n,
                     std::int64_t k,
                     Ts alpha,
                     const Ta *a,
                     std::int64_t lda,
```

(continues on next page)

(continued from previous page)

```
                      const Tb *b,
                      std::int64_t ldb,
                      Ts beta,
                      Tc *c,
                      std::int64_t ldc,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemm(sycl::queue &queue,
                      onemkl::transpose transa,
                      onemkl::transpose transb,
                      std::int64_t m,
                      std::int64_t n,
                      std::int64_t k,
                      Ts alpha,
                      const Ta *a,
                      std::int64_t lda,
                      const Tb *b,
                      std::int64_t ldb,
                      Ts beta,
                      Tc *c,
                      std::int64_t ldc,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

transa Specifies the form of op(A), the transposition operation applied to A.

transb Specifies the form of op(B), the transposition operation applied to B.

m Specifies the number of rows of the matrix op(A) and of the matrix C. The value of m must be at least zero.

n Specifies the number of columns of the matrix op(B) and the number of columns of the matrix C. The value of n must be at least zero.

k Specifies the number of columns of the matrix op(A) and the number of rows of the matrix op(B). The value of k must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Pointer to input matrix A.

	A not transposed	A transposed
Column major	A is an m-by-k matrix so the array a must have size at least lda*k.	A is an k-by-m matrix so the array a must have size at least lda*m
Row major	A is an m-by-k matrix so the array a must have size at least lda*m.	A is an k-by-m matrix so the array a must have size at least lda*k

See *Matrix Storage* for more details.

lda The leading dimension of A. It must be positive.

	A not transposed	A transposed
Column major	lda must be at least m.	lda must be at least k.
Row major	lda must be at least k.	lda must be at least m.

b Pointer to input matrix B.

	B not transposed	B transposed
Column major	B is an k-by-n matrix so the array b must have size at least ldb*n.	B is an n-by-k matrix so the array b must have size at least ldb*k
Row major	B is an k-by-n matrix so the array b must have size at least ldb*k.	B is an n-by-k matrix so the array b must have size at least ldb*n

See *Matrix Storage* for more details.

ldb The leading dimension of B. It must be positive.

	B not transposed	B transposed
Column major	ldb must be at least k.	ldb must be at least n.
Row major	ldb must be at least n.	ldb must be at least k.

beta Scaling factor for matrix C.

c The pointer to input/output matrix C. It must have a size of at least ldc*n if column major layout is used to store matrices or at least ldc*m if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc The leading dimension of C. It must be positive and at least m if column major layout is used to store matrices or at least n if column major layout is used to store matrices.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Pointer to the output matrix, overwritten by alpha*op(A)*op(B) + beta*C.

Notes

If beta = 0, matrix C does not need to be initialized before calling gemm.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

hemm

Computes a matrix-matrix product where one input matrix is Hermitian and one is general.

Description

The `hemm` routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, where one of the matrices in the multiplication is Hermitian. The argument `left_right` determines if the Hermitian matrix, `A`, is on the left of the multiplication (`left_right` = `side::left`) or on the right (`left_right` = `side::right`). Depending on `left_right`, the operation is defined as:

$$C \leftarrow alpha * A * B + beta * C$$

or

$$C \leftarrow alpha * B * A + beta * C$$

where:

`alpha` and `beta` are scalars,

`A` is a Hermitian matrix, either `m`-by-`m` or `n`-by-`n` matrices,

`B` and `C` are `m`-by-`n` matrices.

`hemm` supports the following precisions:

T
`std::complex<float>`
`std::complex<double>`

hemm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void hemm(sycl::queue &queue,
              onemkl::side left_right,
              onemkl::uplo upper_lower,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void hemm(sycl::queue &queue,
              onemkl::side left_right,
              onemkl::uplo upper_lower,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

uplo Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

m Specifies the number of rows of the matrix `B` and `C`.

The value of `m` must be at least zero.

n Specifies the number of columns of the matrix `B` and `C`.

The value of `n` must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Buffer holding input matrix `A`. Must have size at least `lda*m` if `A` is on the left of the multiplication, or `lda*n` if `A` is on the right. See *Matrix Storage* for more details.

lda Leading dimension of `A`. Must be at least `m` if `A` is on the left of the multiplication, or at least `n` if `A` is on the right. Must be positive.

b Buffer holding input matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

beta Scaling factor for matrix `C`.

c The buffer holding the input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

Output Parameters

c Output buffer, overwritten by `alpha*A*B + beta*C` (`left_right = side::left`) or `alpha*B*A + beta*C` (`left_right = side::right`).

Notes

If `beta = 0`, matrix `C` does not need to be initialized before calling `hemm`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

hemm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event hemm(sycl::queue &queue,
                     onemkl::side left_right,
                     onemkl::uplo upper_lower,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T* a,
                     std::int64_t lda,
                     const T* b,
                     std::int64_t ldb,
                     T beta,
```

(continues on next page)

(continued from previous page)

```
                    T* c,
                    std::int64_t ldc,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event hemm(sycl::queue &queue,
                    onemkl::side left_right,
                    onemkl::uplo upper_lower,
                    std::int64_t m,
                    std::int64_t n,
                    T alpha,
                    const T* a,
                    std::int64_t lda,
                    const T* b,
                    std::int64_t ldb,
                    T beta,
                    T* c,
                    std::int64_t ldc,
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

uplo Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

m Specifies the number of rows of the matrix `B` and `C`.

The value of `m` must be at least zero.

n Specifies the number of columns of the matrix `B` and `C`.

The value of `n` must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Pointer to input matrix `A`. Must have size at least `lda*m` if `A` is on the left of the multiplication, or `lda*n` if `A` is on the right. See *Matrix Storage* for more details.

lda Leading dimension of `A`. Must be at least `m` if `A` is on the left of the multiplication, or at least `n` if `A` is on the right. Must be positive.

b Pointer to input matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

beta Scaling factor for matrix `C`.

c The pointer to input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Pointer to the output matrix, overwritten by `alpha*A*B + beta*C` (`left_right = side::left`) or `alpha*B*A + beta*C` (`left_right = side::right`).

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `hemm`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

herk

Performs a Hermitian rank-k update.

Description

The `herk` routines compute a rank-k update of a Hermitian matrix `C` by a general matrix `A`. The operation is defined as:

$$C \leftarrow alpha * op(A) * op(A)^H + beta * C$$

where:

op(`X`) is one of op(`X`) = `X` or op(`X`) = X^H,

`alpha` and `beta` are real scalars,

`C` is a Hermitian matrix and `A` is a general matrix.

Here op(`A`) is `n` x `k`, and `C` is `n` x `n`.

herk supports the following precisions:

T	T_real
std::complex<float>	float
std::complex<double>	double

herk (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void herk(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              std::int64_t n,
              std::int64_t k,
              T_real alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              T_real beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void herk(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              std::int64_t n,
              std::int64_t k,
              T_real alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              T_real beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether A's data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans Specifies op(A), the transposition operation applied to A. See *oneMKL defined datatypes* for more details. Supported operations are transpose::nontrans and transpose::conjtrans.

n The number of rows and columns in C.The value of n must be at least zero.

k Number of columns in op(A).

The value of k must be at least zero.

alpha Real scaling factor for the rank-k update.

a Buffer holding input matrix `A`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

beta Real scaling factor for matrix `C`.

c Buffer holding input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc Leading dimension of `C`. Must be positive and at least `n`.

Output Parameters

c The output buffer, overwritten by `alpha`*op(`A`)*op(`A`)^T + `beta`*`C`. The imaginary parts of the diagonal elements are set to zero.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

herk (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event herk(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
                        std::int64_t k,
                        T_real alpha,
                        const T* a,
                        std::int64_t lda,
                        T_real beta,
                        T* c,
                        std::int64_t ldc,
                        const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event herk(sycl::queue &queue,
                        onemkl::uplo upper_lower,
                        onemkl::transpose trans,
                        std::int64_t n,
                        std::int64_t k,
                        T_real alpha,
                        const T* a,
                        std::int64_t lda,
                        T_real beta,
                        T* c,
                        std::int64_t ldc,
                        const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details. Supported operations are `transpose::nontrans` and `transpose::conjtrans`.

n The number of rows and columns in `C`.The value of `n` must be at least zero.

k Number of columns in op(`A`).

The value of `k` must be at least zero.

alpha Real scaling factor for the rank-k update.

a Pointer to input matrix `A`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

beta Real scaling factor for matrix `C`.

c Pointer to input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc Leading dimension of `C`. Must be positive and at least `n`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Pointer to the output matrix, overwritten by `alpha`*op(`A`)*op(`A`)^T + `beta`*`C`. The imaginary parts of the diagonal elements are set to zero.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

her2k

Performs a Hermitian rank-2k update.

Description

The `her2k` routines perform a rank-2k update of an `n` x `n` Hermitian matrix `C` by general matrices `A` and `B`.

If `trans` = `transpose::nontrans`, the operation is defined as:

$$C \leftarrow alpha * A * B^H + conjg(alpha) * B * A^H + beta * C$$

where `A` is `n` x `k` and `B` is `k` x `n`.

If `trans` = `transpose::conjtrans`, the operation is defined as:

$$C \leftarrow alpha * B * A^H + conjg(alpha) * A * B^H + beta * C$$

where `A` is `k` x `n` and `B` is `n` x `k`.

In both cases:

`alpha` is a complex scalar and `beta` is a real scalar.

`C` is a Hermitian matrix and `A` , `B` are general matrices.

The inner dimension of both matrix multiplications is `k`.

`her2k` supports the following precisions:

T	T_real
`std::complex<float>`	`float`
`std::complex<double>`	`double`

her2k (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void her2k(sycl::queue &queue,
               onemkl::uplo upper_lower,
               onemkl::transpose trans,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               T_real beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void her2k(sycl::queue &queue,
               onemkl::uplo upper_lower,
               onemkl::transpose trans,
               std::int64_t n,
               std::int64_t k,
               T alpha,
```

(continues on next page)

(continued from previous page)

```
                sycl::buffer<T,1> &a,
                std::int64_t lda,
                sycl::buffer<T,1> &b,
                std::int64_t ldb,
                T_real beta,
                sycl::buffer<T,1> &c,
                std::int64_t ldc)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans Specifies the operation to apply, as described above. Supported operations are `transpose::nontrans` and `transpose::conjtrans`.

n The number of rows and columns in `C`. The value of `n` must be at least zero.

k The inner dimension of matrix multiplications. The value of `k` must be at least equal to zero.

alpha Complex scaling factor for the rank-2k update.

a Buffer holding input matrix `A`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b Buffer holding input matrix `B`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`.

See *Matrix Storage* for more details.

ldb The leading dimension of `B`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

beta Real scaling factor for matrix `C`.

c Buffer holding input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc Leading dimension of `C`. Must be positive and at least `n`.

Output Parameters

c Output buffer, overwritten by the updated `C` matrix.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

her2k (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event her2k(sycl::queue &queue,
                      onemkl::uplo upper_lower,
                      onemkl::transpose trans,
                      std::int64_t n,
                      std::int64_t k,
                      T alpha,
                      const T* a,
                      std::int64_t lda,
                      const T* b,
                      std::int64_t ldb,
                      T_real beta,
                      T* c,
                      std::int64_t ldc,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event her2k(sycl::queue &queue,
                      onemkl::uplo upper_lower,
                      onemkl::transpose trans,
                      std::int64_t n,
                      std::int64_t k,
                      T alpha,
                      const T* a,
                      std::int64_t lda,
                      const T* b,
                      std::int64_t ldb,
                      T_real beta,
                      T* c,
                      std::int64_t ldc,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans Specifies the operation to apply, as described above. Supported operations are `transpose::nontrans` and `transpose::conjtrans`.

n The number of rows and columns in `C`. The value of `n` must be at least zero.

k The inner dimension of matrix multiplications. The value of `k` must be at least equal to zero.

alpha Complex scaling factor for the rank-2k update.

a Pointer to input matrix `A`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b Pointer to input matrix `B`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`.

See *Matrix Storage* for more details.

ldb The leading dimension of `B`. It must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

beta Real scaling factor for matrix `C`.

c Pointer to input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc Leading dimension of `C`. Must be positive and at least `n`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Pointer to the output matrix, overwritten by the updated `C` matrix.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

symm

Computes a matrix-matrix product where one input matrix is symmetric and one matrix is general.

Description

The `symm` routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, where one of the matrices in the multiplication is symmetric. The argument `left_right` determines if the symmetric matrix, `A`, is on the left of the multiplication (`left_right` = `side::left`) or on the right (`left_right` = `side::right`). Depending on `left_right`, the operation is defined as:

$$C \leftarrow alpha * A * B + beta * C$$

or

$$C \leftarrow alpha * B * A + beta * C$$

where:

`alpha` and `beta` are scalars,

`A` is a symmetric matrix, either `m`-by-`m` or `n`-by-`n`,

`B` and `C` are `m`-by-`n` matrices.

`symm` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

symm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void symm(sycl::queue &queue,
              onemkl::side left_right,
              onemkl::uplo upper_lower,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void symm(sycl::queue &queue,
              onemkl::side left_right,
              onemkl::uplo upper_lower,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

m Number of rows of `B` and `C`. The value of `m` must be at least zero.

n Number of columns of `B` and `C`. The value of `n` must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Buffer holding input matrix `A`. Must have size at least `lda*m` if `A` is on the left of the multiplication, or `lda*n` if `A` is on the right. See *Matrix Storage* for more details.

lda Leading dimension of `A`. Must be at least `m` if `A` is on the left of the multiplication, or at least `n` if `A` is on the right. Must be positive.

b Buffer holding input matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

beta Scaling factor for matrix `C`.

c The buffer holding the input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldc The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

Output Parameters

c Output buffer, overwritten by `alpha*A*B + beta*C` (`left_right = side::left`) or `alpha*B*A + beta*C` (`left_right = side::right`).

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `symm`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

symm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event symm(sycl::queue &queue,
                     onemkl::side left_right,
                     onemkl::uplo upper_lower,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T* a,
                     std::int64_t lda,
                     const T* b,
                     std::int64_t ldb,
                     T beta,
                     T* c,
                     std::int64_t ldc,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event symm(sycl::queue &queue,
                     onemkl::side left_right,
                     onemkl::uplo upper_lower,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T* a,
                     std::int64_t lda,
                     const T* b,
```

(continues on next page)

(continued from previous page)

```
                        std::int64_t ldb,
                        T beta,
                        T* c,
                        std::int64_t ldc,
                        const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

m Number of rows of `B` and `C`. The value of `m` must be at least zero.

n Number of columns of `B` and `C`. The value of `n` must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Pointer to input matrix `A`. Must have size at least `lda*m` if `A` is on the left of the multiplication, or `lda*n` if `A` is on the right. See *Matrix Storage* for more details.

lda Leading dimension of `A`. Must be at least `m` if `A` is on the left of the multiplication, or at least `n` if `A` is on the right. Must be positive.

b Pointer to input matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

beta Scaling factor for matrix `C`.

c The pointer to input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Pointer to the output matrix, overwritten by `alpha*A*B + beta*C` (`left_right = side::left`) or `alpha*B*A + beta*C` (`left_right = side::right`).

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `symm`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

syrk

Performs a symmetric rank-k update.

Description

The `syrk` routines perform a rank-k update of a symmetric matrix `C` by a general matrix `A`. The operation is defined as:

$$C \leftarrow alpha * op(A) * op(A)^T + beta * C$$

where:

op(`X`) is one of op(`X`) = `X` or op(`X`) = X^T ,

`alpha` and `beta` are scalars,

`C` is a symmetric matrix and `A`is a general matrix.

Here op(`A`) is `n`-by-`k`, and `C` is `n`-by-`n`.

`syrk` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

syrk (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void syrk(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              std::int64_t n,
              std::int64_t k,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void syrk(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose trans,
              std::int64_t n,
              std::int64_t k,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              T beta,
              sycl::buffer<T,1> &c,
              std::int64_t ldc)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A` (See *oneMKL defined datatypes* for more details). Conjugation is never performed, even if `trans` = `transpose::conjtrans`.

n Number of rows and columns in `C`. The value of `n` must be at least zero.

k Number of columns in op(`A`).The value of `k` must be at least zero.

alpha Scaling factor for the rank-k update.

a Buffer holding input matrix `A`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

beta Scaling factor for matrix `C`.

c Buffer holding input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc Leading dimension of `C`. Must be positive and at least `n`.

Output Parameters

c Output buffer, overwritten by `alpha`*op(`A`)*op(`A`)^T + `beta`*`C`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

syrk (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syrk(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     std::int64_t n,
                     std::int64_t k,
                     T alpha,
                     const T* a,
                     std::int64_t lda,
                     T beta,
                     T* c,
                     std::int64_t ldc,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syrk(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose trans,
                     std::int64_t n,
                     std::int64_t k,
                     T alpha,
                     const T* a,
                     std::int64_t lda,
                     T beta,
                     T* c,
                     std::int64_t ldc,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A` (See *oneMKL defined datatypes* for more details). Conjugation is never performed, even if `trans` = `transpose::conjtrans`.

n Number of rows and columns in `C`. The value of `n` must be at least zero.

k Number of columns in op(`A`). The value of `k` must be at least zero.

alpha Scaling factor for the rank-k update.

a Pointer to input matrix `A`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

beta Scaling factor for matrix `C`.

c Pointer to input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details.

ldc Leading dimension of `C`. Must be positive and at least `n`.

Output Parameters

c Pointer to the output matrix, overwritten by `alpha`*op(`A`)*op(`A`)^T + `beta`*`C`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

syr2k

Performs a symmetric rank-2k update.

Description

The `syr2k` routines perform a rank-2k update of an `n` x `n` symmetric matrix `C` by general matrices `A` and `B`.

If `trans` = `transpose::nontrans`, the operation is defined as:

$$C \leftarrow alpha * (A * B^T + B * A^T) + beta * C$$

where `A` is `n` x `k` and `B` is `k` x `n`.

If `trans` = `transpose::trans`, the operation is defined as:

$$C \leftarrow alpha * (A^T * B + B^T * A) + beta * C$$

where `A` is `k` x `n` and `B` is `n` x `k`.

In both cases:

`alpha` and `beta` are scalars,

`C` is a symmetric matrix and `A`,`B` are general matrices,

The inner dimension of both matrix multiplications is `k`.

`syr2k` supports the following precisions:

T
float
double
std::complex<float>
std::complex<double>

syr2k (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void syr2k(sycl::queue &queue,
               onemkl::uplo upper_lower,
               onemkl::transpose trans,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               T beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void syr2k(sycl::queue &queue,
               onemkl::uplo upper_lower,
               onemkl::transpose trans,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               T beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans Specifies the operation to apply, as described above. Conjugation is never performed, even if `trans = transpose::conjtrans`.

n Number of rows and columns in `C`.The value of `n` must be at least zero.

k Inner dimension of matrix multiplications.The value of `k` must be at least zero.

alpha Scaling factor for the rank-2k update.

a Buffer holding input matrix `A`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b Buffer holding input matrix `B`.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`.

See *Matrix Storage* for more details.

ldb The leading dimension of `B`. It must be positive.

	`trans` = `transpose::nontrans`	`trans` = `transpose::trans` or `transpose::conjtrans`
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

beta Scaling factor for matrix `C`.

c Buffer holding input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details

ldc Leading dimension of `C`. Must be positive and at least `n`.

Output Parameters

c Output buffer, overwritten by the updated `C` matrix.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

syr2k (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event syr2k(sycl::queue &queue,
                      onemkl::uplo upper_lower,
                      onemkl::transpose trans,
                      std::int64_t n,
                      std::int64_t k,
                      T alpha,
                      const T* a,
                      std::int64_t lda,
                      const T* b,
                      std::int64_t ldb,
                      T beta,
                      T* c,
                      std::int64_t ldc,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event syr2k(sycl::queue &queue,
                      onemkl::uplo upper_lower,
                      onemkl::transpose trans,
                      std::int64_t n,
                      std::int64_t k,
                      T alpha,
                      const T* a,
                      std::int64_t lda,
                      const T* b,
                      std::int64_t ldb,
                      T beta,
                      T* c,
                      std::int64_t ldc,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `A`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

trans Specifies the operation to apply, as described above. Conjugation is never performed, even if `trans = transpose::conjtrans`.

n Number of rows and columns in `C`. The value of `n` must be at least zero.

k Inner dimension of matrix multiplications.The value of `k` must be at least zero.

alpha Scaling factor for the rank-2k update.

a Pointer to input matrix `A`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b Pointer to input matrix `B`.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`.

See *Matrix Storage* for more details.

ldb The leading dimension of `B`. It must be positive.

	`trans = transpose::nontrans`	`trans = transpose::trans` or `transpose::conjtrans`
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

beta Scaling factor for matrix `C`.

c Pointer to input/output matrix `C`. Must have size at least `ldc*n`. See *Matrix Storage* for more details

ldc Leading dimension of `C`. Must be positive and at least `n`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Pointer to the output matrix, overwritten by the updated `C` matrix.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

trmm

Computes a matrix-matrix product where one input matrix is triangular and one input matrix is general.

Description

The `trmm` routines compute a scalar-matrix-matrix product where one of the matrices in the multiplication is triangular. The argument `left_right` determines if the triangular matrix, `A`, is on the left of the multiplication (`left_right` = `side::left`) or on the right (`left_right` = `side::right`). Depending on `left_right`. The operation is defined as:

$$B \leftarrow alpha * op(A) * B$$

or

$$B \leftarrow alpha * B * op(A)$$

where:

op(`A`) is one of op(`A`) = A, or op(`A`) = A^T, or op(`A`) = A^H,

`alpha` is a scalar,

`A` is a triangular matrix, and `B` is a general matrix.

Here `B` is `m` x `n` and `A` is either `m` x `m` or `n` x `n`, depending on `left_right`.

`trmm` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

trmm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trmm(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose transa,
              onemkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trmm(sycl::queue &queue,
              onemkl::uplo upper_lower,
              onemkl::transpose transa,
              onemkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb)
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

uplo Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_diag Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m Specifies the number of rows of `B`. The value of `m` must be at least zero.

n Specifies the number of columns of `B`. The value of `n` must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Buffer holding input matrix `A`. Must have size at least `lda*m` if `left_right = side::left`, or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda Leading dimension of `A`. Must be at least `m` if `left_right = side::left`, and at least `n` if `left_right = side::right`. Must be positive.

b Buffer holding input/output matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

Output Parameters

b Output buffer, overwritten by `alpha*op(A)*B` or `alpha*B*op(A)`.

Notes

If `alpha` = 0, matrix `B` is set to zero, and `A` and `B` do not need to be initialized at entry.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

trmm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trmm(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose transa,
                     onemkl::diag unit_diag,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T* a,
                     std::int64_t lda,
                     T* b,
                     std::int64_t ldb,
```

(continues on next page)

(continued from previous page)

```
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trmm(sycl::queue &queue,
                     onemkl::uplo upper_lower,
                     onemkl::transpose transa,
                     onemkl::diag unit_diag,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T* a,
                     std::int64_t lda,
                     T* b,
                     std::int64_t ldb,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether `A` is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See *oneMKL defined datatypes* for more details.

uplo Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_diag Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m Specifies the number of rows of `B`. The value of `m` must be at least zero.

n Specifies the number of columns of `B`. The value of `n` must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Pointer to input matrix `A`. Must have size at least `lda*m` if `left_right = side::left`, or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda Leading dimension of `A`. Must be at least `m` if `left_right = side::left`, and at least `n` if `left_right = side::right`. Must be positive.

b Pointer to input/output matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b Pointer to the output matrix, overwritten by `alpha`*op(`A`)*`B` or `alpha`*`B`*op(`A`).

Notes

If `alpha` = 0, matrix `B` is set to zero, and `A` and `B` do not need to be initialized at entry.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

trsm

Solves a triangular matrix equation (forward or backward solve).

Description

The `trsm` routines solve one of the following matrix equations:

$$op(A) * X = alpha * B$$

or

$$X * op(A) = alpha * B$$

where:

op(`A`) is one of op(`A`) = `A`, or op(`A`) = A^T, or op(`A`) = A^H,

`alpha` is a scalar,

`A` is a triangular matrix, and

`B` and `X` are `m` x `n` general matrices.

`A` is either `m` x `m` or `n` x `n`, depending on whether it multiplies `X` on the left or right. On return, the matrix `B` is overwritten by the solution matrix `X`.

`trsm` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

trsm (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trsm(sycl::queue &queue,
              onemkl::side left_right,
              onemkl::uplo upper_lower,
              onemkl::transpose transa,
              onemkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trsm(sycl::queue &queue,
              onemkl::side left_right,
              onemkl::uplo upper_lower,
              onemkl::transpose transa,
              onemkl::diag unit_diag,
              std::int64_t m,
              std::int64_t n,
              T alpha,
              sycl::buffer<T,1> &a,
              std::int64_t lda,
              sycl::buffer<T,1> &b,
              std::int64_t ldb)
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether `A` multiplies `X` on the left (`side::left`) or on the right (`side::right`). See *oneMKL defined datatypes* for more details.

uplo Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_diag Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m Specifies the number of rows of `B`. The value of `m` must be at least zero.

n Specifies the number of columns of `B`. The value of `n` must be at least zero.

alpha Scaling factor for the solution.

a Buffer holding input matrix `A`. Must have size at least `lda*m` if `left_right = side::left`, or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda Leading dimension of `A`. Must be at least `m` if `left_right = side::left`, and at least `n` if `left_right = side::right`. Must be positive.

b Buffer holding input/output matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

Output Parameters

b Output buffer. Overwritten by the solution matrix `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero, and `A` and `B` do not need to be initialized at entry.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

trsm (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trsm(sycl::queue &queue,
                     onemkl::side left_right,
                     onemkl::uplo upper_lower,
                     onemkl::transpose transa,
                     onemkl::diag unit_diag,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T* a,
                     std::int64_t lda,
                     T* b,
                     std::int64_t ldb,
```

(continues on next page)

(continued from previous page)

```
                    const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trsm(sycl::queue &queue,
                     onemkl::side left_right,
                     onemkl::uplo upper_lower,
                     onemkl::transpose transa,
                     onemkl::diag unit_diag,
                     std::int64_t m,
                     std::int64_t n,
                     T alpha,
                     const T* a,
                     std::int64_t lda,
                     T* b,
                     std::int64_t ldb,
                     const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether `A` multiplies `X` on the left (`side::left`) or on the right (`side::right`). See *oneMKL defined datatypes* for more details.

uplo Specifies whether the matrix `A` is upper or lower triangular. See *oneMKL defined datatypes* for more details.

transa Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

unit_diag Specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m Specifies the number of rows of `B`. The value of `m` must be at least zero.

n Specifies the number of columns of `B`. The value of `n` must be at least zero.

alpha Scaling factor for the solution.

a Pointer to input matrix `A`. Must have size at least `lda*m` if `left_right = side::left`, or `lda*n` if `left_right = side::right`. See *Matrix Storage* for more details.

lda Leading dimension of `A`. Must be at least `m` if `left_right = side::left`, and at least `n` if `left_right = side::right`. Must be positive.

b Pointer to input/output matrix `B`. Must have size at least `ldb*n` if column major layout is used to store matrices or at least `ldb*m` if row major layout is used to store matrices. See *Matrix Storage* for more details.

ldb Leading dimension of `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b Pointer to the output matrix. Overwritten by the solution matrix `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero, and `A` and `B` do not need to be initialized at entry.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS Level 3 Routines*

Parent topic: *BLAS Routines*

BLAS-like Extensions

oneAPI Math Kernel Library DPC++ provides additional routines to extend the functionality of the BLAS routines. These include routines to compute many independent vector-vector and matrix-matrix operations.

The following table lists the BLAS-like extensions with their descriptions.

Routines	Description
axpy_batch	Computes groups of vector-scalar products added to a vector.
gemm_batch	Computes groups of matrix-matrix products with general matrices.
trsm_batch	Solves a triangular matrix equation for a group of matrices.
gemmt	Computes a matrix-matrix product with general matrices, but updates only the upper or lower triangular part of the result matrix.
gemm_bias	Computes a matrix-matrix product using general integer matrices with bias

axpy_batch

Computes a group of `axpy` operations.

Description

The `axpy_batch` routines are batched versions of *axpy*, performing multiple `axpy` operations in a single call. Each `axpy` operation adds a scalar-vector product to a vector.

`axpy_batch` supports the following precisions for data.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

axpy_batch (Buffer Version)

Description

The buffer version of `axpy_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
   X and Y are vectors at offset i * stridex, i * stridey in x and y
   Y := alpha * X + Y
end for
```

where:

`alpha` is scalar,

`X` and `Y` are vectors.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void axpy_batch(sycl::queue &queue,
                    std::int64_t n,
                    T alpha,
                    sycl::buffer<T,
                    1> &x,
                    std::int64_t incx,
                    std::int64_t stridex,
                    sycl::buffer<T,
                    1> &y,
                    std::int64_t incy,
                    std::int64_t stridey,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void axpy_batch(sycl::queue &queue,
                    std::int64_t n,
                    T alpha,
                    sycl::buffer<T,
                    1> &x,
                    std::int64_t incx,
                    std::int64_t stridex,
                    sycl::buffer<T,
                    1> &y,
                    std::int64_t incy,
                    std::int64_t stridey,
                    std::int64_t batch_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in `X` and `Y`.

alpha Specifies the scalar `alpha`.

x Buffer holding input vectors `X` with size `stridex` * `batch_size`.

incx Stride of vector `X`.

stridex Stride between different `X` vectors.

y Buffer holding input/output vectors `Y` with size `stridey` * `batch_size`.

incy Stride of vector `Y`.

stridey Stride between different `Y` vectors.

batch_size Specifies the number of `axpy` operations to perform.

Output Parameters

y Output buffer, overwritten by `batch_size` `axpy` operations of the form `alpha` * `X` + `Y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

axpy_batch (USM Version)

Description

The USM version of `axpy_batch` supports the group API and strided API.

The group API operation is defined as

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        X and Y are vectors in x[idx] and y[idx]
        Y := alpha[i] * X + Y
        idx := idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
   X and Y are vectors at offset i * stridex, i * stridey in x and y
   Y := alpha * X + Y
end for
```

where:

`alpha` is scalar,

`X` and `Y` are vectors.

For group API, `x` and `y` arrays contain the pointers for all the input vectors. The total number of vectors in `x` and `y` are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, `x` and `y` arrays contain all the input vectors. The total number of vectors in `x` and `y` are given by the `batch_size` parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event axpy_batch(sycl::queue &queue,
                           std::int64_t *n,
                           T *alpha,
                           const T **x,
                           std::int64_t *incx,
                           T **y,
                           std::int64_t *incy,
                           std::int64_t group_count,
                           std::int64_t *group_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event axpy_batch(sycl::queue &queue,
                           std::int64_t *n,
                           T *alpha,
                           const T **x,
                           std::int64_t *incx,
                           T **y,
                           std::int64_t *incy,
                           std::int64_t group_count,
                           std::int64_t *group_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Array of `group_count` integers. `n[i]` specifies the number of elements in vectors `X` and `Y` for every vector in group `i`.

alpha Array of `group_count` scalar elements. `alpha[i]` specifies the scaling factor for vector `X` in group `i`.

x Array of pointers to input vectors `X` with size `total_batch_count`. The size of array allocated for the `X` vector of the group `i` must be at least (1 + (`n[i]` – 1)*abs(`incx[i]`))``. See *Matrix Storage* for more details.

incx Array of `group_count` integers. `incx[i]` specifies the stride of vector `X` in group `i`.

y Array of pointers to input/output vectors `Y` with size `total_batch_count`. The size of array allocated for the `Y` vector of the group `i` must be at least (1 + (`n[i]` – 1)*abs(`incy[i]`))``. See *Matrix Storage* for more details.

incy Array of `group_count` integers. `incy[i]` specifies the stride of vector `Y` in group `i`.

group_count Number of groups. Must be at least 0.

group_size Array of `group_count` integers. `group_size[i]` specifies the number of `axpy` operations in group `i`. Each element in `group_size` must be at least 0.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Array of pointers holding the `Y` vectors, overwritten by `total_batch_count axpy` operations of the form `alpha` * `X` + `Y`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event axpy_batch(sycl::queue &queue,
                           std::int64_t n,
                           T alpha,
                           const T *x,
                           std::int64_t incx,
                           std::int64_t stridex,
                           T *y,
                           std::int64_t incy,
                           std::int64_t stridey,
                           std::int64_t batch_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event axpy_batch(sycl::queue &queue,
                           std::int64_t n,
                           T alpha,
                           const T *x,
                           std::int64_t incx,
                           std::int64_t stridex,
                           T *y,
                           std::int64_t incy,
                           std::int64_t stridey,
                           std::int64_t batch_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n Number of elements in `X` and `Y`.

alpha Specifies the scalar `alpha`.

x Pointer to input vectors `X` with size `stridex` * `batch_size`.

incx Stride of vector `X`.

stridex Stride between different `X` vectors.

y Pointer to input/output vectors `Y` with size `stridey` * `batch_size`.

incy Stride of vector `Y`.

stridey Stride between different `Y` vectors.

batch_size Specifies the number of `axpy` operations to perform.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

y Output vectors, overwritten by `batch_size` `axpy` operations of the form `alpha` * `X` + `Y`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic:*BLAS-like Extensions*

gemm_batch

Computes a group of `gemm` operations.

Description

The `gemm_batch` routines are batched versions of *gemm*, performing multiple `gemm` operations in a single call. Each `gemm` operation perform a matrix-matrix product with general matrices.

`gemm_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

gemm_batch (Buffer Version)

Description

The buffer version of `gemm_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
    A, B and C are matrices at offset i * stridea, i * strideb, i * stridec in a, b␣
↪and c.
    C := alpha * op(A) * op(B) + beta * C
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` and `beta` are scalars,

`A`, `B`, and `C` are matrices,

op(`A`) is `m` x `k`, op(`B`) is `k` x `n`, and `C` is `m` x `n`.

The `a`, `b` and `c` buffers contain all the input matrices. The stride between matrices is given by the stride parameter. The total number of matrices in `a`, `b` and `c` buffers is given by the `batch_size` parameter.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemm_batch(sycl::queue &queue,
                    onemkl::transpose transa,
                    onemkl::transpose transb,
                    std::int64_t m,
                    std::int64_t n,
                    std::int64_t k,
                    T alpha,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<T,1> &b,
                    std::int64_t ldb,
                    std::int64_t strideb,
                    T beta,
                    sycl::buffer<T,1> &c,
                    std::int64_t ldc,
                    std::int64_t stridec,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemm_batch(sycl::queue &queue,
                    onemkl::transpose transa,
                    onemkl::transpose transb,
                    std::int64_t m,
                    std::int64_t n,
                    std::int64_t k,
                    T alpha,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<T,1> &b,
                    std::int64_t ldb,
```

(continues on next page)

(continued from previous page)

```
                        std::int64_t strideb,
                        T beta,
                        sycl::buffer<T,1> &c,
                        std::int64_t ldc,
                        std::int64_t stridec,
                        std::int64_t batch_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

transa Specifies op(`A`) the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

transb Specifies op(`B`) the transposition operation applied to the matrices `B`. See *oneMKL defined datatypes* for more details.

m Number of rows of op(`A`) and `C`. Must be at least zero.

n Number of columns of op(`B`) and `C`. Must be at least zero.

k Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha Scaling factor for the matrix-matrix products.

a Buffer holding the input matrices `A` with size `stridea` * `batch_size`.

lda The leading dimension of the matrices `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `m`.

stridea Stride between different `A` matrices.

b Buffer holding the input matrices `B` with size `strideb` * `batch_size`.

ldb The leading dimension of the matrices``B``. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

strideb Stride between different `B` matrices.

beta Scaling factor for the matrices `C`.

c Buffer holding input/output matrices `C` with size `stridec` * `batch_size`.

ldc The leading dimension of the matrices `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

stridec Stride between different `C` matrices. Must be at least `ldc` * `n`.

batch_size Specifies the number of matrix multiply operations to perform.

Output Parameters

c Output buffer, overwritten by `batch_size` matrix multiply operations of the form `alpha` * op(`A`)*op(`B`) + `beta` * `C`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_batch`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemm_batch (USM Version)

Description

The USM version of `gemm_batch` supports the group API and strided API.

The group API operation is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        A, B, and C are matrices in a[idx], b[idx] and c[idx]
        C := alpha[i] * op(A) * op(B) + beta[i] * C
        idx = idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    A, B and C are matrices at offset i * stridea, i * strideb, i * stridec in a, b␣
↪and c.
    C := alpha * op(A) * op(B) + beta * C
end for
```

where:

op(X) is one of op(X) = X, or op(X) = X^T, or op(X) = X^H,

`alpha` and `beta` are scalars,

`A`, `B`, and `C` are matrices,

op(`A`) is `m` x `k`, op(`B`) is `k` x `n`, and `C` is `m` x `n`.

For group API, `a`, `b` and `c` arrays contain the pointers for all the input matrices. The total number of matrices in `a`, `b` and `c` are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, `a`, `b`, `c` arrays contain all the input matrices. The total number of matrices in `a`, `b` and `c` are given by the `batch_size` parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm_batch(sycl::queue &queue,
                           onemkl::transpose *transa,
                           onemkl::transpose *transb,
                           std::int64_t *m,
                           std::int64_t *n,
                           std::int64_t *k,
                           T *alpha,
                           const T **a,
                           std::int64_t *lda,
                           const T **b,
                           std::int64_t *ldb,
                           T *beta,
                           T **c,
                           std::int64_t *ldc,
                           std::int64_t group_count,
                           std::int64_t *group_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemm_batch(sycl::queue &queue,
                           onemkl::transpose *transa,
                           onemkl::transpose *transb,
                           std::int64_t *m,
                           std::int64_t *n,
                           std::int64_t *k,
                           T *alpha,
                           const T **a,
                           std::int64_t *lda,
                           const T **b,
                           std::int64_t *ldb,
                           T *beta,
                           T **c,
                           std::int64_t *ldc,
                           std::int64_t group_count,
                           std::int64_t *group_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

transa Array of `group_count` `onemkl::transpose` values. `transa[i]` specifies the form of op(`A`) used in the matrix multiplication in group `i`. See *oneMKL defined datatypes* for more details.

transb Array of `group_count` `onemkl::transpose` values. `transb[i]` specifies the form of op(`B`) used in the matrix multiplication in group `i`. See *oneMKL defined datatypes* for more details.

m Array of `group_count` integers. `m[i]` specifies the number of rows of op(`A`) and `C` for every matrix in group `i`. All entries must be at least zero.

n Array of `group_count` integers. `n[i]` specifies the number of columns of op(`B`) and `C` for every matrix in group `i`. All entries must be at least zero.

k Array of `group_count` integers. `k[i]` specifies the number of columns of op(`A`) and rows of op(`B`) for every matrix in group `i`. All entries must be at least zero.

alpha Array of `group_count` scalar elements. `alpha[i]` specifies the scaling factor for every matrix-matrix product in group `i`.

a Array of pointers to input matrices `A` with size `total_batch_count`.

See *Matrix Storage* for more details.

lda Array of `group_count` integers. `lda[i]` specifies the leading dimension of `A` for every matrix in group `i`. All entries must be positive.

	`A` not transposed	`A` transposed
Column major	`lda[i]` must be at least `m[i]`.	`lda[i]` must be at least `k[i]`.
Row major	`lda[i]` must be at least `k[i]`.	`lda[i]` must be at least `m[i]`.

b Array of pointers to input matrices `B` with size `total_batch_count`.

See *Matrix Storage* for more details.

ldb

Array of `group_count` integers. `ldb[i]` specifies the leading dimension of `B` for every matrix in group `i`. All entries must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb[i]` must be at least `k[i]`.	`ldb[i]` must be at least `n[i]`.
Row major	`ldb[i]` must be at least `n[i]`.	`ldb[i]` must be at least `k[i]`.

beta Array of `group_count` scalar elements. `beta[i]` specifies the scaling factor for matrix `C` for every matrix in group `i`.

c Array of pointers to input/output matrices `C` with size `total_batch_count`.

See *Matrix Storage* for more details.

ldc Array of `group_count` integers. `ldc[i]` specifies the leading dimension of `C` for every matrix in group `i`. All entries must be positive and `ldc[i]` must be at least `m[i]` if column major layout is used to store matrices or at least `n[i]` if row major layout is used to store matrices.

group_count Specifies the number of groups. Must be at least 0.

group_size Array of `group_count` integers. `group_size[i]` specifies the number of matrix multiply products in group `i`. All entries must be at least 0.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Overwritten by the `m[i]`-by-`n[i]` matrix calculated by (`alpha[i]` * op(`A`)*op(`B`) + `beta[i]` * `C`) for group `i`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_batch`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm_batch(sycl::queue &queue,
                           onemkl::transpose transa,
                           onemkl::transpose transb,
                           std::int64_t m,
                           std::int64_t n,
                           std::int64_t k,
                           T alpha,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stridea,
                           const T *b,
                           std::int64_t ldb,
                           std::int64_t strideb,
                           T beta,
                           T *c,
                           std::int64_t ldc,
                           std::int64_t stridec,
                           std::int64_t batch_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemm_batch(sycl::queue &queue,
                           onemkl::transpose transa,
                           onemkl::transpose transb,
                           std::int64_t m,
                           std::int64_t n,
                           std::int64_t k,
                           T alpha,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stridea,
                           const T *b,
                           std::int64_t ldb,
```

(continues on next page)

(continued from previous page)

```
                            std::int64_t strideb,
                            T beta,
                            T *c,
                            std::int64_t ldc,
                            std::int64_t stridec,
                            std::int64_t batch_size,
                            const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

transa Specifies op(`A`) the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

transb Specifies op(`B`) the transposition operation applied to the matrices `B`. See *oneMKL defined datatypes* for more details.

m Number of rows of op(`A`) and `C`. Must be at least zero.

n Number of columns of op(`B`) and `C`. Must be at least zero.

k Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha Scaling factor for the matrix-matrix products.

a Pointer to input matrices `A` with size `stridea` * `batch_size`.

lda The leading dimension of the matrices `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `m`.

stridea Stride between different `A` matrices.

b Pointer to input matrices `B` with size `strideb` * `batch_size`.

ldb The leading dimension of the matrices``B``. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

strideb Stride between different `B` matrices.

beta Scaling factor for the matrices `C`.

c Pointer to input/output matrices `C` with size `stridec` * `batch_size`.

ldc The leading dimension of the matrices `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

stridec Stride between different `C` matrices.

batch_size Specifies the number of matrix multiply operations to perform.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Output matrices, overwritten by `batch_size` matrix multiply operations of the form `alpha` * op(`A`)*op(`B`) + `beta` * `C`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_batch`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

trsm_batch

Computes a group of `trsm` operations.

Description

The `trsm_batch` routines are batched versions of *trsm*, performing multiple `trsm` operations in a single call. Each `trsm` solves an equation of the form op(A) * X = alpha * B or X * op(A) = alpha * B.

`trsm_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

trsm_batch (Buffer Version)

Description

The buffer version of `trsm_batch` supports only the strided API.

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
    A and B are matrices at offset i * stridea and i * strideb in a and b.
    if (left_right == onemkl::side::left) then
        compute X such that op(A) * X = alpha * B
    else
        compute X such that X * op(A) = alpha * B
    end if
    B := X
end for
```

where:

op(`A`) is one of op(`A`) = `A`, or op(A) = A^T, or op(`A`) = A^H,

`alpha` is a scalar,

`A` is a triangular matrix,

`B` and `X` are `m` x `n` general matrices,

`A` is either `m` x `m` or `n` x `n`,depending on whether it multiplies `X` on the left or right. On return, the matrix `B` is overwritten by the solution matrix `X`.

The `a` and `b` buffers contain all the input matrices. The stride between matrices is given by the stride parameter. The total number of matrices in `a` and `b` buffers are given by the `batch_size` parameter.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void trsm_batch(sycl::queue &queue,
                    onemkl::side left_right,
                    onemkl::uplo upper_lower,
                    onemkl::transpose trans,
                    onemkl::diag unit_diag,
                    std::int64_t m,
                    std::int64_t n,
                    T alpha,
                    sycl::buffer<T,1> &a,
                    std::int64_t lda,
                    std::int64_t stridea,
                    sycl::buffer<T,1> &b,
                    std::int64_t ldb,
                    std::int64_t strideb,
                    std::int64_t batch_size)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void trsm_batch(sycl::queue &queue,
```

(continues on next page)

(continued from previous page)

```
                        onemkl::side left_right,
                        onemkl::uplo upper_lower,
                        onemkl::transpose trans,
                        onemkl::diag unit_diag,
                        std::int64_t m,
                        std::int64_t n,
                        T alpha,
                        sycl::buffer<T,1> &a,
                        std::int64_t lda,
                        std::int64_t stridea,
                        sycl::buffer<T,1> &b,
                        std::int64_t ldb,
                        std::int64_t strideb,
                        std::int64_t batch_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether the matrices `A` multiply `X` on the left (`side::left`) or on the right (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower Specifies whether the matrices `A` are upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

unit_diag Specifies whether the matrices `A` are assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m Number of rows of the `B` matrices. Must be at least zero.

n Number of columns of the `B` matrices. Must be at least zero.

alpha Scaling factor for the solutions.

a Buffer holding the input matrices `A` with size `stridea` * `batch_size`.

lda Leading dimension of the matrices `A`. Must be at least `m` if `left_right` = `side::left`, and at least `n` if `left_right` = `side::right`. Must be positive.

stridea Stride between different `A` matrices.

b Buffer holding the input matrices `B` with size `strideb` * `batch_size`.

ldb Leading dimension of the matrices `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

strideb Stride between different `B` matrices.

batch_size Specifies the number of triangular linear systems to solve.

Output Parameters

b Output buffer, overwritten by `batch_size` solution matrices `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero and the matrices `A` and `B` do not need to be initialized before calling `trsm_batch`.

trsm_batch (USM Version)

Description

The USM version of `trsm_batch` supports the group API and strided API.

The group API operation is defined as:

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        A and B are matrices in a[idx] and b[idx]
        if (left_right == onemkl::side::left) then
            compute X such that op(A) * X = alpha[i] * B
        else
            compute X such that X * op(A) = alpha[i] * B
        end if
        B := X
        idx = idx + 1
    end for
end for
```

The strided API operation is defined as:

```
for i = 0 ... batch_size - 1
    A and B are matrices at offset i * stridea and i * strideb in a and b.
    if (left_right == onemkl::side::left) then
        compute X such that op(A) * X = alpha * B
    else
        compute X such that X * op(A) = alpha * B
    end if
    B := X
end for

where:
```

op(`A`) is one of op(`A`) = `A`, or op(A) = A^T, or op(`A`) = A^H,

`alpha` is a scalar,

`A` is a triangular matrix,

`B` and `X` are `m` x `n` general matrices,

`A` is either `m` x `m` or `n` x `n`,depending on whether it multiplies `X` on the left or right. On return, the matrix `B` is overwritten by the solution matrix `X`.

For group API, `a` and `b` arrays contain the pointers for all the input matrices. The total number of matrices in `a` and `b` are given by:

$$total\_batch\_count = \sum_{i=0}^{group\_count-1} group\_size[i]$$

For strided API, `a` and `b` arrays contain all the input matrices. The total number of matrices in `a` and `b` are given by the `batch_size` parameter.

Group API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trsm_batch(sycl::queue &queue,
                           onemkl::side *left_right,
                           onemkl::uplo *upper_lower,
                           onemkl::transpose *trans,
                           onemkl::diag *unit_diag,
                           std::int64_t *m,
                           std::int64_t *n,
                           T *alpha,
                           const T **a,
                           std::int64_t *lda,
                           T **b,
                           std::int64_t *ldb,
                           std::int64_t group_count,
                           std::int64_t *group_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trsm_batch(sycl::queue &queue,
                           onemkl::side *left_right,
                           onemkl::uplo *upper_lower,
                           onemkl::transpose *trans,
                           onemkl::diag *unit_diag,
                           std::int64_t *m,
                           std::int64_t *n,
                           T *alpha,
                           const T **a,
                           std::int64_t *lda,
                           T **b,
                           std::int64_t *ldb,
                           std::int64_t group_count,
                           std::int64_t *group_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Array of `group_count` `onemkl::side` values. `left_right[i]` specifies whether `A` multiplies `X` on the left (`side::left`) or on the right (`side::right`) for every `trsm` operation in group `i`. See *oneMKL defined datatypes* for more details.

upper_lower Array of `group_count` `onemkl::uplo` values. `upper_lower[i]` specifies whether `A` is upper or lower triangular for every matrix in group `i`. See *oneMKL defined datatypes* for more details.

trans Array of `group_count` `onemkl::transpose` values. `trans[i]` specifies the form of op(`A`) used for every `trsm` operation in group `i`. See *oneMKL defined datatypes* for more details.

unit_diag Array of `group_count` `onemkl::diag` values. `unit_diag[i]` specifies whether `A` is assumed to be unit triangular (all diagonal elements are 1) for every matrix in group `i`. See *oneMKL defined datatypes* for more details.

m Array of `group_count` integers. `m[i]` specifies the number of rows of `B` for every matrix in group `i`. All entries must be at least zero.

n Array of `group_count` integers. `n[i]` specifies the number of columns of `B` for every matrix in group `i`. All entries must be at least zero.

alpha Array of `group_count` scalar elements. `alpha[i]` specifies the scaling factor in group `i`.

a Array of pointers to input matrices `A` with size `total_batch_count`. See *Matrix Storage* for more details.

lda Array of `group_count` integers. `lda[i]` specifies the leading dimension of `A` for every matrix in group `i`. All entries must be at least `m` if `left_right` is `side::left`, and at least `n` if `left_right` is `side::right`. All entries must be positive.

b Array of pointers to input matrices `B` with size `total_batch_count`. See *Matrix Storage* for more details.

ldb Array of `group_count` integers. `ldb[i]` specifies the leading dimension of `B` for every matrix in group `i`. All entries must be positive and at least `m` and positive if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

group_count Specifies the number of groups. Must be at least 0.

group_size Array of `group_count` integers. `group_size[i]` specifies the number of `trsm` operations in group `i`. All entries must be at least 0.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

b Output buffer, overwritten by the `total_batch_count` solution matrices `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero and the matrices `A` and `B` do not need to be initialized before calling `trsm_batch`.

Return Values

Output event to wait on to ensure computation is complete.

Strided API

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event trsm_batch(sycl::queue &queue,
                           onemkl::side left_right,
                           onemkl::uplo upper_lower,
                           onemkl::transpose trans,
                           onemkl::diag unit_diag,
                           std::int64_t m,
                           std::int64_t n,
                           T alpha,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stridea,
                           T *b,
                           std::int64_t ldb,
                           std::int64_t strideb,
                           std::int64_t batch_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event trsm_batch(sycl::queue &queue,
                           onemkl::side left_right,
                           onemkl::uplo upper_lower,
                           onemkl::transpose trans,
                           onemkl::diag unit_diag,
                           std::int64_t m,
                           std::int64_t n,
                           T alpha,
                           const T *a,
                           std::int64_t lda,
                           std::int64_t stridea,
                           T *b,
                           std::int64_t ldb,
                           std::int64_t strideb,
                           std::int64_t batch_size,
                           const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right Specifies whether the matrices `A` multiply `X` on the left (`side::left`) or on the right (`side::right`). See *oneMKL defined datatypes* for more details.

upper_lower Specifies whether the matrices `A` are upper or lower triangular. See *oneMKL defined datatypes* for more details.

trans Specifies op(`A`), the transposition operation applied to the matrices `A`. See *oneMKL defined datatypes* for more details.

unit_diag Specifies whether the matrices `A` are assumed to be unit triangular (all diagonal elements are 1). See *oneMKL defined datatypes* for more details.

m Number of rows of the `B` matrices. Must be at least zero.

n Number of columns of the `B` matrices. Must be at least zero.

alpha Scaling factor for the solutions.

a Pointer to input matrices `A` with size `stridea` * `batch_size`.

lda Leading dimension of the matrices `A`. Must be at least `m` if `left_right` = `side::left`, and at least `n` if `left_right` = `side::right`. Must be positive.

stridea Stride between different `A` matrices.

b Pointer to input matrices `B` with size `strideb` * `batch_size`.

ldb Leading dimension of the matrices `B`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if row major layout is used to store matrices.

strideb Stride between different `B` matrices.

batch_size Specifies the number of triangular linear systems to solve.

Output Parameters

b Output matrices, overwritten by `batch_size` solution matrices `X`.

Notes

If `alpha` = 0, matrix `B` is set to zero and the matrices `A` and `B` do not need to be initialized before calling `trsm_batch`.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *BLAS-like Extensions*

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

gemmt

Computes a matrix-matrix product with general matrices, but updates only the upper or lower triangular part of the result matrix.

Description

The gemmt routines compute a scalar-matrix-matrix product and add the result to the upper or lower part of a scalar-matrix product, with general matrices. The operation is defined as:

$$C \leftarrow alpha * op(A) * op(B) + beta * C$$

where:

op(`X`) is one of op(`X`) = `X`, or op(`X`) = X^T, or op(`X`) = X^H,

`alpha` and `beta` are scalars

`A`, `B`, and `C` are matrices

op(`A`) is `n` x `k`, op(`B`) is `k` x `n`, and `C` is `n` x `n`.

`gemmt` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

gemmt (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemmt(sycl::queue &queue,
               onemkl::uplo upper_lower,
               onemkl::transpose transa,
               onemkl::transpose transb,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               T beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemmt(sycl::queue &queue,
               onemkl::uplo upper_lower,
               onemkl::transpose transa,
               onemkl::transpose transb,
               std::int64_t n,
               std::int64_t k,
               T alpha,
               sycl::buffer<T,1> &a,
               std::int64_t lda,
               sycl::buffer<T,1> &b,
               std::int64_t ldb,
               T beta,
               sycl::buffer<T,1> &c,
               std::int64_t ldc)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `C`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

transa Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

transb Specifies op(`B`), the transposition operation applied to `B`. See *oneMKL defined datatypes* for more details.

n Number of rows of op(`A`), columns of op(`B`), and columns and rows of`C`. Must be at least zero.

k Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Buffer holding the input matrix `A`.

	`A` not transposed	`A` transposed
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`.

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b Buffer holding the input matrix `B`.

	`B` not transposed	`B` transposed
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`.

See *Matrix Storage* for more details.

ldb The leading dimension of `B`. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

beta Scaling factor for matrix `C`.

c Buffer holding the input/output matrix `C`. Must have size at least `ldc * n`. See *Matrix Storage* for more details.

ldc Leading dimension of `C`. Must be positive and at least `m`.

Output Parameters

c Output buffer, overwritten by the upper or lower triangular part of `alpha` * op(`A`)*op(`B`) + `beta` * `C`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling gemmt.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemmt (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemmt(sycl::queue &queue,
                      onemkl::uplo upper_lower,
                      onemkl::transpose transa,
                      onemkl::transpose transb,
                      std::int64_t n,
                      std::int64_t k,
                      T alpha,
                      const T* a,
                      std::int64_t lda,
                      const T* b,
                      std::int64_t ldb,
                      T beta,
                      T* c,
                      std::int64_t ldc,
                      const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemmt(sycl::queue &queue,
                      onemkl::uplo upper_lower,
                      onemkl::transpose transa,
                      onemkl::transpose transb,
                      std::int64_t n,
                      std::int64_t k,
                      T alpha,
                      const T* a,
                      std::int64_t lda,
```

(continues on next page)

(continued from previous page)

```
                        const T* b,
                        std::int64_t ldb,
                        T beta,
                        T* c,
                        std::int64_t ldc,
                        const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Specifies whether `C`'s data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

transa Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

transb Specifies op(`B`), the transposition operation applied to `B`. See *oneMKL defined datatypes* for more details.

n Number of columns of op(`A`), columns of op(`B`), and columns of`C`. Must be at least zero.

k Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Pointer to input matrix `A`.

	`A` not transposed	`A` transposed
Column major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*n`
Row major	`A` is an `n`-by-`k` matrix so the array `a` must have size at least `lda*n`.	`A` is an `k`-by-`n` matrix so the array `a` must have size at least `lda*k`

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `n`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `n`.

b Pointer to input matrix `B`.

	`B` not transposed	`B` transposed
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`

See *Matrix Storage* for more details.

ldb The leading dimension of `B`. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

beta Scaling factor for matrix `C`.

c Pointer to input/output matrix `C`. Must have size at least `ldc` * `n`. See *Matrix Storage* for more details.

ldc Leading dimension of `C`. Must be positive and at least `m`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Pointer to the output matrix, overwritten by the upper or lower triangular part of `alpha` * op(`A`)*op(`B`) + `beta` * `C`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling gemmt.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

gemm_bias

Computes a matrix-matrix product using general integer matrices with bias.

Description

The gemm_bias routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, using general integer matrices with biases/offsets. The operation is defined as:

$$C \leftarrow alpha*(op(A)-A\_offset)*(op(B)-B\_offset)+beta*C+C\_offset$$

where:

op(`X`) is one of op(`X`) = `X`, or op(`X`) = X^T, or op(`X`) = X^H,

`alpha` and `beta` are scalars,

`A_offset` is an `m`-by-`k` matrix with every element equal to the value ao,

`B_offset` is a `k`-by-`n` matrix with every element equal to the value bo,

`C_offset` is an `m`-by-`n` matrix defined by the co buffer as described below,

`A`, `B`, and `C` are matrices,

op(`A`) is `m` x `k`, op(`B`) is `k` x `n`, and `C` is `m` x `n`.

`gemm_bias` supports the following precisions.

Ts	Ta	Tb	Tc
`float`	`std::uint8_t`	`std::uint8_t`	`std::int32_t`
`float`	`std::int8_t`	`std::uint8_t`	`std::int32_t`
`float`	`std::uint8_t`	`std::int8_t`	`std::int32_t`
`float`	`std::int8_t`	`std::int8_t`	`std::int32_t`

gemm_bias (Buffer Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    void gemm_bias(sycl::queue &queue,
                   onemkl::transpose transa,
                   onemkl::transpose transb,
                   onemkl::offset offset_type,
                   std::int64_t m,
                   std::int64_t n,
                   std::int64_t k,
                   Ts alpha,
                   sycl::buffer<Ta,1> &a,
                   std::int64_t lda,
                   Ta ao,
                   sycl::buffer<Tb,1> &b,
                   std::int64_t ldb,
                   Tb bo,
                   Ts beta,
                   sycl::buffer<Tc,1> &c,
                   std::int64_t ldc,
                   sycl::buffer<Tc,1> &co)
}
```

```
namespace oneapi::mkl::blas::row_major {
    void gemm_bias(sycl::queue &queue,
                   onemkl::transpose transa,
                   onemkl::transpose transb,
                   onemkl::offset offset_type,
                   std::int64_t m,
                   std::int64_t n,
                   std::int64_t k,
                   Ts alpha,
                   sycl::buffer<Ta,1> &a,
                   std::int64_t lda,
                   Ta ao,
                   sycl::buffer<Tb,1> &b,
                   std::int64_t ldb,
                   Tb bo,
                   Ts beta,
                   sycl::buffer<Tc,1> &c,
                   std::int64_t ldc,
                   sycl::buffer<Tc,1> &co)
}
```

Input Parameters

queue The queue where the routine should be executed.

transa Specifies op(`A`), the transposition operation applied to `A`. See *oneMKL defined datatypes* for more details.

transb Specifies op(`B`), the transposition operation applied to `B`. See *oneMKL defined datatypes* for more details.

offset_type Specifies the form of `C_offset` used in the matrix multiplication. See *oneMKL defined datatypes* for more details.

m Number of rows of op(`A`) and `C`. Must be at least zero.

n Number of columns of op(`B`) and `C`. Must be at least zero.

k Number of columns of op(`A`) and rows of op(`B`). Must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a The buffer holding the input matrix `A`.

	`A` not transposed	`A` transposed
Column major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*k`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*m`
Row major	`A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*m`.	`A` is an `k`-by-`m` matrix so the array `a` must have size at least `lda*k`

See *Matrix Storage* for more details.

lda The leading dimension of `A`. It must be positive.

	`A` not transposed	`A` transposed
Column major	`lda` must be at least `m`.	`lda` must be at least `k`.
Row major	`lda` must be at least `k`.	`lda` must be at least `m`.

ao Specifies the scalar offset value for matrix `A`.

b Buffer holding the input matrix `B`.

	`B` not transposed	`B` transposed
Column major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*n`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*k`
Row major	`B` is an `k`-by-`n` matrix so the array `b` must have size at least `ldb*k`.	`B` is an `n`-by-`k` matrix so the array `b` must have size at least `ldb*n`

See *Matrix Storage* for more details.

ldb The leading dimension of `B`. It must be positive.

	`B` not transposed	`B` transposed
Column major	`ldb` must be at least `k`.	`ldb` must be at least `n`.
Row major	`ldb` must be at least `n`.	`ldb` must be at least `k`.

bo Specifies the scalar offset value for matrix `B`.

beta Scaling factor for matrix `C`.

c Buffer holding the input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

co Buffer holding the offset values for matrix `C`.

If `offset_type` = `offset::fix`, the `co` array must have size at least 1.

If `offset_type` = `offset::col`, the `co` array must have size at least `max(1,m)`.

If `offset_type` = `offset::row`, the `co` array must have size at least `max(1,n)`.

Output Parameters

c Output buffer, overwritten by `alpha` * (op(`A`) - `A_offset`)*(op(`B`) - `B_offset`) + `beta` * `C` + `C_offset`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_bias`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

gemm_bias (USM Version)

Syntax

```
namespace oneapi::mkl::blas::column_major {
    sycl::event gemm_bias(sycl::queue &queue,
                          onemkl::transpose transa,
                          onemkl::transpose transb,
                          onemkl::offset offset_type,
                          std::int64_t m,
                          std::int64_t n,
                          std::int64_t k,
                          Ts alpha,
                          const Ta *a,
                          std::int64_t lda,
                          Ta ao,
                          const Tb *b,
                          std::int64_t ldb,
                          Tb bo,
                          Ts beta,
                          Tc *c,
                          std::int64_t ldc,
                          const Tc *co,
                          const sycl::vector_class<sycl::event> &dependencies = {})
}
```

```
namespace oneapi::mkl::blas::row_major {
    sycl::event gemm_bias(sycl::queue &queue,
                          onemkl::transpose transa,
                          onemkl::transpose transb,
                          onemkl::offset offset_type,
                          std::int64_t m,
                          std::int64_t n,
                          std::int64_t k,
                          Ts alpha,
                          const Ta *a,
                          std::int64_t lda,
                          Ta ao,
                          const Tb *b,
                          std::int64_t ldb,
                          Tb bo,
                          Ts beta,
                          Tc *c,
                          std::int64_t ldc,
                          const Tc *co,
                          const sycl::vector_class<sycl::event> &dependencies = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

transa Specifies op(A), the transposition operation applied to A. See *oneMKL defined datatypes* for more details.

transb Specifies op(B), the transposition operation applied to B. See *oneMKL defined datatypes* for more details.

offset_type Specifies the form of C_offset used in the matrix multiplication. See *oneMKL defined datatypes* for more details.

m Number of rows of op(A) and C. Must be at least zero.

n Number of columns of op(B) and C. Must be at least zero.

k Number of columns of op(A) and rows of op(B). Must be at least zero.

alpha Scaling factor for the matrix-matrix product.

a Pointer to input matrix A.

	A not transposed	A transposed
Column major	A is an m-by-k matrix so the array a must have size at least lda*k.	A is an k-by-m matrix so the array a must have size at least lda*m
Row major	A is an m-by-k matrix so the array a must have size at least lda*m.	A is an k-by-m matrix so the array a must have size at least lda*k

See *Matrix Storage* for more details.

lda The leading dimension of A. It must be positive.

	A not transposed	A transposed
Column major	lda must be at least m.	lda must be at least k.
Row major	lda must be at least k.	lda must be at least m.

ao Specifies the scalar offset value for matrix A.

b Pointer to input matrix B.

	B not transposed	B transposed
Column major	B is an k-by-n matrix so the array b must have size at least ldb*n.	B is an n-by-k matrix so the array b must have size at least ldb*k
Row major	B is an k-by-n matrix so the array b must have size at least ldb*k.	B is an n-by-k matrix so the array b must have size at least ldb*n

See *Matrix Storage* for more details.

ldb The leading dimension of B. It must be positive.

	B not transposed	B transposed
Column major	ldb must be at least k.	ldb must be at least n.
Row major	ldb must be at least n.	ldb must be at least k.

bo Specifies the scalar offset value for matrix B.

beta Scaling factor for matrix C.

c Pointer to input/output matrix `C`. It must have a size of at least `ldc*n` if column major layout is used to store matrices or at least `ldc*m` if row major layout is used to store matrices . See *Matrix Storage* for more details.

ldc The leading dimension of `C`. It must be positive and at least `m` if column major layout is used to store matrices or at least `n` if column major layout is used to store matrices.

co Pointer to offset values for matrix `C`.

If `offset_type` = `offset::fix`, the `co` array must have size at least 1.

If `offset_type` = `offset::col`, the `co` array must have size at least `max(1,m)`.

If `offset_type` = `offset::row`, the `co` array must have size at least `max(1,n)`.

dependencies List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

Output Parameters

c Pointer to the output matrix, overwritten by `alpha` * (op(`A`) - `A_offset`)*(op(`B`) - `B_offset`) + `beta` * `C` + `C_offset`.

Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_bias`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::invalid_argument

oneapi::mkl::unsupported_device

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

Parent topic: *BLAS-like Extensions*

Parent topic: *BLAS Routines*

Parent topic: *Dense Linear Algebra*

LAPACK Routines

oneMKL provides a DPC++ interface to select routines from the Linear Algebra PACKage (LAPACK), as well as several LAPACK-like extension routines.

LAPACK Linear Equation Routines

LAPACK Linear Equation routines are used for factoring a matrix, solving a system of linear equations, solving linear least squares problems, and inverting a matrix. The following table lists the LAPACK Linear Equation routine groups.

Routines	Scratchpad Size Routines	Description
geqrf	*geqrf_scratchpad_size*	Computes the QR factorization of a general m-by-n matrix.
gerqf	*gerqf_scratchpad_size*	Computes the RQ factorization of a general m-by-n matrix.
getrf	*getrf_scratchpad_size*	Computes the LU factorization of a general m-by-n matrix.
getri	*getri_scratchpad_size*	Computes the inverse of an LU-factored general matrix.
getrs	*getrs_scratchpad_size*	Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.
hetrf	*hetrf_scratchpad_size*	Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.
orgqr	*orgqr_scratchpad_size*	Generates the real orthogonal matrix Q of the QR factorization formed by geqrf.
ormqr	*ormqr_scratchpad_size*	Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by geqrf.
ormrq	*ormrq_scratchpad_size*	Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by gerqf.
potrf	*potrf_scratchpad_size*	Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.
potri	*potri_scratchpad_size*	Computes the inverse of a Cholesky-factored symmetric (Hermitian) positive-definite matrix.
potrs	*potrs_scratchpad_size*	Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix, with multiple right-hand sides.
sytrf	*sytrf_scratchpad_size*	Computes the Bunch-Kaufman factorization of a symmetric matrix.
trtrs	*trtrs_scratchpad_size*	Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.
ungqr	*ungqr_scratchpad_size*	Generates the complex unitary matrix Q of the QR factorization formed by geqrf.
unmqr	*unmqr_scratchpad_size*	Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by geqrf.
unmrq	*unmrq_scratchpad_size*	Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by gerqf.

geqrf

Computes the QR factorization of a general $m \times n$ matrix.

Description

`geqrf` supports the following precisions:

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine forms the QR factorization of a general $m \times n$ matrix A. No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

geqrf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void geqrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer
↪<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in A ($0 \leq n$).

a Buffer holding input matrix A. Must have size at least lda $\cdot n$.

lda The leading dimension of A; at least $\max(1, m)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *geqrf_scratchpad_size* function.

Output Parameters

a Output buffer, overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the $\min(m, n) \times n$ upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array tau, represent the orthogonal matrix Q as a product of $\min(m, n)$ elementary reflectors.

tau Output buffer, size at least $\max(1, \min(m, n))$. Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

geqrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event geqrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in A ($0 \leq n$).

a Pointer to memory holding input matrix A. Must have size at least lda $\cdot$ n.

lda The leading dimension of A; at least $\max(1, m)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *geqrf_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the $\min(m, n) \times n$ upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array tau, represent the orthogonal matrix Q as a product of $\min(m, n)$ elementary reflectors.

tau Array, size at least $\max(1, \min(m, n))$. Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

geqrf_scratchpad_size

Computes size of scratchpad memory required for *geqrf* function.

Description

`geqrf_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *geqrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t geqrf_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *geqrf* function will be performed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *geqrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

gerqf

Computes the RQ factorization of a general $m \times n$ matrix.

Description

`gerqf` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine forms the RQ factorization of a general $m \times n$ matrix A. No pivoting is performed. The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation

gerqf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void gerqf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer
↪<T> &a, std::int64_t lda, cl::sycl::buffer<T> &tau, cl::sycl::buffer<T> &scratchpad,
↪ std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

a Buffer holding input matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`, at least $\max(1, m)$.

scratchpad Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *gerqf_scratchpad_size* function.

Output Parameters

a Output buffer, overwritten by the factorization data as follows:

If $m \leq n$, the upper triangle of the subarray `a(1:m, n-m+1:n)` contains the $m \times m$ upper triangular matrix R; if $m \geq n$, the elements on and above the $(m-n)$-th subdiagonal contain the $m \times n$ upper trapezoidal matrix R

In both cases, the remaining elements, with the array `tau`, represent the orthogonal/unitary matrix Q as a product of $\min(m, n)$ elementary reflectors.

tau Array, size at least $\min(m, n)$.

Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

gerqf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event gerqf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

a Buffer holding input matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`, at least $\max(1, m)$.

scratchpad Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *gerqf_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Output buffer, overwritten by the factorization data as follows:

If $m \leq n$, the upper triangle of the subarray `a(1:m, n-m+1:n)` contains the $m \times m$ upper triangular matrix R; if $m \geq n$, the elements on and above the $(m-n)$-th subdiagonal contain the $m \times n$ upper trapezoidal matrix R

In both cases, the remaining elements, with the array `tau`, represent the orthogonal/unitary matrix Q as a product of $\min(m, n)$ elementary reflectors.

tau Array, size at least $\min(m, n)$.

Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

gerqf_scratchpad_size

Computes size of scratchpad memory required for *gerqf* function.

Description

`gerqf_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *gerqf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

gerqf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t gerqf_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by the gerqf (buffer or USM version) function will be performed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

lda The leading dimension of `a`; at least $\max(1, m)$.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *gerqf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

getrf

Computes the LU factorization of a general $m \times n$ matrix.

Description

`getrf` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine computes the LU factorization of a general $m \times n$ matrix A as $A = PLU$,

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

getrf (BUFFER Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void getrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer
↪<T,1> &a, std::int64_t lda, cl::sycl::buffer<std::int64_t,1> &ipiv, cl::sycl::buffer
↪<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in A ($0 \leq n$).

a Buffer holding input matrix A. The buffer a contains the matrix A. The second dimension of a must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getrf_scratchpad_size* function.

Output Parameters

a Overwritten by L and U. The unit diagonal elements of L are not stored.

ipiv Array, size at least $\max(1, \min(m, n))$. Contains the pivot indices; for $1 \leq i \leq \min(m, n)$, row i was interchanged with row ipiv(i).

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

If info equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

getrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, std::int64_t *ipiv, T *scratchpad, std::int64_t scratchpad_size,␣
↪const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in A ($0 \leq n$).

a Pointer to array holding input matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getrf_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by L and U. The unit diagonal elements of L are not stored.

ipiv Array, size at least $\max(1, \min(m, n))$. Contains the pivot indices; for $1 \leq i \leq \min(m, n)$, row i was interchanged with row ipiv(i).

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

If info equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

getrf_scratchpad_size

Computes size of scratchpad memory required for *getrf* function.

Description

`getrf_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *getrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

getrf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrf_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *getrf* function will be performed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in A ($0 \leq n$).

lda The leading dimension of `a` ($n \leq$ lda).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *getrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

getri

Computes the inverse of an LU-factored general matrix determined by *getrf*.

Description

`getri` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine computes the inverse A^{-1} of a general matrix A. Before calling this routine, call *getrf* to factorize A.

getri (BUFFER Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void getri(cl::sycl::queue &queue, std::int64_t n, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<std::int64_t,1> &ipiv, cl::sycl::buffer<T,1> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

n The order of the matrix A ($0 \leq n$).

a The buffer `a` as returned by *getrf*. Must be of size at least lda $\cdot \max(1, n)$.

lda The leading dimension of `a` ($n \leq$ lda).

ipiv The buffer as returned by *getrf*. The dimension of `ipiv` must be at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getri_scratchpad_size* function.

Output Parameters

a Overwritten by the $n \times n$ matrix A.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

getri (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getri(cl::sycl::queue &queue, std::int64_t n, T *a, std::int64_t␣
↪lda, std::int64_t *ipiv, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

n The order of the matrix A ($0 \leq n$).

a The array as returned by *getrf*. Must be of size at least lda $\cdot \max(1, n)$.

lda The leading dimension of `a` ($n \leq$ lda).

ipiv The array as returned by *getrf*. The dimension of `ipiv` must be at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getri_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by the $n \times n$ matrix A.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If info equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

getri_scratchpad_size

Computes size of scratchpad memory required for *getri* function.

Description

`getri_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *getri* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

getri_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getri_scratchpad_size(cl::sycl::queue &queue, std::int64_t n,␣
↪std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *getri* function will be performed.

n The order of the matrix A ($0 \leq n$).

lda The leading dimension of `a` ($n \leq$ lda).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *getri* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

getrs

Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.

Description

`getrs` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine solves for X the following systems of linear equations:

$AX = B$	if `trans=onemkl::transpose::nontrans`
$A^T X = B$	if `trans=onemkl::transpose::trans`
$A^H X = B$	if `trans=onemkl::transpose::conjtrans`

Before calling this routine, you must call *getrf* to compute the LU factorization of A.

getrs (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void getrs(cl::sycl::queue &queue, onemkl::transpose trans, std::int64_t n,␣
↪std::int64_t nrhs, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer
↪<std::int64_t,1> &ipiv, cl::sycl::buffer<T,1> &b, std::int64_t ldb, cl::sycl::buffer
↪<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

trans Indicates the form of the equations:

If `trans=onemkl::transpose::nontrans`, then $AX = B$ is solved for X.

If `trans=onemkl::transpose::trans`, then $A^T X = B$ is solved for X.

If `trans=onemkl::transpose::conjtrans`, then $A^H X = B$ is solved for X.

n The order of the matrix A and the number of rows in matrix $B (0 \leq n)$.

nrhs The number of right-hand sides ($0 \leq$ nrhs).

a Buffer containing the factorization of the matrix A, as returned by *getrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

ipiv Array, size at least $\max(1, n)$. The `ipiv` array, as returned by *getrf*.

b The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, \text{nrhs})$.

ldb The leading dimension of `b`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getrs_scratchpad_size* function.

Output Parameters

b The buffer `b` is overwritten by the solution matrix X.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If `info=-i`, the i-th parameter had an illegal value.
>
> If `info=i`, the i-th diagonal element of U is zero, and the solve could not be completed.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

getrs (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrs(cl::sycl::queue &queue, onemkl::transpose trans, std::int64_t␣
↪n, std::int64_t nrhs, T *a, std::int64_t lda, std::int64_t *ipiv, T *b, std::int64_
↪t ldb, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class
↪<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

trans Indicates the form of the equations:

If `trans=onemkl::transpose::nontrans`, then $AX = B$ is solved for X.

If `trans=onemkl::transpose::trans`, then $A^T X = B$ is solved for X.

If `trans=onemkl::transpose::conjtrans`, then $A^H X = B$ is solved for X.

n The order of the matrix A and the number of rows in matrix $B (0 \leq n)$.

nrhs The number of right-hand sides ($0 \leq$ nrhs).

a Pointer to array containing the factorization of the matrix A, as returned by *getrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

ipiv Array, size at least $\max(1, n)$. The `ipiv` array, as returned by *getrf*.

b The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, \text{nrhs})$.

ldb The leading dimension of `b`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *getrs_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b The array `b` is overwritten by the solution matrix X.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If `info=-i`, the i-th parameter had an illegal value.
>
> If `info=i`, the i-th diagonal element of U is zero, and the solve could not be completed.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

getrs_scratchpad_size

Computes size of scratchpad memory required for *getrs* function.

Description

`getrs_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *getrs* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

getrs_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrs_scratchpad_size(cl::sycl::queue &queue, onemkl::transpose trans,␣
↪std::int64_t n, std::int64_t nrhs, std::int64_t lda, std::int64_t ldb)
}
```

Input Parameters

queue Device queue where calculations by *getrs* function will be performed.

trans Indicates the form of the equations:

If `trans=onemkl::transpose::nontrans`, then $AX = B$ is solved for X.

If `trans=onemkl::transpose::trans`, then $A^T X = B$ is solved for X.

If `trans=onemkl::transpose::conjtrans`, then $A^H X = B$ is solved for X.

n The order of the matrix A $(0 \leq n)$ and the number of rows in matrix $B(0 \leq n)$.

nrhs The number of right-hand sides ($0 \leq$ nrhs).

lda The leading dimension of `a`.

ldb The leading dimension of `b`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *getrs* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

hetrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Description

`hetrf` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

- if `upper_lower=uplo::upper`, $A = UDU^H$
- if `upper_lower=uplo::lower`, $A = LDL^H$

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a Hermitian block-diagonal matrix with 1×1 and 2×2 diagonal blocks. U and L have 2×2 unit diagonal blocks corresponding to the 2×2 blocks of D.

hetrf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void hetrf(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<int_64,1> &ipiv,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the buffer `a` stores the upper triangular part of the matrix A, and A is factored as UDU^H.

If `upper_lower=uplo::lower`, the buffer `a` stores the lower triangular part of the matrix A, and A is factored as LDL^H.

n The order of matrix A ($0 \leq n$).

a The buffer `a`, size $\max(1, \text{lda} \cdot n)$. The buffer `a` contains either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hetrf_scratchpad_size* function.

Output Parameters

a The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv Buffer, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If $\text{ipiv}(i) = k > 0$, then d_{ii} is a 1×1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=oneapi::mkl::uplo::upper` and $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$, then D has a 2×2 block in rows/columns i and i-1, and $(i-1)$-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=oneapi::mkl::uplo::lower` and $\text{ipiv}(i) = \text{ipiv}(i+1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If info $= i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

hetrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event hetrf(cl::sycl::queue &queue, oneapi::mkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, int_64 *ipiv, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and A is factored as UDU^H.

If `upper_lower=uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and A is factored as LDL^H.

n The order of matrix A ($0 \leq n$).

a The pointer to A, size $\max(1, \text{lda} \cdot n)$, containing either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad Pointer to scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hetrf_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv Pointer to array of size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If $\text{ipiv}(i) = k > 0$, then d_{ii} is a 1×1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=oneapi::mkl::uplo::upper` and $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i-1$, and $(i-1)$-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=oneapi::mkl::uplo::lower` and $\text{ipiv}(i) = \text{ipiv}(i+1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If info $= i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

hetrf_scratchpad_size

Computes size of scratchpad memory required for *hetrf* function.

Description

`hetrf_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *hetrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

hetrf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t hetrf_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::uplo upper_
↪lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *hetrf* function will be performed.

upper_lower Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the buffer `a` stores the upper triangular part of the matrix A, and A is factored as UDU^H.

If `upper_lower=uplo::lower`, the buffer `a` stores the lower triangular part of the matrix A, and A is factored as LDL^H

n The order of the matrix A ($0 \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *hetrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

orgqr

Generates the real orthogonal matrix Q of the QR factorization formed by *geqrf*.

Description

`orgqr` supports the following precisions.

T
`float`
`double`

The routine generates the whole or part of $m \times m$ orthogonal matrix Q of the QR factorization formed by the routine *geqrf*.

Usually Q is determined from the QR factorization of an `m` by `p` matrix A with $m \geq p$. To compute the whole matrix Q, use:

```
oneapi::mkl::lapack::orgqr(queue, m, m, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
oneapi::mkl::lapack::orgqr(queue, m, p, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the matrix Q^k of the QR factorization of leading k columns of the matrix A:

```
oneapi::mkl::lapack::orgqr(queue, m, m, k, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by leading k columns of the matrix A):

```
oneapi::mkl::lapack::orgqr(queue, m, k, k, a, lda, tau, scratchpad, scratchpad_size)
```

orgqr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void orgqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The buffer `a` as returned by *geqrf*.

lda The leading dimension of `a` (lda $\leq m$).

tau The buffer `tau` as returned by *geqrf*.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgqr_scratchpad_size* function.

Output Parameters

a Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

orgqr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The pointer to `a` as returned by *geqrf*.

lda The leading dimension of `a` (lda $\leq m$).

tau The pointer to `tau` as returned by *geqrf*.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgqr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

orgqr_scratchpad_size

Computes size of scratchpad memory required for *orgqr* function.

Description

`orgqr_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *orgqr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

orgqr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgqr_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *orgqr* function will be performed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *orgqr* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

ormqr

Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by *geqrf*.

Description

`ormqr` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q of the QR factorization formed by the routine *geqrf*.

Depending on the parameters `left_right` and `trans`, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result on C).

ormqr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ormqr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::transpose trans,
↪ std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c, std::int64_
↪t ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right If `left_right=onemkl::side::left`, Q or Q^T is applied to C from the left.

If `left_right=onemkl::side::right`, Q or Q^T is applied to C from the right.

trans If `trans=onemkl::transpose::trans`, the routine multiplies C by Q.

If `trans=onemkl::transpose::nontrans`, the routine multiplies C by Q^T.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The buffer `a` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

lda The leading dimension of `a`.

tau The buffer `tau` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

c The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ormqr_scratchpad_size* function.

Output Parameters

c Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by left_right and trans).

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ormqr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ormqr(cl::sycl::queue &queue, onemkl::side left_right,␣
↪onemkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right If `left_right=onemkl::side::left`, Q or Q^T is applied to C from the left.

> If `left_right=onemkl::side::right`, Q or Q^T is applied to C from the right.

trans If `trans=onemkl::transpose::trans`, the routine multiplies C by Q.

> If `trans=onemkl::transpose::nontrans`, the routine multiplies C by Q^T.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The pointer to `a` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

lda The leading dimension of `a`.

tau The pointer to `tau` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

c The pointer to the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ormqr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by left_right and trans).

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

ormqr_scratchpad_size

Computes size of scratchpad memory required for *ormqr* function.

Description

`ormqr_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *ormqr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ormqr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ormqr_scratchpad_size(cl::sycl::queue &queue, onemkl::side left_right,␣
↪onemkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, std::int64_
↪t lda, std::int64_t ldc, std::int64_t &scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations by *ormqr* function will be performed.

left_right If `left_right=onemkl::side::left`, Q or Q^T is applied to C from the left.

If `left_right=onemkl::side::right`, Q or Q^T is applied to C from the right.

trans If `trans=onemkl::transpose::trans`, the routine multiplies C by Q.

If `trans=onemkl::transpose::nontrans`, the routine multiplies C by Q^T.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda The leading dimension of `a`.

ldc The leading dimension of `c`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ormqr* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

ormrq

Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by *gerqf*.

Description

`ormrq` supports the following precisions.

T
`float`
`double`

The routine multiplies a real $m \times n$ matrix C by Q or Q^T, where Q is the real orthogonal matrix defined as a product of k elementary reflectors $H_i : Q = H_1 H_2 ... H_k$ as returned by the RQ factorization routine *gerqf*. Depending on the parameters `side` and `trans`, the routine can form one of the matrix products QC, $Q^T C$, CQ, or CQ^T (overwriting the result over C).

ormrq (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ormrq(cl::sycl::queue &queue, oneapi::mkl::side side, oneapi::mkl::transpose␣
↪trans, std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c, std::int64_
↪t ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

side If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans If `trans=oneapi::mkl::transpose::trans`, the routine multiplies C by Q.

If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q^T.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a Buffer holding the result of the *gerqf* function. The second dimension of `a` must be at least $\max(1, k)$.

lda The leading dimension of `a`.

tau Buffer holding `tau` returned by the *gerqf* function.

c Buffer holding the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c`.

scratchpad Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *ormrq_scratchpad_size* function.

Output Parameters

c Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ormrq (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ormrq(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

side If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans If `trans=oneapi::mkl::transpose::trans`, the routine multiplies C by Q.

If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q^T.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a Buffer holding the result of the *gerqf* function. The second dimension of `a` must be at least $\max(1, k)$.

lda The leading dimension of `a`.

tau Buffer holding `tau` returned by the *gerqf* function.

c Buffer holding the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c`.

scratchpad Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the *ormrq_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If `info = -i`, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

ormrq_scratchpad_size

Computes size of scratchpad memory required for *ormrq* function.

Description

`ormrq_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *ormrq* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ormrq_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ormrq_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪std::int64_t lda, std::int64_t ldc);
}
```

Input Parameters

queue Device queue where calculations by the ormrq function will be performed.

side If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans If `trans=oneapi::mkl::transpose::trans`, the routine multiplies C by Q.

If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q^T.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda The leading dimension of `a`.

ldc The leading dimension of `c`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ormrq* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

Description

`potrf` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A:

$A = U^T U$ for real data, $A = U^H U$ for complex data	if upper_lower=`onemkl::uplo::upper`
$A = LL^T$ for real data, $A = LL^H$ for complex data	if upper_lower=`onemkl::uplo::lower`

where L is a lower triangular matrix and U is upper triangular.

potrf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void potrf(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &scratchpad,␣
↪std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If upper_lower=`onemkl::uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and the strictly lower triangular part of the matrix is not referenced.

If upper_lower=`onemkl::uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and the strictly upper triangular part of the matrix is not referenced.

n Specifies the order of the matrix A ($0 \leq n$).

a Buffer holding input matrix A. The buffer `a` contains either the upper or the lower triangular part of the matrix A (see upper_lower). The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potrf_scratchpad_size* function.

Output Parameters

a The buffer `a` is overwritten by the Cholesky factor U or L, as specified by `upper_lower`.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, and *detail()* returns 0, then the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

potrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrf(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_
↪t n, T *a, std::int64_t lda, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If upper_lower=`onemkl::uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and the strictly lower triangular part of the matrix is not referenced.

If upper_lower=`onemkl::uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and the strictly upper triangular part of the matrix is not referenced.

n Specifies the order of the matrix A ($0 \leq n$).

a Pointer to input matrix A. The array `a` contains either the upper or the lower triangular part of the matrix A (see upper_lower). The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potrf_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a The memory pointer to by pointer `a` is overwritten by the Cholesky factor U or L, as specified by `upper_lower`.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, and *detail()* returns 0, then the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

potrf_scratchpad_size

Computes size of scratchpad memory required for *potrf* function.

Description

`potrf_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *potrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

potrf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrf_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
↪ std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *potrf* function will be performed.

upper_lower Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower = onemkl::uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and the strictly lower triangular part of the matrix is not referenced.

If `upper_lower = onemkl::uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and the strictly upper triangular part of the matrix is not referenced.

n Specifies the order of the matrix A ($0 \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *potrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

potri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix using the Cholesky factorization.

Description

`potri` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine computes the inverse A^{-1} of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A. Before calling this routine, call *potrf* to factorize A.

potri (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void potri(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &scratchpad,␣
↪std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates how the input matrix A has been factored:

If `upper_lower = onemkl::uplo::upper`, the upper triangle of A is stored.

If `upper_lower = onemkl::uplo::lower`, the lower triangle of A is stored.

n Specifies the order of the matrix A ($0 \leq n$).

a Contains the factorization of the matrix A, as returned by *potrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potri_scratchpad_size* function.

Output Parameters

a Overwritten by the upper or lower triangle of the inverse of A. Specified by `upper_lower`.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, the i-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

potri (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potri(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_
↪t n, T *a, std::int64_t lda, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates how the input matrix A has been factored:

If `upper_lower = onemkl::uplo::upper`, the upper triangle of A is stored.

If `upper_lower = onemkl::uplo::lower`, the lower triangle of A is stored.

n Specifies the order of the matrix A ($0 \leq n$).

a Contains the factorization of the matrix A, as returned by *potrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potri_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by the upper or lower triangle of the inverse of A. Specified by `upper_lower`.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, the i-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

potri_scratchpad_size

Computes size of scratchpad memory required for *potri* function.

Description

`potri_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *potri* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

potri_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potri_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
↪ std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *potri* function will be performed.

upper_lower Indicates how the input matrix A has been factored:

If `upper_lower = onemkl::uplo::upper`, the upper triangle of A is stored.

If `upper_lower = onemkl::uplo::lower`, the lower triangle of A is stored.

n Specifies the order of the matrix A ($0 \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *potri* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

Description

`potrs` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine solves for X the system of linear equations $AX = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A, given the Cholesky factorization of A:

$A = U^T U$ for real data, $A = U^H U$ for complex data	if `upper_lower=onemkl::uplo::upper`
$A = LL^T$ for real data, $A = LL^H$ for complex data	if `upper_lower=onemkl::uplo::lower`

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

Before calling this routine, you must call *potrf* to compute the Cholesky factorization of A.

potrs (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void potrs(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,␣
↪std::int64_t nrhs, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1>
↪ &b, std::int64_t ldb, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_
↪size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates how the input matrix has been factored:

If `upper_lower = onemkl::uplo::upper`, the upper triangle U of A is stored, where $A = U^T$ for real data, $A = U^H U$ for complex data.

If `upper_lower = onemkl::uplo::lower`, the lower triangle L of A is stored, where $A = LL^T$ for real data, $A = LL^H$ for complex data.

n The order of matrix A ($0 \leq n$).

nrhs The number of right-hand sides ($0 \leq$ nrhs).

a Buffer containing the factorization of the matrix A, as returned by *potrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

b The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, \text{nrhs})$.

ldb The leading dimension of `b`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potrs_scratchpad_size* function.

Output Parameters

b Overwritten by the solution matrix X.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, the i-th diagonal element of the Cholesky factor is zero, and the solve could not be completed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

potrs (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrs(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_
↪t n, std::int64_t nrhs, T *a, std::int64_t lda, T *b, std::int64_t ldb, T␣
↪*scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class
↪<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates how the input matrix has been factored:

If `upper_lower = onemkl::uplo::upper`, the upper triangle U of A is stored, where $A = U^T U$ for real data, $A = U^H U$ for complex data.

If `upper_lower = onemkl::uplo::lower`, the lower triangle L of A is stored, where $A = LL^T$ for real data, $A = LL^H$ for complex data.

n The order of matrix A ($0 \leq n$).

nrhs The number of right-hand sides ($0 \leq$ nrhs).

a Pointer to array containing the factorization of the matrix A, as returned by *potrf*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

b The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` must be at least $\max(1, \text{nrhs})$.

ldb The leading dimension of `b`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *potrs_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b Overwritten by the solution matrix X.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If info $= i$, the i-th diagonal element of the Cholesky factor is zero, and the solve could not be completed.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

potrs_scratchpad_size

Computes size of scratchpad memory required for *potrs* function.

Description

`potrs_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *potrs* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

potrs_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrs_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
↪ std::int64_t n, std::int64_t nrhs, std::int64_t lda, std::int64_t ldb)
}
```

Input Parameters

queue Device queue where calculations by *potrs* function will be performed.

upper_lower Indicates how the input matrix has been factored:

If `upper_lower = onemkl::uplo::upper`, the upper triangle U of A is stored, where $A = U^T U$ for real data, $A = U^H U$ for complex data.

If `upper_lower = onemkl::uplo::lower`, the lower triangle L of A is stored, where $A = LL^T$ for real data, $A = LL^H$ for complex data.

n The order of matrix A ($0 \leq n$).

nrhs The number of right-hand sides ($0 \leq nrhs$).

lda The leading dimension of `a`.

ldb The leading dimension of `b`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *potrs* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

sytrf

Computes the Bunch-Kaufman factorization of a symmetric matrix.

Description

`sytrf` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine computes the factorization of a real/complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

- if `upper_lower=uplo::upper`, $A = UDU^T$
- if `upper_lower=uplo::lower`, $A = LDL^T$

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1×1 and 2×2 diagonal blocks. U and L have 2×2 unit diagonal blocks corresponding to the 2×2 blocks of D.

sytrf (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void sytrf(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<int_64,1> &ipiv,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the buffer `a` stores the upper triangular part of the matrix A, and A is factored as UDU^T.

If `upper_lower=uplo::lower`, the buffer `a` stores the lower triangular part of the matrix A, and A is factored as LDL^T.

n The order of matrix A ($0 \leq n$).

a The buffer `a`, size $\max(1, lda \cdot n)$. The buffer `a` contains either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sytrf_scratchpad_size* function.

Output Parameters

a The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv Buffer, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If ipiv$(i) = k > 0$, then d_{ii} is a 1×1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=onemkl::uplo::upper` and ipiv$(i) =$ ipiv$(i-1) = -m < 0$, then D has a 2×2 block in rows/columns i and i-1, and $(i-1)$-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=onemkl::uplo::lower` and ipiv$(i) =$ ipiv$(i+1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

sytrf (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event sytrf(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_
↪t n, T *a, std::int64_t lda, int_64 *ipiv, T *scratchpad, std::int64_t scratchpad_
↪size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the array `a` stores the upper triangular part of the matrix A, and A is factored as UDU^T.

If `upper_lower=uplo::lower`, the array `a` stores the lower triangular part of the matrix A, and A is factored as LDL^T.

n The order of matrix A ($0 \leq n$).

a The pointer to A, size $\max(1, \text{lda} \cdot n)$, containing either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sytrf_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv Pointer to array of size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If $\text{ipiv}(i) = k > 0$, then d_{ii} is a 1×1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=onemkl::uplo::upper` and $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i-1$, and $(i-1)$-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=onemkl::uplo::lower` and $\text{ipiv}(i) = \text{ipiv}(i+1) = -m < 0$, then D has a 2×2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If info $= i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

sytrf_scratchpad_size

Computes size of scratchpad memory required for *sytrf* function.

Description

`sytrf_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *sytrf* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

sytrf_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t sytrf_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
↪ std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *sytrf* function will be performed.

upper_lower Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the buffer `a` stores the upper triangular part of the matrix A, and A is factored as UDU^T.

If `upper_lower=uplo::lower`, the buffer `a` stores the lower triangular part of the matrix A, and A is factored as LDL^T

n The order of the matrix A ($0 \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *sytrf* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

trtrs

Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.

Description

`trtrs` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine solves for X the following systems of linear equations with a triangular matrix A, with multiple right-hand sides stored in B:

$AX = B$		if `transa` =`transpose::nontrans`,
$A^TX = B$		if `transa` =`transpose::trans`,
$A^HX = B$		if `transa` =`transpose::conjtrans` (for complex matrices only).

trtrs (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void trtrs(cl::sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose␣
↪transa, onemkl::diag unit_diag, std::int64_t n, std::int64_t nrhs, cl::sycl::buffer
↪<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &b, std::int64_t ldb,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates whether A is upper or lower triangular:

If upper_lower = `uplo::upper`, then A is upper triangular.

If upper_lower = `uplo::lower`, then A is lower triangular.

transa If transa = `transpose::nontrans`, then $AX = B$ is solved for X.

If transa = `transpose::trans`, then $A^TX = B$ is solved for X.

If transa = `transpose::conjtrans`, then $A^HX = B$ is solved for X.

unit_diag If unit_diag = `diag::nonunit`, then A is not a unit triangular matrix.

If unit_diag = `diag::unit`, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array `a`.

n The order of A; the number of rows in B; $n \geq 0$.

nrhs The number of right-hand sides; nrhs ≥ 0.

a Buffer containing the matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; lda $\geq \max(1, n)$.

b Buffer containing the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` at least $\max(1, \text{nrhs})$.

ldb The leading dimension of `b`; ldb $\geq \max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *trtrs_scratchpad_size* function.

Output Parameters

b Overwritten by the solution matrix X.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

trtrs (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event trtrs(cl::sycl::queue &queue, onemkl::uplo upper_lower,␣
↪onemkl::transpose transa, onemkl::diag unit_diag, std::int64_t n, std::int64_t nrhs,
↪ T *a, std::int64_t lda, T *b, std::int64_t ldb, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Indicates whether A is upper or lower triangular:

If upper_lower = `uplo::upper`, then A is upper triangular.

If upper_lower = `uplo::lower`, then A is lower triangular.

transa If transa = `transpose::nontrans`, then $AX = B$ is solved for X.

If transa = `transpose::trans`, then $A^T X = B$ is solved for X.

If transa = `transpose::conjtrans`, then $A^H X = B$ is solved for X.

unit_diag If unit_diag = `diag::nonunit`, then A is not a unit triangular matrix.

If unit_diag = `diag::unit`, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array `a`.

n The order of A; the number of rows in B; $n \geq 0$.

nrhs The number of right-hand sides; nrhs ≥ 0.

a Array containing the matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; lda $\geq \max(1, n)$.

b Array containing the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of `b` at least $\max(1, \text{nrhs})$.

ldb The leading dimension of `b`; ldb $\geq \max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *trtrs_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b Overwritten by the solution matrix X.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

trtrs_scratchpad_size

Computes size of scratchpad memory required for *trtrs* function.

Description

`trtrs_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *trtrs* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

trtrs_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t trtrs_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
↪ onemkl::transpose trans, onemkl::diag diag, std::int64_t n, std::int64_t nrhs,␣
↪std::int64_t lda, std::int64_t ldb)
}
```

Input Parameters

queue Device queue where calculations by *trtrs* function will be performed.

upper_lower Indicates whether A is upper or lower triangular:

If upper_lower = `uplo::upper`, then A is upper triangular.

If upper_lower = `uplo::lower`, then A is lower triangular.

trans Indicates the form of the equations:

If `trans=onemkl::transpose::nontrans`, then $AX = B$ is solved for X.

If `trans=onemkl::transpose::trans`, then $A^T X = B$ is solved for X.

If `trans=onemkl::transpose::conjtrans`, then $A^H X = B$ is solved for X.

diag If diag = `onemkl::diag::nonunit`, then A is not a unit triangular matrix.

If unit_diag = `diag::unit`, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array `a`.

n The order of A; the number of rows in B; $n \geq 0$.

nrhs The number of right-hand sides ($0 \leq$ nrhs).

lda The leading dimension of `a`; lda $\geq \max(1, n)$.

ldb The leading dimension of `b`; ldb $\geq \max(1, n)$.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *trtrs* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

ungqr

Generates the complex unitary matrix Q of the QR factorization formed by *geqrf*.

Description

`ungqr` supports the following precisions.

T
std::complex<float>
std::complex<double>

The routine generates the whole or part of $m \times m$ unitary matrix Q of the QR factorization formed by the routines *geqrf*.

Usually Q is determined from the QR factorization of an $m \times p$ matrix A with $m \geq p$. To compute the whole matrix Q, use:

```
oneapi::mkl::lapack::ungqr(queue, m, m, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
oneapi::mkl::lapack::ungqr(queue, m, p, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the matrix Q^k of the QR factorization of the leading k columns of the matrix A:

```
oneapi::mkl::lapack::ungqr(queue, m, m, k, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by the leading k columns of the matrix A):

```
oneapi::mkl::lapack::ungqr(queue, m, k, k, a, lda, tau, scratchpad, scratchpad_size)
```

ungqr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ungqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The buffer `a` as returned by *geqrf*.

lda The leading dimension of `a` (lda $\leq m$).

tau The buffer `tau` as returned by *geqrf*.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ungqr_scratchpad_size* function.

Output Parameters

a Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ungqr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The pointer to `a` as returned by *geqrf*.

lda The leading dimension of `a` (lda $\leq m$).

tau The pointer to `tau` as returned by *geqrf*.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ungqr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

ungqr_scratchpad_size

Computes size of scratchpad memory required for *ungqr* function.

Description

`ungqr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *ungqr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ungqr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungqr_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *ungqr* function will be performed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ungqr* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by *unmqr*.

Description

`unmqr` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine multiplies a rectangular complex matrix $C \times Q$ or Q^H, where Q is the unitary matrix Q of the `QR` factorization formed by the routines *geqrf*.

Depending on the parameters `left_right` and `trans`, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result on C).

unmqr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void unmqr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::transpose trans,
↪ std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c, std::int64_
↪t ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right If `left_right=onemkl::side::left`, Q or Q^H is applied to C from the left.

If `left_right=onemkl::side::right`, Q or Q^H is applied to C from the right.

trans If `trans=onemkl::transpose::trans`, the routine multiplies C by Q.

If `trans=onemkl::transpose::nontrans`, the routine multiplies C by Q^H.

m The number of rows in the matrix A ($m \geq 0$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The buffer `a` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

lda The leading dimension of `a`.

tau The buffer `tau` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

c The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *unmqr_scratchpad_size* function.

Output Parameters

c Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by `left_right` and `trans`).

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

unmqr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event unmqr(cl::sycl::queue &queue, onemkl::side left_right,␣
↪onemkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

left_right If `left_right=onemkl::side::left`, Q or Q^H is applied to C from the left.

If `left_right=onemkl::side::right`, Q or Q^H is applied to C from the right.

trans If `trans=onemkl::transpose::trans`, the routine multiplies C by Q.

If `trans=onemkl::transpose::nontrans`, the routine multiplies C by Q^H.

m The number of rows in the matrix A ($m \geq 0$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The pointer to `a` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

lda The leading dimension of `a`.

tau The pointer to `tau` as returned by *geqrf*. The second dimension of `a` must be at least $\max(1, k)$.

c The array `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *unmqr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by `left_right` and `trans`).

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

unmqr_scratchpad_size

Computes size of scratchpad memory required for *unmqr* function.

Description

`unmqr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *unmqr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

unmqr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t unmqr_scratchpad_size(cl::sycl::queue &queue, onemkl::side left_right,␣
↪onemkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, std::int64_
↪t lda, std::int64_t ldc, std::int64_t &scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations by *unmqr* function will be performed.

left_right If `left_right=onemkl::side::left`, Q or Q^H is applied to C from the left.

If `left_right=onemkl::side::right`, Q or Q^H is applied to C from the right.

trans If `trans=onemkl::transpose::trans`, the routine multiplies C by Q.

If `trans=onemkl::transpose::conjtrans`, the routine multiplies C by Q^H.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda The leading dimension of `a`.

ldc The leading dimension of `c`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *unmqr* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

unmrq

Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by *gerqf*.

Description

`unmrq` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine multiplies a complex $m \times n$ matrix C by Q or Q^H, where Q is the complex unitary matrix defined as a product of k elementary reflectors $H(i)$ of order n: $Q = H(1)^H H(2)^H ... H(k)^H$ Has returned by the RQ factorization routine *gerqf*.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products QC, $Q^H C$, CQ, or CQ^H (overwriting the result over C).

unmrq (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void unmrq(cl::sycl::queue &queue, oneapi::mkl::side side, oneapi::mkl::transpose␣
↪trans, std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T,1> &a,␣
↪std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c, std::int64_
↪t ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

side If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans If `trans=oneapi::mkl::transpose::trans`, the routine multiplies C by Q.

If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q^T.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a Buffer holding the result of the *gerqf* function. The second dimension of `a` must be at least $\max(1, k)$.

lda The leading dimension of `a`.

tau Buffer holding `tau` returned by the *gerqf* function.

c Buffer holding the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c`.

scratchpad Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the unmrq_scratchpad_size function.

Output Parameters

c Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

unmrq (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event unmrq(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

side If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left.

If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans If `trans=oneapi::mkl::transpose::trans`, the routine multiplies C by Q.

If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q^T.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a Buffer holding the result of the *gerqf* function. The second dimension of `a` must be at least $\max(1, k)$.

lda The leading dimension of `a`.

tau Buffer holding `tau` returned by the *gerqf* function.

c Buffer holding the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c`.

scratchpad Buffer holding scratchpad memory to be used by the routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the unmrq_scratchpad_size function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `side` and `trans`).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Linear Equation Routines*

unmrq_scratchpad_size

Computes size of scratchpad memory required for *unmrq* function.

Description

`unmrq_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *unmrq* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

unmrq_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t unmrq_scratchpad_size(cl::sycl::queue &queue, oneapi::mkl::side side,␣
↪oneapi::mkl::transpose trans, std::int64_t m, std::int64_t n, std::int64_t k,␣
↪std::int64_t lda, std::int64_t ldc)
}
```

Input Parameters

queue Device queue where calculations by the unmrq function will be performed.

side If `side = oneapi::mkl::side::left`, Q or Q^T is applied to C from the left. If `side = oneapi::mkl::side::right`, Q or Q^T is applied to C from the right.

trans If `trans=oneapi::mkl::transpose::trans`, the routine multiplies C by Q.

If `trans=oneapi::mkl::transpose::nontrans`, the routine multiplies C by Q^T.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n \leq m$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda The leading dimension of `a`.

ldc The leading dimension of `c`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *unmrq* function should be able to hold.

Parent topic: *LAPACK Linear Equation Routines*

LAPACK Singular Value and Eigenvalue Problem Routines

LAPACK Singular Value and Eigenvalue Problem routines are used for singular value and eigenvalue problems, and for performing a number of related computational tasks. The following table lists the LAPACK Singular Value and Eigenvalue Problem routine groups.

Routines	Scratchpad Size Routines	Description
gebrd	*gebrd_scratchpad_size*	Reduces a general matrix to bidiagonal form.
gesvd	*gesvd_scratchpad_size*	Computes the singular value decomposition of a general rectangular matrix.
heevd	*heevd_scratchpad_size*	Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.
hegvd	*hegvd_scratchpad_size*	Computes all eigenvalues and, optionally, all eigenvectors of a complex generalized Hermitian definite eigenproblem using divide and conquer algorithm.
hetrd	*hetrd_scratchpad_size*	Reduces a complex Hermitian matrix to tridiagonal form.
orgbr	*orgbr_scratchpad_size*	Generates the real orthogonal matrix Q or P^T determined by gebrd.
orgtr	*orgtr_scratchpad_size*	Generates the real orthogonal matrix Q determined by sytrd.
ormtr	*ormtr_scratchpad_size*	Multiplies a real matrix by the orthogonal matrix Q determined by sytrd.
syevd	*syevd_scratchpad_size*	Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.
sygvd	*sygvd_scratchpad_size*	Computes all eigenvalues and, optionally, all eigenvectors of a real generalized symmetric definite eigenproblem using divide and conquer algorithm.
sytrd	*sytrd_scratchpad_size*	Reduces a real symmetric matrix to tridiagonal form.
ungbr	*ungbr_scratchpad_size*	Generates the complex unitary matrix Q or P^T determined by gebrd.
ungtr	*ungtr_scratchpad_size*	Generates the complex unitary matrix Q determined by hetrd.
unmtr	*unmtr_scratchpad_size*	Multiplies a complex matrix by the unitary matrix Q determined by hetrd.

gebrd

Reduces a general matrix to bidiagonal form.

Description

`gebrd` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

The routine reduces a general $m \times n$ matrix A to a bidiagonal matrix B by an orthogonal (unitary) transformation.

If $m \geq n$, the reduction is given by $A = QBP^H = \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P_H$

where B_1 is an $n \times n$ upper diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; Q_1 consists of the first n columns of Q.

If $m < n$, the reduction is given by

$$A = QBP^H = Q \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P_1^H,$$

where B_1 is an $m \times m$ lower diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; P_1 consists of the first m columns of P.

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices Q and P in this representation:

If the matrix A is real,

- to compute Q and P explicitly, call *orgbr*.

If the matrix A is complex,

- to compute Q and P explicitly, call *ungbr*

gebrd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void gebrd(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer
↪<T,1> &a, std::int64_t lda, cl::sycl::buffer<realT,1> &d, cl::sycl::buffer<realT,1>␣
↪&e, cl::sycl::buffer<T,1> &tauq, cl::sycl::buffer<T,1> &taup, cl::sycl::buffer<T,1>␣
↪&scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

a The buffer a, size (`lda, *`). The buffer `a` contains the matrix A. The second dimension of `a` must be at least $\max(1, m)$.

lda The leading dimension of a.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *gebrd_scratchpad_size* function.

Output Parameters

a If $m \geq n$, the diagonal and first super-diagonal of a are overwritten by the upper bidiagonal matrix B. The elements below the diagonal, with the buffer tauq, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the buffer `taup`, represent the orthogonal matrix P as a product of elementary reflectors.

If $m < n$, the diagonal and first sub-diagonal of a are overwritten by the lower bidiagonal matrix B. The elements below the first subdiagonal, with the buffer tauq, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the buffer `taup`, represent the orthogonal matrix P as a product of elementary reflectors.

d Buffer, size at least $\max(1, \min(m, n))$. Contains the diagonal elements of B.

e Buffer, size at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of B.

tauq Buffer, size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix Q.

taup Buffer, size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix P.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

gebrd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event gebrd(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, RealT *d, RealT *e, T *tauq, T *taup, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

a Pointer to matrix A. The second dimension of `a` must be at least $\max(1, m)$.

lda The leading dimension of `a`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by *gebrd_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a If $m \geq n$, the diagonal and first super-diagonal of a are overwritten by the upper bidiagonal matrix B. The elements below the diagonal, with the array tauq, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array `taup`, represent the orthogonal matrix P as a product of elementary reflectors.

If $m < n$, the diagonal and first sub-diagonal of a are overwritten by the lower bidiagonal matrix B. The elements below the first subdiagonal, with the array tauq, represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array `taup`, represent the orthogonal matrix P as a product of elementary reflectors.

d Pointer to memory of size at least $\max(1, \min(m, n))$. Contains the diagonal elements of B.

e Pointer to memory of size at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of B.

tauq Pointer to memory of size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix Q.

taup Pointer to memory of size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix P.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If `info=-i`, the `i`-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

gebrd_scratchpad_size

Computes size of scratchpad memory required for *gebrd* function.

Description

`gebrd_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *gebrd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t gebrd_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *gebrd* function will be performed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *gebrd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

gesvd

Computes the singular value decomposition of a general rectangular matrix.

Description

`gesvd` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

gesvd (Buffer Version)

Description

The routine computes the singular value decomposition (SVD) of a real/complex $m \times n$ matrix A, optionally computing the left and/or right singular vectors. The SVD is written as

$A = U\Sigma V^T$ for real routines

$A = U\Sigma V^H$ for complex routines

where Σ is an $m \times n$ diagonal matrix, U is an $m \times m$ orthogonal/unitary matrix, and V is an $n \times n$ orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A.

Syntax

```
namespace oneapi::mkl::lapack {
  void gesvd(cl::sycl::queue &queue, onemkl::job jobu, onemkl::job jobvt, std::int64_
↪t m, std::int64_t n, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer
↪<realT,1> &s, cl::sycl::buffer<T,1> &u, std::int64_t ldu, cl::sycl::buffer<T,1> &vt,
↪ std::int64_t ldvt, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

jobu Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix U.

If `jobu = job::allvec`, all m columns of U are returned in the buffer `u`;

if `jobu = job::somevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the buffer `u`;

if `jobu = job::overwritevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the buffer a;

if `jobu = job::novec`, no columns of U (no left singular vectors) are computed.

jobvt Must be `job::allvec, job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix V^T/V^H.

If `jobvt = job::allvec`, all n columns of V^T/V^H are returned in the buffer vt;

if `jobvt = job::somevec`, the first $\min(m, n)$ columns of V^T/V^H (the left singular vectors) are returned in the buffer vt;

if `jobvt = job::overwritevec`, the first $\min(m, n)$ columns of V^T/V^H (the left singular vectors) are overwritten on the buffer `a`;

if `jobvt = job::novec`, no columns of V^T/V^H (no left singular vectors) are computed.

`jobvt` and `jobu` cannot both be `job::overwritevec`.

m The number of rows in the matrix A ($0 \leq m$).

a The buffer `a`, size `(lda,*)`. The buffer `a` contains the matrix A. The second dimension of `a` must be at least $\max(1, m)$.

lda The leading dimension of `a`.

ldu The leading dimension of `u`.

ldvt The leading dimension of `vt`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *gesvd_scratchpad_size* function.

Output Parameters

a On exit,

If `jobu = job::overwritevec`, `a` is overwritten with the first $\min(m, n)$ columns of U (the left singular vectors stored columnwise);

If `jobvt = job::overwritevec`, `a` is overwritten with the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored rowwise);

If `jobu` $\neq$ `job::overwritevec` and `jobvt` $\neq$ `job::overwritevec`, the contents of a are destroyed.

s Buffer containing the singular values, size at least $\max(1, \min(m, n))$. Contains the singular values of A sorted so that $s(i) \geq s(i + 1)$.

u Buffer containing U; the second dimension of `u` must be at least $\max(1, m)$ if `jobu = job::allvec`, and at least $\max(1, \min(m, n))$ if `jobu = job::somevec`.

If `jobu = job::allvec`, `u` contains the $m \times m$ orthogonal/unitary matrix U.

If `jobu = job::somevec`, `u` contains the first $\min(m, n)$ columns of U (the left singular vectors stored column-wise).

If `jobu = job::novec` or `job::overwritevec`, `u` is not referenced.

vt Buffer containing V^T; the second dimension of `vt` must be at least $\max(1, n)$.

If `jobvt = job::allvec`, `vt` contains the $n \times n$ orthogonal/unitary matrix V^T/V^H.

If `jobvt = job::somevec`, `vt` contains the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored row-wise).

If `jobvt = job::novec` or `job::overwritevec`, `vt` is not referenced.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, then if `bdsqr` did not converge, i specifies how many superdiagonals of the intermediate bidiagonal form B did not converge to zero, and `scratchpad(2:min(m,n))` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in `s` (not necessarily sorted). B satisfies $A = UBV^T$, so it has the same singular values as A, and singular vectors related by U and V^T.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

gesvd (USM Version)

Description

The routine computes the singular value decomposition (SVD) of a real/complex $m \times n$ matrix A, optionally computing the left and/or right singular vectors. The SVD is written as

$A = U\Sigma V^T$ for real routines

$A = U\Sigma V^H$ for complex routines

where Σ is an $m \times n$ diagonal matrix, U is an $m \times m$ orthogonal/unitary matrix, and V is an $n \times n$ orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event gesvd(cl::sycl::queue &queue, onemkl::job jobu, onemkl::job jobvt,␣
↪std::int64_t m, std::int64_t n, T *a, std::int64_t lda, RealT *s, T *u, std::int64_
↪t ldu, T *vt, std::int64_t ldvt, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

jobu Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix U.

If `jobu = job::allvec`, all m columns of U are returned in the array `u`;

if `jobu = job::somevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the array `u`;

if `jobu = job::overwritevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the array a;

if `jobu = job::novec`, no columns of U (no left singular vectors) are computed.

jobvt Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix V^T/V^H.

If `jobvt = job::allvec`, all n columns of V^T/V^H are returned in the array `vt`;

if `jobvt = job::somevec`, the first $\min(m,n)$ columns of V^T/V^H (the left singular vectors) are returned in the array vt;

if `jobvt = job::overwritevec`, the first $\min(m,n)$ columns of V^T/V^H (the left singular vectors) are overwritten on the array `a`;

if `jobvt = job::novec`, no columns of V^T/V^H (no left singular vectors) are computed.

`jobvt` and `jobu` cannot both be `job::overwritevec`.

m The number of rows in the matrix A ($0 \leq m$).

a Pointer to array `a`, size `(lda,*)`, containing the matrix A. The second dimension of `a` must be at least $\max(1,m)$.

lda The leading dimension of `a`.

ldu The leading dimension of `u`.

ldvt The leading dimension of `vt`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *gesvd_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a On exit,

If `jobu = job::overwritevec`, `a` is overwritten with the first $\min(m,n)$ columns of U (the left singular vectors stored columnwise);

If `jobvt = job::overwritevec`, `a` is overwritten with the first $\min(m,n)$ rows of V^T/V^H (the right singular vectors stored rowwise);

If `jobu` $\neq$ `job::overwritevec` and `jobvt` $\neq$ `job::overwritevec`, the contents of a are destroyed.

s Array containing the singular values, size at least $\max(1,\min(m,n))$. Contains the singular values of A sorted so that $s(i) \geq s(i+1)$.

u Array containing U; the second dimension of `u` must be at least $\max(1,m)$ if `jobu = job::allvec`, and at least $\max(1,\min(m,n))$ if `jobu = job::somevec`.

If `jobu = job::allvec`, `u` contains the $m \times m$ orthogonal/unitary matrix U.

If `jobu = job::somevec`, `u` contains the first $\min(m,n)$ columns of U (the left singular vectors stored column-wise).

If `jobu = job::novec` or `job::overwritevec`, `u` is not referenced.

vt Array containing V^T; the second dimension of `vt` must be at least $\max(1,n)$.

If `jobvt = job::allvec`, `vt` contains the $n \times n$ orthogonal/unitary matrix V^T/V^H.

If `jobvt = job::somevec`, `vt` contains the first $\min(m,n)$ rows of V^T/V^H (the right singular vectors stored row-wise).

If `jobvt = job::novec` or `job::overwritevec`, `vt` is not referenced.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If `info=-i`, the i-th parameter had an illegal value.
>
> If `info=i`, then if `bdsqr` did not converge, i specifies how many superdiagonals of the intermediate bidiagonal form B did not converge to zero, and `scratchpad(2:min(m,n))` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in `s` (not necessarily sorted). B satisfies $A = UBV^T$, so it has the same singular values as A, and singular vectors related by U and V^T.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

gesvd_scratchpad_size

Computes size of scratchpad memory required for *gesvd* function.

Description

`gesvd_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *gesvd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

gesvd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t gesvd_scratchpad_size(cl::sycl::queue &queue, onemkl::job jobu,␣
↪onemkl::job jobvt, std::int64_t m, std::int64_t n, std::int64_t lda, std::int64_t␣
↪ldu, std::int64_t ldvt)
}
```

Input Parameters

queue Device queue where calculations by *gesvd* function will be performed.

jobu Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix U.

If `jobu = job::allvec`, all m columns of U are returned in the buffer `u`;

if `jobu = job::somevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the buffer `v`;

if `jobu = job::overwritevec`, the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the buffer `a`;

if `jobu = job::novec`, no columns of U (no left singular vectors) are computed.

jobvt Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix V^T/V^H.

If `jobvt = job::allvec`, all n columns of V^T/V^H are returned in the buffer `vt`;

if `jobvt = job::somevec`, the first $\min(m, n)$ columns of V^T/V^H (the left singular vectors) are returned in the buffer `vt`;

if `jobvt = job::overwritevec`, the first $\min(m, n)$ columns of V^T/V^H (the left singular vectors) are overwritten on the buffer `a`;

if `jobvt = job::novec`, no columns of V^T/V^H (no left singular vectors) are computed.

m The number of rows in the matrix A ($0 \leq m$).

n The number of columns in the matrix A ($0 \leq n$).

lda The leading dimension of `a`.

ldu The leading dimension of `u`.

ldvt The leading dimension of `vt`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *gesvd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

heevd

Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

Description

`heevd` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A. In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$Az_i = \lambda_i z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

heevd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void heevd(cl::sycl::queue &queue, onemkl::job jobz, onemkl::uplo upper_lower,␣
↪std::int64_t n, butter<T,1> &a, std::int64_t lda, cl::sycl::buffer<realT,1> &w,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n The order of the matrix A ($0 \leq n$).

a The buffer `a`, size (`lda,*`). The buffer `a` contains the matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`. Must be at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *heevd_scratchpad_size* function.

Output Parameters

a If `jobz = job::vec`, then on exit this buffer is overwritten by the unitary matrix Z which contains the eigenvectors of A.

w Buffer, size at least n. Contains the eigenvalues of the matrix A in ascending order.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info=i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

heevd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event heevd(cl::sycl::queue &queue, onemkl::job jobz, onemkl::uplo upper_
↪lower, std::int64_t n, butter<T,1> &a, std::int64_t lda, RealT *w, T *scratchpad,␣
↪std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events␣
↪= {})
}
```

Input Parameters

queue The queue where the routine should be executed.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n The order of the matrix A ($0 \leq n$).

a Pointer to array containing A, size (`lda, *`).The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`. Must be at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *heevd_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a If `jobz = job::vec`, then on exit this array is overwritten by the unitary matrix Z which contains the eigenvectors of A.

w Pointer to array of size at least n. Contains the eigenvalues of the matrix A in ascending order.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info=i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

heevd_scratchpad_size

Computes size of scratchpad memory required for *heevd* function.

Description

`heevd_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *heevd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

heevd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t heevd_scratchpad_size(cl::sycl::queue &queue, onemkl::job jobz,␣
↪onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *heevd* function will be performed.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n The order of the matrix A ($0 \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *heevd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

hegvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

Description

`hegvd` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be Hermitian and B is also positive definite.

It uses a divide and conquer algorithm.

hegvd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void hegvd(cl::sycl::queue &queue, std::int64_t itype, onemkl::job jobz,␣
↪onemkl::uplo upper_lower, std::int64_t n, cl::sycl::buffer<T,1> &a, std::int64_t␣
↪lda, cl::sycl::buffer<T,1> &b, std::int64_t ldb, cl::sycl::buffer<realT,1> &w,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

itype Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = uplo::lower`, `a` and `b` stores the lower triangular part of A and B.

n The order of the matrices A and B ($0 \leq n$).

a Buffer, size `a(lda,*)` contains the upper or lower triangle of the Hermitian matrix A, as specified by upper_lower.

The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; at least $\max(1, n)$.

b Buffer, size `b(ldb,*)` contains the upper or lower triangle of the Hermitian matrix B, as specified by upper_lower.

The second dimension of `b` must be at least $\max(1, n)$.

ldb The leading dimension of `b`; at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hegvd_scratchpad_size* function.

Output Parameters

a On exit, if `jobz = job::vec`, then if info $= 0$, `a` contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if itype $= 1$ or itype $= 2$, $Z^H B Z = I$;

if itype $= 3$, $Z^H B^{-1} Z = I$;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

b On exit, if info $\leq n$, the part of `b` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H U$ or $B = L L^H$.

w Buffer, size at least n. If info $= 0$, contains the eigenvalues of the matrix A in ascending order.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

For info $\leq n$:

If info $= i$, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero;

If info $= i$, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

For info $> n$:

If info $= n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

hegvd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event hegvd(cl::sycl::queue &queue, std::int64_t itype, onemkl::job jobz,␣
↪onemkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda, T *b, std::int64_
↪t ldb, RealT *w, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

itype Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = uplo::lower`, `a` and `b` stores the lower triangular part of A and B.

n The order of the matrices A and B ($0 \leq n$).

a Pointer to array of size `a(lda,*)` containing the upper or lower triangle of the Hermitian matrix A, as specified by upper_lower. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; at least $\max(1, n)$.

b Pointer to array of size `b(ldb,*)` containing the upper or lower triangle of the Hermitian matrix B, as specified by upper_lower. The second dimension of `b` must be at least $\max(1, n)$.

ldb The leading dimension of `b`; at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hegvd_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a On exit, if `jobz = job::vec`, then if info $= 0$, `a` contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if itype $= 1$ or itype $= 2$, $Z^H B Z = I$;

if itype $= 3$, $Z^H B^{-1} Z = I$;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

b On exit, if info $\leq n$, the part of `b` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H U$ or $B = L L^H$.

w Pointer to array of size at least n. If info $= 0$, contains the eigenvalues of the matrix A in ascending order.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

For info $\leq n$:

If info $= i$, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero;

If info $= i$, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

For info $> n$:

If info $= n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

hegvd_scratchpad_size

Computes size of scratchpad memory required for *hegvd* function.

Description

`hegvd_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *hegvd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

hegvd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t hegvd_scratchpad_size(cl::sycl::queue &queue, std::int64_t itype,␣
↪onemkl::job jobz, onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda,␣
↪std::int64_t ldb)
}
```

Input Parameters

queue Device queue where calculations by *hegvd* function will be performed.

itype Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = uplo::lower`, `a` and `b` store the lower triangular part of A and B.

n The order of the matrices A and B ($0 \leq n$).

lda The leading dimension of `a`. Currently `lda` is not referenced in this function.

ldb The leading dimension of `b`. Currently `ldb` is not referenced in this function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *hegvd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

hetrd

Reduces a complex Hermitian matrix to tridiagonal form.

Description

`hetrd` supports the following precisions.

Routine name	T
`chetrd`	`std::complex<float>`
`zhetrd`	`std::complex<double>`

The routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = QTQ^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided to work with Q in this representation.

hetrd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void hetrd(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<realT,1> &d,␣
↪cl::sycl::buffer<realT,1> &e, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n The order of the matrices A ($0 \leq n$).

a Buffer, size `(lda,*)`. The buffer `a` contains either the upper or lower triangle of the Hermitian matrix A, as specified by upper_lower.

The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; at least $\max(1, n)$

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hetrd_scratchpad_size* function.

Output Parameters

a On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the buffer `tau`, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the buffer `tau`, represent the orthogonal matrix Q as a product of elementary reflectors.

d Buffer containing the diagonal elements of the matrix T. The dimension of `d` must be at least $\max(1, n)$.

e Buffer containing the off diagonal elements of the matrix T. The dimension of `e` must be at least $\max(1, n-1)$.

tau Buffer, size at least $\max(1, n-1)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n-1$ elementary reflectors.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

hetrd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event hetrd(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_
↪t n, T *a, std::int64_t lda, RealT *d, RealT *e, T *tau, T *scratchpad, std::int64_
↪t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n The order of the matrices A ($0 \leq n$).

a The pointer to matrix A, size `(lda,*)`. Contains either the upper or lower triangle of the Hermitian matrix A, as specified by `upper_lower`. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; at least $\max(1, n)$

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *hetrd_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors.

d Pointer to diagonal elements of the matrix T. The dimension of `d` must be at least $\max(1, n)$.

e Pointer to off diagonal elements of the matrix T. The dimension of `e` must be at least $\max(1, n-1)$.

tau Pointer to array of size at least $\max(1, n-1)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n-1$ elementary reflectors.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

hetrd_scratchpad_size

Computes size of scratchpad memory required for *hetrd* function.

Description

`hetrd_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *hetrd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

hetrd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t hetrd_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
↪ std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *hetrd* function will be performed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A and B.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n The order of the matrices A and B ($0 \leq n$).

lda The leading dimension of `a`. Currently, `lda` is not referenced in this function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *hetrd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

orgbr

Generates the real orthogonal matrix Q or P^T determined by *gebrd*.

`orgbr` supports the following precisions.

T
float
double

Description

The routine generates the whole or part of the orthogonal matrices Q and P^T formed by the routines *gebrd*. All valid combinations of arguments are described in *Input parameters*. In most cases you need the following:

To compute the whole $m \times m$ matrix Q:

```
orgbr(queue, generate::q, m, m, n, a, ...)
```

(note that the array `a` must have at least m columns).

To form the n leading columns of Q if $m > n$:

```
orgbr(queue, generate::q, m, n, n, a, ...)
```

To compute the whole $n \times n$ matrix P^T:

```
orgbr(queue, generate::p, n, n, m, a, ...)
```

(note that the array `a` must have at least n rows).

To form the m leading rows of P^T if $m < n$:

```
orgbr(queue, generate::p, m, n, m, a, ...)
```

orgbr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void orgbr(cl::sycl::queue &queue, onemkl::generate gen, std::int64_t m, std::int64_
↪t n, std::int64_t k, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,
↪1> &tau, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

gen Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix Q.

If `gen = generate::p`, the routine generates the matrix P^T.

m The number of rows in the matrix Q or P^T to be returned $(0 \leq m)$.

If `gen = generate::q`, $m \leq n \leq \min(m, k)$.

If `gen = generate::p`, $n \leq m \leq \min(n, k)$.

n The number of rows in the matrix Q or P^T to be returned $(0 \leq n)$. See m for constraints.

k If `gen = generate::q`, the number of columns in the original $m \times k$ matrix reduced by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix reduced by *gebrd*.

a The buffer `a` as returned by *gebrd*.

lda The leading dimension of `a`.

tau Buffer, size $\min(m, k)$ if `gen = generate::q`, size $\min(n, k)$ if `gen = generate::p`. Scalar factor of the elementary reflectors, as returned by *gebrd* in the array tauq or taup.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgbr_scratchpad_size* function.

Output Parameters

a Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

orgbr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgbr(cl::sycl::queue &queue, onemkl::generate gen, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad,␣
↪std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events␣
↪= {})
}
```

Input Parameters

queue The queue where the routine should be executed.

gen Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix Q.

If `gen = generate::p`, the routine generates the matrix P^T.

m The number of rows in the matrix Q or P^T to be returned $(0 \leq m)$.

If `gen = generate::q`, $m \leq n \leq \min(m, k)$.

If `gen = generate::p`, $n \leq m \leq \min(n, k)$.

n The number of rows in the matrix Q or P^T to be returned $(0 \leq n)$. See m for constraints.

k If `gen = generate::q`, the number of columns in the original $m \times k$ matrix reduced by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix reduced by *gebrd*.

a Pointer to array `a` as returned by *gebrd*.

lda The leading dimension of `a`.

tau Pointer to array of size $\min(m, k)$ if `gen = generate::q`, size $\min(n, k)$ if `gen = generate::p`. Scalar factor of the elementary reflectors, as returned by *gebrd* in the array tauq or taup.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgbr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by n leading columns of the $m \times m$ orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

orgbr_scratchpad_size

Computes size of scratchpad memory required for *orgbr* function.

`orgbr_scratchpad_size` supports the following precisions.

T
`float`
`double`

Description

Computes the number of elements of type `T` the scratchpad memory to be passed to *orgbr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

orgbr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgbr_scratchpad_size(cl::sycl::queue &queue, onemkl::generate gen,␣
↪std::int64_t m, std::int64_t n, std::int64_t k, std::int64_t lda, std::int64_t &
↪scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations by *orgbr* function will be performed.

gen Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix Q.

If `gen = generate::p`, the routine generates the matrix P^T.

m The number of rows in the matrix Q or P^T to be returned ($0 \leq m$).

If `gen = generate::q`, $m \leq n \leq \min(m, k)$.

If `gen = generate::p`, $n \leq m \leq \min(n, k)$.

n The number of rows in the matrix Q or P^T to be returned ($0 \leq n$). See `m` for constraints.

k If `gen = generate::q`, the number of columns in the original $m \times k$ matrix returned by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix returned by *gebrd*.

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *orgbr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

orgtr

Generates the real orthogonal matrix Q determined by *sytrd*.

Description

`orgtr` supports the following precisions.

T
`float`
`double`

The routine explicitly generates the $n \times n$ orthogonal matrix Q formed by *sytrd* when reducing a real symmetric matrix A to tridiagonal form: $A = QTQ^T$. Use this routine after a call to *sytrd*.

orgtr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void orgtr(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

n The order of the matrix Q ($0 \leq n$).

a The buffer `a` as returned by *sytrd*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a` ($n \leq$ lda).

tau The buffer `tau` as returned by *sytrd*. The dimension of `tau` must be at least $\max(1, n - 1)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgtr_scratchpad_size* function.

Output Parameters

a Overwritten by the orthogonal matrix Q.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

orgtr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgtr(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_
↪t n, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size,␣
↪const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

n The order of the matrix Q ($0 \leq n$).

a The pointer to `a` as returned by *sytrd*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a` ($n \leq$ lda).

tau The pointer to `tau` as returned by *sytrd*. The dimension of `tau` must be at least $\max(1, n-1)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *orgtr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by the orthogonal matrix Q.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

orgtr_scratchpad_size

Computes size of scratchpad memory required for *orgtr* function.

Description

`orgtr_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *orgtr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

orgtr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgtr_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
↪ std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *orgtr* function will be performed.

upper_lower Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

n The order of the matrix Q ($0 \leq n$).

lda The leading dimension of `a` ($n \leq$ lda).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *orgtr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ormtr

Multiplies a real matrix by the real orthogonal matrix Q determined by *sytrd*.

Description

`ormtr` supports the following precisions.

T
`float`
`double`

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q formed by:ref:*onemkl_lapack_sytrd* when reducing a real symmetric matrix A to tridiagonal form: $A = QTQ^T$. Use this routine after a call to *sytrd*.

Depending on the parameters left_right and trans, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result on C).

ormtr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ormtr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_
↪lower, onemkl::transpose trans, std::int64_t m, std::int64_t n, cl::sycl::buffer<T,
↪1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c,␣
↪std::int64_t ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

$r = m$	if `left_right = side::left`
$r = n$	if `left_right = side::right`

queue The queue where the routine should be executed.

left_right Must be either `side::left` or `side::right`.

If `left_right = side::left`, Q or Q^T is applied to C from the left.

If `left_right = side::right`, Q or Q^T is applied to C from the right.

upper_lower Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

trans Must be either `transpose::nontrans` or `transpose::trans`.

If `trans = transpose::nontrans`, the routine multiplies C by Q.

If `trans = transpose::trans`, the routine multiplies C by Q^T.

m The number of rows in the matrix C ($m \geq 0$).

n The number of columns in the matrix C ($n \geq 0$).

a The buffer `a` as returned by *sytrd*.

lda The leading dimension of `a` ($\max(1, r) \leq$ lda).

tau The buffer `tau` as returned bya *sytrd*. The dimension of `tau` must be at least $\max(1, r-1)$.

c The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c` ($\max(1, n) \leq$ ldc).

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ormtr_scratchpad_size* function.

Output Parameters

c Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `left_right` and `trans`).

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ormtr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ormtr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::uplo␣
↪upper_lower, onemkl::transpose trans, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

$r = m$	if `left_right = side::left`
$r = n$	if `left_right = side::right`

queue The queue where the routine should be executed.

left_right Must be either `side::left` or `side::right`.

If `left_right = side::left`, Q or Q^T is applied to C from the left.

If `left_right = side::right`, Q or Q^T is applied to C from the right.

upper_lower Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

trans Must be either `transpose::nontrans` or `transpose::trans`.

If `trans = transpose::nontrans`, the routine multiplies C by Q.

If `trans = transpose::trans`, the routine multiplies C by Q^T.

m The number of rows in the matrix C ($m \geq 0$).

n The number of columns in the matrix C ($n \geq 0$).

a The pointer to `a` as returned by *sytrd*.

lda The leading dimension of `a` ($\max(1, r) \leq$ lda).

tau The buffer `tau` as returned by *sytrd*. The dimension of `tau` must be at least $\max(1, r - 1)$.

c The pointer to memory containing the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c` ($\max(1, n) \leq$ ldc).

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ormtr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by `left_right` and `trans`).

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ormtr_scratchpad_size

Computes size of scratchpad memory required for *ormtr* function.

Description

`ormtr_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *ormtr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ormtr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ormtr_scratchpad_size(cl::sycl::queue &queue, onemkl::side left_right,␣
↪onemkl::uplo upper_lower, onemkl::transpose trans, std::int64_t m, std::int64_t n,␣
↪std::int64_t lda, std::int64_t ldc)
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

$r = m$	if `left_right = side::left`
$r = n$	if `left_right = side::right`

queue Device queue where calculations by *ormtr* function will be performed.

left_right Must be either `side::left` or `side::right`.

If `left_right = side::left`, Q or Q^T is applied to C from the left.

If `left_right = side::right`, Q or Q^T is applied to C from the right.

upper_lower Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *sytrd*.

trans Must be either `transpose::nontrans` or `transpose::trans`.

If `trans = transpose::nontrans`, the routine multiplies C by Q.

If `trans = transpose::trans`, the routine multiplies C by Q^T.

m The number of rows in the matrix C ($m \geq 0$).

n The number of rows in the matrix C ($n \geq 0$).

lda The leading dimension of `a` ($\max(1, r) \leq$ lda).

ldc The leading dimension of `c` ($\max(1, n) \leq$ ldc).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ormtr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

syevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

Description

`syevd` supports the following precisions.

T
`float`
`double`

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A. In other words, it can compute the spectral factorization of A as: $A = Z\lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$Az_i = \lambda_i z_i$ for $i = 1, 2, ..., n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

syevd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void syevd(cl::sycl::queue &queue, jobz jobz, onemkl::uplo upper_lower, std::int64_
↪t n, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &w,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n The order of the matrix A ($0 \leq n$).

a The buffer `a`, size (`lda, *`). The buffer `a` contains the matrix A. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`. Must be at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *syevd_scratchpad_size* function.

Output Parameters

a If `jobz = job::vec`, then on exit this buffer is overwritten by the orthogonal matrix Z which contains the eigenvectors of A.

w Buffer, size at least n. Contains the eigenvalues of the matrix A in ascending order.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If info $= i$, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

syevd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event syevd(cl::sycl::queue &queue, jobz jobz, onemkl::uplo upper_lower,␣
↪std::int64_t n, T *a, std::int64_t lda, T *w, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n The order of the matrix A ($0 \leq n$).

a Pointer to array containing A, size (`lda,` $\star$). The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`. Must be at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *syevd_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a If `jobz = job::vec`, then on exit this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of A.

w Pointer to array of size at least n. Contains the eigenvalues of the matrix A in ascending order.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If info $= i$, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If info $= i$, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

syevd_scratchpad_size

Computes size of scratchpad memory required for *syevd* function.

Description

`syevd_scratchpad_size` supports the following precisions.

T
float
double

Computes the number of elements of type `T` the scratchpad memory to be passed to *syevd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

syevd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t syevd_scratchpad_size(cl::sycl::queue &queue, onemkl::job jobz,␣
↪onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *syevd* function will be performed.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

n The order of the matrix A ($0 \leq n$).

lda The leading dimension of `a`. Currently `lda` is not referenced in this function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *syevd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

Description

`sygvd` supports the following precisions.

T
float
double

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be symmetric and B is also positive definite.

It uses a divide and conquer algorithm.

sygvd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void sygvd(cl::sycl::queue &queue, std::int64_t itype, onemkl::job jobz,␣
↪onemkl::uplo upper_lower, std::int64_t n, cl::sycl::buffer<T,1> &a, std::int64_t␣
↪lda, cl::sycl::buffer<T,1> &b, std::int64_t ldb, cl::sycl::buffer<T,1> &w,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

itype Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = job::lower`, `a` and `b` stores the lower triangular part of A and B.

n The order of the matrices A and B ($0 \leq n$).

a Buffer, size `a(lda,*)` contains the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; at least $\max(1, n)$.

b Buffer, size `b (ldb,*)` contains the upper or lower triangle of the symmetric matrix B, as specified by `upper_lower`. The second dimension of `b` must be at least $\max(1, n)$.

ldb The leading dimension of `b`; at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sygvd_scratchpad_size* function.

Output Parameters

a On exit, if `jobz = job::vec`, then if info $= 0$, `a` contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if itype $= 1$ or 2 , $Z^T BZ = I$;

if itype $= 3$, $Z^T B^{-1} Z = I$;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

b On exit, if info $\leq n$, the part of `b` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^T U$ or $B = LL^T$.

w Buffer, size at least n. If info $= 0$, contains the eigenvalues of the matrix A in ascending order.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

For info $\leq n$:

If info $= i$, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If info $= i$, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

For info $> n$:

If info $= n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

sygvd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event sygvd(cl::sycl::queue &queue, std::int64_t itype, onemkl::job jobz,␣
↪onemkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda, T *b, std::int64_
↪t ldb, T *w, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_
↪class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

itype Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = job::lower`, `a` and `b` stores the lower triangular part of A and B.

n The order of the matrices A and B ($0 \leq n$).

a Pointer to array of size a `(lda,*)` containing the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; at least $\max(1, n)$.

b Pointer to array of size `b (ldb,*)` contains the upper or lower triangle of the symmetric matrix B, as specified by `upper_lower`. The second dimension of `b` must be at least $\max(1, n)$.

ldb The leading dimension of `b`; at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sygvd_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a On exit, if `jobz = job::vec`, then if info $= 0$, a contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if itype $= 1$ or 2, $Z^T BZ = I$;

if itype $= 3$, $Z^T B^{-1} Z = I$;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

b On exit, if info $\leq n$, the part of `b` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^T U$ or $B = LL^T$.

w Pointer to array of size at least `n`. If info $= 0$, contains the eigenvalues of the matrix A in ascending order.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

For info $\leq n$:

If info $= i$, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If info $= i$, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns info$/(n+1)$ through mod(info, $n+1$).

For info $> n$:

If info $= n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

sygvd_scratchpad_size

Computes size of scratchpad memory required for *sygvd* function.

Description

sygvd_scratchpad_size ` supports the following precisions.

T
float
double

Computes the number of elements of type `T` the scratchpad memory to be passed to *sygvd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

sygvd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t sygvd_scratchpad_size(cl::sycl::queue &queue, std::int64_t itype,␣
↪onemkl::job jobz, onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda,␣
↪std::int64_t ldb)
}
```

Input Parameters

queue Device queue where calculations by *sygvd* function will be performed.

itype Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype $= 1$, the problem type is $Ax = \lambda Bx$;

if itype $= 2$, the problem type is $ABx = \lambda x$;

if itype $= 3$, the problem type is $BAx = \lambda x$.

jobz Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, `a` and `b` store the upper triangular part of A and B.

If `upper_lower = job::lower`, `a` and `b` stores the lower triangular part of A and B.

n The order of the matrices A and B ($0 \leq n$).

lda The leading dimension of `a`.

ldb The leading dimension of `b`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *sygvd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

sytrd

Reduces a real symmetric matrix to tridiagonal form.

Description

`sytrd` supports the following precisions.

T
float
double

The routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = QTQ^T$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n - 1$ elementary reflectors. Routines are provided for working with Q in this representation .

sytrd (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void sytrd(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &d,␣
↪cl::sycl::buffer<T,1> &e, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n The order of the matrices A ($0 \leq n$).

a The buffer `a`, size `(lda,*)`. Contains the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`.

The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sytrd_scratchpad_size* function.

Output Parameters

a On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the buffer `tau`, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the buffer `tau`, represent the orthogonal matrix Q as a product of elementary reflectors.

d Buffer containing the diagonal elements of the matrix T. The dimension of `d` must be at least $\max(1, n)$.

e Buffer containing the off diagonal elements of the matrix T. The dimension of `e` must be at least $\max(1, n - 1)$.

tau Buffer, size at least $\max(1, n)$. Stores $(n - 1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n - 1$ elementary reflectors. $\tau(n)$ is used as workspace.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If info $= -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

sytrd (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event sytrd(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_
↪t n, T *a, std::int64_t lda, T *d, T *e, T *tau, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` stores the upper triangular part of A.

If `upper_lower = uplo::lower`, `a` stores the lower triangular part of A.

n The order of the matrices A ($0 \leq n$).

a The pointer to matrix A, size `(lda,*)`. Contains the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a`; at least $\max(1, n)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *sytrd_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors.

d Pointer to diagonal elements of the matrix T. The dimension of `d` must be at least $\max(1, n)$.

e Pointer to off diagonal elements of the matrix T. The dimension of `e` must be at least $\max(1, n-1)$.

tau Pointer to array of size at least $\max(1, n)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n-1$ elementary reflectors. $\tau(n)$ is used as workspace.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

sytrd_scratchpad_size

Computes size of scratchpad memory required for *sytrd* function.

Description

`sytrd_scratchpad_size` supports the following precisions.

T
`float`
`double`

Computes the number of elements of type `T` the scratchpad memory to be passed to *sytrd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

sytrd_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t sytrd_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
↪ std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *sytrd* function will be performed.

upper_lower Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, a stores the upper triangular part of A.

If `upper_lower = uplo::lower`, a stores the lower triangular part of A.

n The order of the matrices A ($0 \leq n$).

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *sytrd* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ungbr

Generates the complex unitary matrix Q or P^t determined by *gebrd*.

Description

`ungbr` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine generates the whole or part of the unitary matrices Q and P^H formed by the routines *gebrd*. All valid combinations of arguments are described in *Input Parameters*; in most cases you need the following:

To compute the whole $m \times m$ matrix Q, use:

```
oneapi::mkl::lapack::ungbr(queue, generate::q, m, m, n, a, ...)
```

(note that the buffer `a` must have at least m columns).

To form the n leading columns of Q if $m > n$, use:

```
oneapi::mkl::lapack::ungbr(queue, generate::q, m, n, n, a, ...)
```

To compute the whole $n \times n$ matrix P^T, use:

```
oneapi::mkl::lapack::ungbr(queue, generate::p, n, n, m, a, ...)
```

(note that the array `a` must have at least n rows).

To form the m leading rows of P^T if $m < n$, use:

```
oneapi::mkl::lapack::ungbr(queue, generate::p, m, n, m, a, ...)
```

ungbr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ungbr(cl::sycl::queue &queue, onemkl::generate gen, std::int64_t m, std::int64_
↪t n, std::int64_t k, cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,
↪1> &tau, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

gen Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix Q.

If `gen = generate::p`, the routine generates the matrix P^T.

m The number of rows in the matrix Q or P^T to be returned ($0 \leq m$).

If `gen = generate::q`, $m \geq n \geq \min(m, k)$.

If `gen = generate::p`, $n \geq m \geq \min(n, k)$.

n The number of columns in the matrix Q or P^T to be returned ($0 \leq n$). See `m` for constraints.

k If `gen = generate::q`, the number of columns in the original $m \times k$ matrix returned by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix returned by *gebrd*.

a The buffer `a` as returned by *gebrd*.

lda The leading dimension of `a`.

tau For `gen = generate::q`, the array `tauq` as returned by *gebrd*. For `gen = generate::p`, the array `taup` as returned by *gebrd*.

The dimension of `tau` must be at least $\max(1, \min(m, k))$ for `gen = generate::q`, or $\max(1, \min(m, k))$ for `gen = generate::p`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by *ungbr_scratchpad_size* function.

Output Parameters

a Overwritten by n leading columns of the $m \times m$ unitary matrix Q or P^T, (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If $info = -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ungbr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungbr(cl::sycl::queue &queue, onemkl::generate gen, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad,␣
↪std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events␣
↪= {})
}
```

Input Parameters

queue The queue where the routine should be executed.

gen Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix Q.

If `gen = generate::p`, the routine generates the matrix P^T.

m The number of rows in the matrix Q or P^T to be returned ($0 \geq m$).

If `gen = generate::q`, $m \geq n \geq \min(m, k)$.

If `gen = generate::p`, $n \geq m \geq \min(n, k)$.

n The number of columns in the matrix Q or P^T to be returned ($0 \leq n$). See `m` for constraints.

k If `gen = generate::q`, the number of columns in the original $m \times k$ matrix returned by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix returned by *gebrd*.

a The pointer to `a` as returned by *gebrd*.

lda The leading dimension of `a`.

tau For `gen = generate::q`, the array `tauq` as returned by *gebrd*. For `gen = generate::p`, the array `taup` as returned by *gebrd*.

The dimension of `tau` must be at least $\max(1, \min(m, k))$ for `gen = generate::q`, or $\max(1, \min(m, k))$ for `gen = generate::p`.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by *ungbr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by n leading columns of the $m \times m$ unitary matrix Q or P^T, (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If $info = -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ungbr_scratchpad_size

Computes size of scratchpad memory required for *ungbr* function.

Description

`ungbr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type T the scratchpad memory to be passed to *ungbr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ungbr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungbr_scratchpad_size(cl::sycl::queue &queue, onemkl::generate gen,␣
↪std::int64_t m, std::int64_t n, std::int64_t k, std::int64_t lda, std::int64_t &
↪scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations by *ungbr* function will be performed.

gen Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix Q.

If `gen = generate::p`, the routine generates the matrix P^T.

m The number of rows in the matrix Q or P^T to be returned $(0 \leq m)$.

If `gen = generate::q`, $m \geq n \geq \min(m, k)$.

If `gen = generate::p`, $n \geq m \geq \min(n, k)$.

n The number of columns in the matrix Q or P^T to be returned $(0 \leq n)$. See m for constraints.

k If `gen = generate::q`, the number of columns in the original $m \times k$ matrix reduced by *gebrd*.

If `gen = generate::p`, the number of rows in the original $k \times n$ matrix reduced by *gebrd*.

lda The leading dimension of `a`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *ungbr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ungtr

Generates the complex unitary matrix Q determined by *hetrd*.

Description

`ungtr` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine explicitly generates the $n \times n$ unitary matrix Q formed by *hetrd* when reducing a complex Hermitian matrix A to tridiagonal form: $A = QTQ^H$. Use this routine after a call to *hetrd*.

ungtr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void ungtr(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,␣
↪cl::sycl::buffer<T,1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau,␣
↪cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

n The order of the matrix Q ($0 \leq n$).

a The buffer `a` as returned by *hetrd*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a` ($n \leq$ lda).

tau The buffer `tau` as returned by *hetrd*. The dimension of `tau` must be at least $\max(1, n-1)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ungtr_scratchpad_size* function.

Output Parameters

a Overwritten by the unitary matrix Q.

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

ungtr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungtr(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_
↪t n, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size,␣
↪const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue The queue where the routine should be executed.

upper_lower Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

n The order of the matrix Q ($0 \leq n$).

a The pointer to `a` as returned by *hetrd*. The second dimension of `a` must be at least $\max(1, n)$.

lda The leading dimension of `a` ($n \leq$ lda).

tau The pointer to `tau` as returned by *hetrd*. The dimension of `tau` must be at least $\max(1, n-1)$.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *ungtr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Overwritten by the unitary matrix Q.

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

> Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:
>
> If info $= -i$, the i-th parameter had an illegal value.
>
> If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

ungtr_scratchpad_size

Computes size of scratchpad memory required for *ungtr* function.

Description

`ungtr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *ungtr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

ungtr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungtr_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
↪ std::int64_t n, std::int64_t lda)
}
```

Input Parameters

queue Device queue where calculations by *ungtr* function will be performed.

upper_lower Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

n The order of the matrix Q $(0 \leq n)$.

lda The leading dimension of `a` $(n \leq lda)$.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type T the scratchpad memory to be passed to *ungtr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

unmtr

Multiplies a complex matrix by the complex unitary matrix Q determined by *hetrd*.

Description

`unmtr` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

The routine multiplies a complex matrix C by Q or Q^H, where Q is the unitary matrix Q formed by *hetrd* when reducing a complex Hermitian matrix A to tridiagonal form: $A = QTQ^H$. Use this routine after a call to *hetrd*.

Depending on the parameters `left_right` and `trans`, the routine can form one of the matrix products QC, Q^HC, CQ, or CQ^H (overwriting the result on C).

unmtr (Buffer Version)

Syntax

```
namespace oneapi::mkl::lapack {
  void unmtr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_
↪lower, onemkl::transpose trans, std::int64_t m, std::int64_t n, cl::sycl::buffer<T,
↪1> &a, std::int64_t lda, cl::sycl::buffer<T,1> &tau, cl::sycl::buffer<T,1> &c,␣
↪std::int64_t ldc, cl::sycl::buffer<T,1> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

r=m	if `left_right = side::left`
r=n	if `left_right = side::right`

queue The queue where the routine should be executed.

left_right Must be either `side::left` or `side::right`.

If `left_right=side::left`, Q or Q^H is applied to C from the left.

If `left_right=side::right`, Q or Q^H is applied to C from the right.

upper_lower Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

trans Must be either `transpose::nontrans` or `transpose::conjtrans`.

If `trans=transpose::nontrans`, the routine multiplies C by Q.

If `trans=transpose::conjtrans`, the routine multiplies C by Q^H.

m The number of rows in the matrix C ($m \geq 0$).

n The number of columns the matrix C ($n \geq 0$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The buffer `a` as returned by *hetrd*.

lda The leading dimension of `a` ($\max(1, r) \leq$ lda).

tau The buffer `tau` as returned by *hetrd*. The dimension of `tau` must be at least $\max(1, r-1)$.

c The buffer `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c` ($\max(1, n) \leq$ ldc).

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *unmtr_scratchpad_size* function.

Output Parameters

c Overwritten by the product QC, $Q^H C$, CQ, or CQ^H (as specified by `left_right` and `trans`).

scratchpad Buffer holding scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If $info = -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

unmtr (USM Version)

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event unmtr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::uplo␣
↪upper_lower, onemkl::transpose trans, std::int64_t m, std::int64_t n, T *a,␣
↪std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

In the descriptions below, `r` denotes the order of Q:

r=m	if `left_right = side::left`
r=n	if `left_right = side::right`

queue The queue where the routine should be executed.

left_right Must be either `side::left` or `side::right`.

If `left_right=side::left`, Q or Q^H is applied to C from the left.

If `left_right=side::right`, Q or Q^H is applied to C from the right.

upper_lower Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

trans Must be either `transpose::nontrans` or `transpose::conjtrans`.

If `trans=transpose::nontrans`, the routine multiplies C by Q.

If `trans=transpose::conjtrans`, the routine multiplies C by Q^H.

m The number of rows in the matrix C ($m \geq 0$).

n The number of columns the matrix C ($n \geq 0$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a The pointer to `a` as returned by *hetrd*.

lda The leading dimension of `a` ($\max(1, r) \leq$ lda).

tau The pointer to `tau` as returned by *hetrd*. The dimension of `tau` must be at least $\max(1, r-1)$.

c The array `c` contains the matrix C. The second dimension of `c` must be at least $\max(1, n)$.

ldc The leading dimension of `c` ($\max(1, n) \leq$ ldc).

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by *unmtr_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

c Overwritten by the product QC, Q^HC, CQ, or CQ^H (as specified by `left_right` and trans).

scratchpad Pointer to scratchpad memory to be used by routine for storing intermediate results.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::host_bad_alloc

oneapi::mkl::device_bad_alloc

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

oneapi::mkl::lapack::computation_error

Exception is thrown in case of problems during calculations. The `info` code of the problem can be obtained by *info()* method of exception object:

If $info = -i$, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by *detail()* method of exception object.

Return Values

Output event to wait on to ensure computation is complete.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

unmtr_scratchpad_size

Computes size of scratchpad memory required for *unmtr* function.

Description

`unmtr_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Computes the number of elements of type `T` the scratchpad memory to be passed to *unmtr* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

unmtr_scratchpad_size

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t unmtr_scratchpad_size(cl::sycl::queue &queue, onemkl::side left_right,␣
↪onemkl::uplo upper_lower, onemkl::transpose trans, std::int64_t m, std::int64_t n,␣
↪std::int64_t lda, std::int64_t ldc)
}
```

Input Parameters

queue Device queue where calculations by *unmtr* function will be performed.

left_right Must be either `side::left` or `side::right`.

If `left_right=side::left`, Q or Q^H is applied to C from the left.

If `left_right=side::right`, Q or Q^H is applied to C from the right.

upper_lower Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to *hetrd*.

trans Must be either `transpose::nontrans` or `transpose::conjtrans`.

If `trans=transpose::nontrans`, the routine multiplies C by Q.

If `trans=transpose::conjtrans`, the routine multiplies C by Q^H.

m The number of rows in the matrix C ($m \geq 0$).

n The number of columns the matrix C ($n \geq 0$).

k The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

lda The leading dimension of a ($\max(1, r) \leq$ lda).

ldc The leading dimension of c ($\max(1, n) \leq$ ldc).

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Return Value

The number of elements of type `T` the scratchpad memory to be passed to *unmtr* function should be able to hold.

Parent topic: *LAPACK Singular Value and Eigenvalue Problem Routines*

LAPACK-like Extensions Routines

oneAPI Math Kernel Library DPC++ provides additional routines to extend the functionality of the LAPACK routines. These include routines to compute many independent factorizations, linear equation solutions, and similar. The following table lists the LAPACK-like Extensions routine groups.

Routines	Scratchpad Size Routines	Description
geqrf_batch	*geqrf_batch_scratchpad_size*	Computes the QR factorizations of a batch of general matrices.
getrf_batch	*getrf_batch_scratchpad_size*	Computes the LU factorizations of a batch of general matrices.
getri_batch	*getri_batch_scratchpad_size*	Computes the inverses of a batch of LU-factored general matrices.
getrs_batch	*getrs_batch_scratchpad_size*	Solves systems of linear equations with a batch of LU-factored square coefficient matrices, with multiple right-hand sides.
orgqr_batch	*orgqr_batch_scratchpad_size*	Generates the real orthogonal/complex unitary matrix Q_i of the QR factorization formed by geqrf_batch.
potrf_batch	*potrf_batch_scratchpad_size*	Computes the Cholesky factorization of a batch of symmetric (Hermitian) positive-definite matrices.
potrs_batch	*potrs_batch_scratchpad_size*	Solves systems of linear equations with a batch of Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrices, with multiple right-hand sides.
ungqr_batch	*ungqr_batch_scratchpad_size*	Generates the complex unitary matrix Q_i with the QR factorization formed by geqrf_batch.

geqrf_batch

Computes the QR factorizations of a batch of general matrices.

Description

`geqrf_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

geqrf_batch (Buffer Version)

Description

The buffer version of `geqrf_batch` supports only the strided API.

Strided API

Syntax

```
namespace oneapi::mkl::lapack {
  void geqrf_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a, cl::sycl::buffer<T>
↪ &tau, std::int64_t stride_tau, std::int64_t batch_size, cl::sycl::buffer<T> &
↪scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in matrices A_i ($0 \leq m$).

n Number of columns in matrices A_i ($0 \leq n$).

a Array holding input matrices A_i.

lda Leading dimension of matrices A_i.

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as the number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *geqrf_batch_scratchpad_size* function.

Output Parameters

a Factorization data as follows: The elements on and above the diagonal of A_i contain the $\min(m, n) \times n$ upper trapezoidal matrices R_i (R_i is upper triangular if $m \geq n$); the elements below the diagonal, with the array τ_i, contain the orthogonal matrix Q_i as a product of $\min(m, n)$ elementary reflectors.

tau Array to store batch of τ_i, each of size $\min(m, n)$, containing scalars that define elementary reflectors for the matrices Q_i in its decomposition in a product of elementary reflectors.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

geqrf_batch (USM Version)

Description

The USM version of `geqrf_batch` supports the group API and strided API.

Group API

The routine forms the $Q_i R_i$ factorizations of a general $m \times n$ matrices A_i, $i \in \{1...batch\_size\}$, where `batch_size` is the sum of all parameter group sizes as provided with `group_sizes` array. No pivoting is performed during factorization. The routine does not form the matrices Q_i explicitly. Instead, Q_i is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q_i in this representation. The total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event geqrf_batch(cl::sycl::queue &queue, std::int64_t *m, std::int64_t␣
↪*n, T **a, std::int64_t *lda, T **tau, std::int64_t group_count, std::int64_t␣
↪*group_sizes, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_
↪class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Array of `group_count` m_g parameters. Each m_g specifies the number of rows in matrices A_i from array `a`, belonging to group g.

n Array of `group_count` n_g parameters. Each n_g specifies the number of columns in matrices A_i from array `a`, belonging to group g.

a Array of `batch_size` pointers to input matrices A_i, each of size $\text{lda}_g \cdot n_g$ (g is an index of group to which A_i belongs)

lda Array of `group_count` lda_g parameters, each representing the leading dimensions of input matrices A_i from array `a`, belonging to group g.

group_count Specifies the number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as the number of floating point elements of type `T`. Size should not be less than the value returned by the Group API of the *geqrf_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Factorization data as follows: The elements on and above the diagonal of A_i contain the $\min(m_g, n_g) \times n_g$ upper trapezoidal matrices R_i (R_i is upper triangular if $m_g \geq n_g$); the elements below the diagonal, with the array τ_i, contain the orthogonal matrix Q_i as a product of $\min(m_g, n_g)$ elementary reflectors. Here g is the index of the parameters group corresponding to the i-th decomposition.

tau Array of pointers to store arrays τ_i, each of size $\min(m_g, n_g)$, containing scalars that define elementary reflectors for the matrices Q_i in its decomposition in a product of elementary reflectors. Here g is the index of the parameters group corresponding to the i-th decomposition.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Strided API

The routine forms the $Q_i R_i$ factorizations of general $m \times n$ matrices A_i. No pivoting is performed. The routine does not form the matrices Q_i explicitly. Instead, Q_i is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q_i in this representation.

Syntax

```
namespace oneapi::mkl::lapack {
  sycl::event geqrf_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T␣
↪*a, std::int64_t lda, std::int64_t stride_a, T *tau, std::int64_t stride_tau,␣
↪std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in matrices A_i ($0 \leq m$).

n Number of columns in matrices A_i ($0 \leq n$).

a Array holding input matrices A_i.

lda Leading dimensions of A_i.

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as the number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *geqrf_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Factorization data as follows: The elements on and above the diagonal of A_i contain the $\min(m, n) \times n$ upper trapezoidal matrices R_i (R_i is upper triangular if $m \geq n$); the elements below the diagonal, with the array τ_i, contain the orthogonal matrix Q_i as a product of $\min(m, n)$ elementary reflectors.

tau Array to store batch of τ_i, each of size $\min(m, n)$, containing scalars that define elementary reflectors for the matrices Q_i in its decomposition in a product of elementary reflectors.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Parent topic: *LAPACK-like Extensions Routines*

geqrf_batch_scratchpad_size

Computes size of scratchpad memory required for the *geqrf_batch* function.

Description

`geqrf_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *geqrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t geqrf_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *m,␣
↪std::int64_t *n, std::int64_t *lda, std::int64_t group_count, std::int64_t *group_
↪sizes)
}
```

Input Parameters

queue Device queue where calculations will be performed.

m

Array of `group_count` m_g parameters.
Each of m_g specifies the number of rows in the matrices A_i belonging to group g.

n

Array of `group_count` n_g parameters.
Each of n_g specifies the number of columns in the matrices A_i belonging to group g.

lda Array of `group_count` lda_g parameters, each representing the leading dimensions of input matrices belonging to group g.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *geqrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *geqrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t geqrf_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda, std::int64_t stride_a, std::int64_t stride_tau,␣
↪std::int64_t batch_size)
};
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in the matrices A_i ($0 \leq m$).

n Number of columns in A_i ($0 \leq n$).

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *geqrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getrf_batch

Computes the LU factorizations of a batch of general matrices.

Description

`getrf_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

getrf_batch (Buffer Version)

Description

The buffer version of `getrf_batch` supports only the strided API.

Strided API

The routine computes the LU factorizations of general $m \times n$ matrices A_i as $A_i = P_i L_i U_i$, where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U_i is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

Syntax

```
namespace oneapi::mkl::lapack {
  void getrf_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a, cl::sycl::buffer
↪<std::int64_t> &ipiv, std::int64_t stride_ipiv, std::int64_t batch_size,␣
↪cl::sycl::buffer<T> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in matrices A_i ($0 \leq m$).

n Number of columns in matrices A_i ($0 \leq n$).

a Array holding input matrices A_i.

lda Leading dimension of matrices A_i.

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_ipiv Stride between the beginnings of arrays $ipiv_i$ inside the array `ipiv`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *getrf_batch_scratchpad_size* function.

Output Parameters

a L_i and U_i. The unit diagonal elements of L_i are not stored.

ipiv Array containing batch of the pivot indices ipiv_i each of size at least $\max(1, \min(m, n))$; for $1 \leq k \leq \min(m, n)$, where row k of A_i was interchanged with row $\text{ipiv}_i(k)$.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is positive, then the factorization has been completed, but some of U_i are exactly singular. Division by 0 will occur if you use the factor U_i for solving a system of linear equations.

The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

getrf_batch (USM Version)

Description

The USM version of `getrf_batch` supports the group API and strided API.

Group API

The routine computes the batch of LU factorizations of general $m \times n$ matrices A_i ($i \in \{1...batch\_size\}$) as $A_i = P_i L_i U_i$, where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U_i is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges. Total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrf_batch(cl::sycl::queue &queue, std::int64_t *m, std::int64_t␣
↪*n, T **a, std::int64_t *lda, std::int64_t **ipiv, std::int64_t group_count,␣
↪std::int64_t *group_sizes, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Array of `group_count` parameters m_g specifying the number of rows in matrices A_i ($0 \leq m_g$) belonging to group g.

n Array of `group_count` parameters n_g specifying the number of columns in matrices A_i ($0 \leq n_g$) belonging to group g.

a Array holding `batch_size` pointers to input matrices A_i.

lda Array of `group_count` parameters lda_g specifying the leading dimensions of A_i belonging to group g.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of group_count integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Group API of the *getrf_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a L_i and U_i. The unit diagonal elements of L_i are not stored.

ipiv Arrays of batch_size pointers to arrays containing pivot indices ipiv_i each of size at least $\max(1, \min(m_g, n_g))$; for $1 \leq k \leq \min(m_g, n_g)$, where row k of A_i was interchanged with row $\text{ipiv}_i(k)$.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is positive, then the factorization has been completed, but some of U_i are exactly singular. Division by 0 will occur if you use the factor U_i for solving a system of linear equations.

The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

Strided API

The routine computes the LU factorizations of general $m \times n$ matrices A_i as $A_i = P_i L_i U_i$, where P_i is a permutation matrix, L_i is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U_i is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrf_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪T *a, std::int64_t lda, std::int64_t stride_a, std::int64_t *ipiv, std::int64_t␣
↪stride_ipiv, std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size,␣
↪const cl::sycl::vector_class<cl::sycl::event> &events = {})
};
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in matrices A_i ($0 \leq m$).

n Number of columns in matrices A_i ($0 \leq n$).

a Array holding input matrices A_i.

lda Leading dimension of matrices A_i.

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_ipiv Stride between the beginnings of arrays ipiv_i inside the array `ipiv`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *getrf_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a L_i and U_i. The unit diagonal elements of L_i are not stored.

ipiv Array containing batch of the pivot indices ipiv_i each of size at least $\max(1, \min(m, n))$; for $1 \leq k \leq \min(m, n)$, where row k of A_i was interchanged with row $\text{ipiv}_i(k)$.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is positive, then the factorization has been completed, but some of U_i are exactly singular. Division by 0 will occur if you use the factor U_i for solving a system of linear equations.

The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getrf_batch_scratchpad_size

Computes size of scratchpad memory required for the *getrf_batch* function.

Description

`getrf_batch_scratchpad_size` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrf_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *m,␣
↪std::int64_t *n, std::int64_t *lda, std::int64_t group_count, std::int64_t *group_
↪sizes)
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Array of `group_count` parameters m_g specifying the number of rows in the matrices belonging to group g.

n Array of `group_count` parameters n_g specifying the number of columns in matrices belonging to group g.

lda Array of `group_count` parameters lda_g specifying the leading dimensions of matrices belonging to group g.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrf_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t lda, std::int64_t stride_a, std::int64_t stride_ipiv,␣
↪std::int64_t batch_size)
};
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in the matrices A_i ($0 \leq m$).

n Number of columns in A_i ($0 \leq n$).

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_ipiv Stride between the beginnings of arrays ipiv_i inside the array `ipiv`.

batch_size Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getri_batch

Computes the inverses of a batch of LU-factored matrices determined by *getrf_batch*.

Description

`getri_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

getri_batch (Buffer Version)

Description

The buffer version of `getri_batch` supports only the strided API.

Strided API

The routine computes the inverses A_i^{-1} of general matrices A_i. Before calling this routine, call the Strided API of the *getrf_batch (Buffer Version)* function to factorize A_i.

Syntax

```
namespace oneapi::mkl::lapack {
  void getri_batch(cl::sycl::queue &queue, std::int64_t n, cl::sycl::buffer<T> &a,␣
↪std::int64_t lda, std::int64_t stride_a, cl::sycl::buffer<std::int64_t> &ipiv,␣
↪std::int64_t stride_ipiv, std::int64_t batch_size, cl::sycl::buffer<T> &scratchpad,␣
↪std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

n Order of the matrices A_i ($0 \leq n$).

a Result of the Strided API of the *getrf_batch (Buffer Version)* function.

lda Leading dimension of A_i ($n \leq$ lda).

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

ipiv Arrays returned by the Strided API of the *getrf_batch (Buffer Version)* function.

stride_ipiv Stride between the beginnings of arrays ipiv_i inside the array `ipiv`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *getri_batch_scratchpad_size* function.

Output Parameters

a Inverse $n \times n$ matrices A_i^{-1}.

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

getri_batch (USM Version)

Description

The USM version of `getri_batch` supports the group API and strided API.

Group API

The routine computes the inverses A_i^{-1} of general matrices A_i, $i \in \{1...batch\_size\}$. Before calling this routine, call the Group API of the *getrf_batch (USM Version)* function to factorize A_i. Total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getri_batch(cl::sycl::queue &queue, std::int64_t *n, T **a,␣
↪std::int64_t *lda, std::int64_t **ipiv, std::int64_t group_count, std::int64_t␣
↪*group_sizes, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_
↪class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

n Array of `group_count` n_g parameters specifying the order of the matrices A_i ($0 \leq n_g$) belonging to group g.

a Result of the Group API of the *getrf_batch (USM Version)* function.

lda Array of `group_count` lda_g parameters specifying the leading dimensions of the matrices A_i ($n_g \leq \text{lda}_g$) belonging to group g.

ipiv Arrays returned by the Group API of the *getrf_batch (USM Version)* function.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the Group API of the *getri_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Inverse $n_g \times n_g$ matrices A_i^{-1}.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Strided API

The routine computes the inverses A_i^{-1} of general matrices A_i. Before calling this routine, call the Strided API of the *getrf_batch (USM Version)* function to factorize A_i.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getri_batch(cl::sycl::queue &queue, std::int64_t n, T *a,␣
↪std::int64_t lda, std::int64_t stride_a, std::int64_t *ipiv, std::int64_t stride_
↪ipiv, std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
};
```

Input Parameters

queue Device queue where calculations will be performed.

n Order of the matrices A_i ($0 \leq n$).

a Result of the Strided API of the *getrf_batch (USM Version)* function.

lda Leading dimension of A_i ($n \leq$ lda).

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

ipiv Arrays returned by the Strided API of the *getrf_batch (USM Version)* function.

stride_ipiv Stride between the beginnings of arrays ipiv_i inside the array `ipiv`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by the Strided API of the *getri_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Inverse $n \times n$ matrices A_i^{-1}.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Parent topic: *LAPACK-like Extensions Routines*

getri_batch_scratchpad_size

Computed size of scratchpad memory required for the *getri_batch* function.

Description

`getri_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getri_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getri_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *n,␣
↪std::int64_t *lda, std::int64_t group_count, std::int64_t *group_sizes)
}
```

Input Parameters

queue Device queue where calculations will be performed.

n Array of `group_count` n_g parameters specifying the order of the matrices belonging to group g.

lda Array of `group_count` lda_g parameters specifying the leading dimensions of the matrices belonging to group g.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getri_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getri_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getri_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t n,␣
↪std::int64_t lda, std::int64_t stride_a, std::int64_t stride_ipiv, std::int64_t␣
↪batch_size)
};
```

Input Parameters

queue Device queue where calculations will be performed.

n The order of the matrices A_i (0 le n).

lda Leading dimension of A_i ($n \leq$ lda).

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_ipiv Stride between the beginnings of arrays $ipiv_i$ inside the array ipiv.

batch_size Specifies the number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getri_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getrs_batch

Solves a system of linear equations with a batch of LU-factored square coefficient matrices, with multiple right-hand sides.

Description

`getrs_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

getrs_batch (Buffer Version)

Description

The buffer version of `getrs_batch` supports only the strided API.

Strided API

The routine solves for the following systems of linear equations X_i:
$A_i X_i = B_i$, if `trans=mkl::transpose::nontrans`
$A_i^T X_i = B_i$, if `trans=mkl::transpose::trans`
$A_i^H X_i = B_i$, if `trans=mkl::transpose::conjtrans`
Before calling this routine, the Strided API of the *getrf_batch (Buffer Version)* function should be called to compute the LU factorizations of A_i.

Syntax

```
namespace oneapi::mkl::lapack {
  void getrs_batch(cl::sycl::queue &queue, mkl::transpose trans, std::int64_t n,␣
↪std::int64_t nrhs, cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a,␣
↪cl::sycl::buffer<std::int64_t> &ipiv, std::int64_t stride_ipiv, cl::sycl::buffer<T>␣
↪&b, std::int64_t ldb, std::int64_t stride_b, std::int64_t batch_size,␣
↪cl::sycl::buffer<T> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

trans

Form of the equations:
If `trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for Xi.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i^H X_i = B_i$ is solved for X_i.

n Order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n$).

nrhs Number of right-hand sides ($0 \leq$ nrhs).

a Array containing the factorizations of the matrices A_i, as returned the Strided API of the *getrf_batch (Buffer Version)* function.

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices B_i inside the batch array `b`.

ipiv `ipiv` array, as returned by the Strided API of the *getrf_batch (Buffer Version)* function.

stride_ipiv Stride between the beginnings of arrays ipiv$_i$ inside the array `ipiv`.

b Array containing the matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb Leading dimension of B_i.

batch_size Specifies the number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *getrs_batch_scratchpad_size* function.

Output Parameters

b Solution matrices X_i.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then diagonal element of some of U_i is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

getrs_batch (USM Version)

Description

The USM version of `getrs_batch` supports the group API and strided API.

Group API

The routine solves the following systems of linear equations for X_i ($i \in \{1...batch\_size\}$):

$A_i X_i = B_i$, if `trans=mkl::transpose::nontrans`

$A_i^T X_i = B_i$, if `trans=mkl::transpose::trans`

$A_i^H X_i = B_i$, if `trans=mkl::transpose::conjtrans`

Before calling this routine, call the Group API of the *getrf_batch (USM Version)* function to compute the LU factorizations of A_i.

Total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrs_batch(cl::sycl::queue &queue, mkl::transpose *trans,␣
↪std::int64_t *n, std::int64_t *nrhs, T **a, std::int64_t *lda, std::int64_t **ipiv,␣
↪T **b, std::int64_t *ldb, std::int64_t group_count, std::int64_t *group_sizes, T␣
↪*scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class
↪<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

trans

Array of `group_count` parameters $trans_g$ indicating the form of the equations for the group g:
If `trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i^H X_i = B_i$ is solved for X_i.

n Array of `group_count` parameters n_g specifying the order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n_g$) belonging to group g.

nrhs Array of `group_count` parameters nrhs_g specifying the number of right-hand sides ($0 \leq \text{nrhs}_g$) for group g.

a Array of `batch_size` pointers to factorizations of the matrices A_i, as returned by the Group API of the:ref:*onemkl_lapack_getrf_batch_usm* function.

lda Array of `group_count` parameters lda_g specifying the leading dimensions of A_i from group g.

ipiv `ipiv` array, as returned by the Group API of the *getrf_batch (USM Version)* function.

b The array containing `batch_size` pointers to the matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb Array of `group_count` parameters ldb_g specifying the leading dimensions of B_i in the group g.

group_count Specifies the number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Group API of the *getrs_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b Solution matrices X_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of problems during calculations. The info code of the problem can be obtained by info() method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then diagonal element of some of U_i is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

Strided API

The routine solves the following systems of linear equations for X_i:
$A_i X_i = B_i$, if `trans=mkl::transpose::nontrans`
$A_i^T X_i = B_i$, if `trans=mkl::transpose::trans`
$A_i^H X_i = B_i$, if `trans=mkl::transpose::conjtrans`
Before calling this routine, the Strided API of the *getrf_batch* function should be called to compute the LU factorizations of A_i.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event getrs_batch(cl::sycl::queue &queue, mkl::transpose trans,␣
↪std::int64_t n, std::int64_t nrhs, T *a, std::int64_t lda, std::int64_t stride_a,␣
↪std::int64_t *ipiv, std::int64_t stride_ipiv, T *b, std::int64_t ldb, std::int64_t␣
↪stride_b, std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size,␣
↪const cl::sycl::vector_class<cl::sycl::event> &events = {})
};
```

Input Parameters

queue Device queue where calculations will be performed.

trans

Form of the equations:
If `trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i^H X_i = B_i$ is solved for X_i.

n Order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n$).

nrhs Number of right-hand sides ($0 \leq$ nrhs).

a Array containing the factorizations of the matrices A_i, as returned by the Strided API of the:ref:*onemkl_lapack_getrf_batch_usm* function.

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices B_i inside the batch array `b`.

ipiv `ipiv` array, as returned by getrf_batch (USM) function.

stride_ipiv Stride between the beginnings of arrays ipiv$_i$ inside the array `ipiv`.

b Array containing the matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb Leading dimensions of B_i.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *getrs_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b Solution matrices X_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then diagonal element of some of U_i is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these U_i matrices can be obtained by *exceptions()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

getrs_batch_scratchpad_size

Computes size of scratchpad memory required for the *getrs_batch* function.

Description

`getrs_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getrs_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrs_batch_scratchpad_size(cl::sycl::queue &queue, mkl::transpose␣
↪*trans, std::int64_t *n, std::int64_t *nrhs, std::int64_t *lda, std::int64_t *ldb,␣
↪std::int64_t group_count, std::int64_t *group_sizes)
}
```

Input Parameters

queue Device queue where calculations will be performed.

trans

Array of `group_count` parameters trans_g indicating the form of the equations for the group g:
If `trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i H X_i = B_i$ is solved for X_i.

n Array of `group_count` parameters n_g specifying the order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n_g$) belonging to group g.

nrhs Array of `group_count` parameters nrhsg specifying the number of right-hand sides ($0 \leq \text{nrhs}_g$) for group g.

lda Array of `group_count` parameters lda_g specifying the leading dimensions of A_i from group g.

ldb Array of `group_count` parameters ldb_g specifying the leading dimensions of B_i in the group g.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *getrs_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getrs_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t getrs_batch_scratchpad_size(cl::sycl::queue &queue, mkl::transpose␣
↪trans, std::int64_t n, std::int64_t nrhs, std::int64_t lda, std::int64_t stride_a,␣
↪std::int64_t stride_ipiv, std::int64_t ldb, std::int64_t stride_b, std::int64_t␣
↪batch_size)
};
```

Input Parameters

queue Device queue where calculations will be performed.

trans

Indicates the form of the equations:
`If trans = mkl::transpose::nontrans`, then $A_i X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::trans`, then $A_i^T X_i = B_i$ is solved for X_i.
If `trans = mkl::transpose::conjtrans`, then $A_i^H X_i = B_i$ is solved for X_i.

n Order of the matrices A_i and the number of rows in matrices B_i ($0 \leq n$).

nrhs Number of right-hand sides ($0 \leq$ nrhs).

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices B_i inside the batch array `b`.

stride_ipiv Stride between the beginnings of arrays ipivi inside the array `ipiv`.

ldb Leading dimension of B_i.

batch_size Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *getrs_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

orgqr_batch

Generates the orthogonal/unitary matrix Q_i of the QR factorizations for a group of general matrices.

Description

`orgqr_batch` supports the following precisions.

T
`float`
`double`

orgqr_batch (Buffer Version)

Description

The buffer version of `orgqr_batch` supports only the strided API.

Strided API

The routine generates the wholes or parts of $m \times n$ orthogonal matrices Q_i of the batch of QR factorizations formed by the Strided API of the *geqrf_batch (Buffer Version)* function.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with $m \geq p$.

To compute the whole matrices Q_i, use:

```
orgqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

```
orgqr_batch(queue, m, p, p, a, ...)
```

To compute the matrices Q_i^k of the QR factorizations of leading k columns of the matrices A_i:

```
orgqr_batch(queue, m, m, k, a, ...)
```

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
orgqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  void orgqr_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_
↪t k, cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a,␣
↪cl::sycl::buffer<T> &tau, std::int64_t stride_tau, std::int64_t batch_size,␣
↪cl::sycl::buffer<T> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in the matrices A_i ($0 \leq m$).

n Number of columns in the matrices A_i ($0 \leq n$).

k Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

a Array resulting after call to the Strided API of the *geqrf_batch (Buffer Version)* function.

lda Leading dimension of A_i (lda $\leq m$).

stride_a The stride between the beginnings of matrices A_i inside the batch array `a`.

tau Array resulting from call to the Strided API of the *geqrf_batch (Buffer Version)* function.

stride_tau Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size Specifies the number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *orgqr_batch_scratchpad_size* function.

Output Parameters

a Batch of n leading columns of the $m \times m$ orthogonal matrices Q_i.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

orgqr_batch (USM Version)

Description

The USM version of `orgqr_batch` supports the group API and strided API.

Group API

The routine generates the wholes or parts of $m \times n$ orthogonal matrices Q_i of the batch of QR factorizations formed by the Group API of the *geqrf_batch (USM Version)* function.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with $m \geq p$.

To compute the whole matrices Q_i, use:

```
orgqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

```
orgqr_batch(queue, m, p, p, a, ...)
```

To compute the matrices Q_i^k of the QR factorizations of leading k columns of the matrices A_i:

```
orgqr_batch(queue, m, m, k, a, ...)
```

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
orgqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgqr_batch(cl::sycl::queue &queue, std::int64_t *m, std::int64_t␣
↪*n, std::int64_t *k, T **a, std::int64_t *lda, T **tau, std::int64_t group_count,␣
↪std::int64_t *group_sizes, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Array of `group_count` m_g parameters as previously supplied to group version of geqrf_batch function.

n Array of `group_count` n_g parameters as previously supplied to group version of geqrf_batch function.

k Array of `group_count` k_g parameters as previously supplied to the Group API of the *geqrf_batch (USM Version)* function. The number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k_g \leq n_g$).

a Array resulting after call to the Group API of the *geqrf_batch (USM Version)* function.

lda Array of leading dimensions of A_i as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.

tau Array resulting after call to the Group API of the *geqrf_batch (USM Version)* function.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by Group API of the *orgqr_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a n_g leading columns of the $m_g \times m_g$ orthogonal matrices Q_i, where g is an index of group of parameters corresponding to Q_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Strided API

The routine generates the wholes or parts of $m \times n$ orthogonal matrices Q_i of the batch of QR factorizations formed by the Strided API of the *geqrf_batch (USM Version)* function.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with $m \geq p$.

To compute the whole matrices Q_i, use:

```
orgqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

```
orgqr_batch(queue, m, p, p, a, ...)
```

To compute the matrices Q_i^k of the QR factorizations of leading k columns of the matrices A_i:

```
orgqr_batch(queue, m, m, k, a, ...)
```

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
orgqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event orgqr_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪std::int64_t k, T *a, std::int64_t lda, std::int64_t stride_a, T *tau, std::int64_t␣
↪stride_tau, std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size,␣
↪const cl::sycl::vector_class<cl::sycl::event> &events = {})
};
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in the matrices A_i ($0 \leq m$).

n Number of columns in the matrices A_i ($0 \leq n$).

k Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

a Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

lda Leading dimension of A_i (lda $\leq m$).

stride_a The stride between the beginnings of matrices A_i inside the batch array `a`.

tau Array resulting from call to the Strided API of the *geqrf_batch (USM Version)* function.

stride_tau Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size Specifies the number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *orgqr_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Batch of n leading columns of the $m \times m$ orthogonal matrices Q_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Parent topic: *LAPACK-like Extensions Routines*

orgqr_batch_scratchpad_size

Computes size of scratchpad memory required for the *orgqr_batch* function.

Description

`orgqr_batch_scratchpad_size` supports the following precisions.

T
float
double

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *orgqr_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgqr_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *m,␣
↪std::int64_t *n, std::int64_t *k, std::int64_t *lda, std::int64_t group_count,␣
↪std::int64_t *group_sizes)
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Array of `group_count` m_g parameters.

n Array of `group_count` n_g parameters.

k Array of `group_count` kg parameters. The number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k_g \leq n_g$).

lda Array of leading dimensions of A_i.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *orgqr_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *orgqr_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t orgqr_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, std::int64_t lda, std::int64_t stride_a, std::int64_
↪t stride_tau, std::int64_t batch_size)
};
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in the matrices A_i ($0 \leq m$).

n Number of columns in the matrices Ai ($0 \leq n$).

k Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

lda Leading dimension of A_i (lda $\leq m$).

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau Stride between the beginnings of arrays tau_i inside the array `tau`.

batch_size Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *orgqr_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

potrf_batch

Computes the LU factorizations of a batch of general matrices.

Description

`potrf_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

potrf_batch (Buffer Version)

Description

The buffer version of `potrf_batch` supports only the strided API.

Strided API

> The routine forms the Cholesky factorizations of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, $i \in \{1...batch\_size\}$:
> $A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data if `uplo = mkl::uplo::upper`,
> $A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data if `uplo = mkl::uplo::lower`,
> where L_i is a lower triangular matrix and U_i is upper triangular.

Syntax

```
namespace oneapi::mkl::lapack {
  void potrf_batch(cl::sycl::queue &queue, mkl::uplo uplo, std::int64_t n,␣
↪cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a, std::int64_t batch_
↪size, cl::sycl::buffer<T> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

Indicates whether the upper or lower triangular part of A_i is stored and how A_i is factored:
If `uplo = mkl::uplo::upper`, the array `a` stores the upper triangular parts of the matrices A_i,
If `uplo = mkl::uplo::lower`, the array `a` stores the lower triangular parts of the matrices A_i.

n Order of the matrices A_i, ($0 \leq n$).

a Array containing batch of input matrices A_i, each of A_i being of size lda $\cdot n$ and holding either upper or lower triangular parts of the matrices A_i (see `uplo`).

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices A_i inside the batch.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *potrf_batch_scratchpad_size* function.

Output Parameters

a Cholesky factors U_i or L_i, as specified by `uplo`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then the leading minors of some of matrices (and therefore some matrices A_i themselves) are not positive-definite, and the factorizations could not be completed for these matrices from the batch. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The orders of corresponding not positive-definite leading minors of these matrices can be obtained by *exceptions()* method of exception object.

potrf_batch (USM Version)

Description

The USM version of `potrf_batch` supports the group API and strided API.

Group API

The routine forms the Cholesky factorizations of symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, $i \in \{1...batch\_size\}$:
$A_i = U_i^T U_i$ for real data ($A_i = U_i^H U_i$ for complex), if uplo_g is `mkl::uplo::upper`,
$A_i = L_i L_i^T$ for real data ($A_i = L_i L_i^H$ for complex), if uplo_g is `mkl::uplo::lower`,
where L_i is a lower triangular matrix and U_i is upper triangular, g is an index of group of parameters corresponding to A_i, and total number of problems to solve, `batch_size`, is a sum of sizes of all of the groups of parameters as provided by `group_sizes` array

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrf_batch(cl::sycl::queue &queue, mkl::uplo *uplo, std::int64_t␣
↪*n, T **a, std::int64_t *lda, std::int64_t group_count, std::int64_t *group_sizes,␣
↪T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class
↪<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

Array of `group_count` uplo_g parameters. Each uplo_g indicates whether the upper or lower triangular parts of the input matrices are provided:
If uplo_g is `mkl::uplo::upper`, input matrices from array `a` belonging to group g store the upper triangular parts,
If uplo_g is `mkl::uplo::lower`, input matrices from array `a` belonging to group g store the lower triangular parts.

n Array of `group_count` n_g parameters. Each n_g specifies the order of the input matrices from array a belonging to group g.

a Array of `batch_size` pointers to input matrices A_i, each being of size $\text{lda}_g \cdot n_g$ (g is an index of group to which A_i belongs to) and holding either upper or lower triangular part as specified by uplo_g.

lda Array of `group_count` lda_g parameters. Each lda_g specifies the leading dimensions of the matrices from a belonging to group g.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of group_count integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Group API of the *potrf_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Cholesky factors U_i or L_i, as specified by uplo_g from corresponding group of parameters.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then the leading minors of some of the input matrices (and therefore some matrices themselves) are not positive-definite, and the factorizations could not be completed for these matrices from the batch. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The orders of corresponding not positive-definite leading minors of these matrices can be obtained by *exceptions()* method of the exception object.

Strided API

The routine forms the Cholesky factorizations of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, $i \in \{1...batch\_size\}$:
$A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data if `uplo = mkl::uplo::upper`,

$A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data if `uplo = mkl::uplo::lower`, where L_i is a lower triangular matrix and U_i is upper triangular.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrf_batch(cl::sycl::queue &queue, mkl::uplo uplo, std::int64_t n,␣
↪T *a, std::int64_t lda, std::int64_t stride_a, std::int64_t batch_size, T␣
↪*scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class
↪<cl::sycl::event> &events = {})
};
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

Indicates whether the upper or lower triangular part of A_i is stored and how A_i is factored:
If `uplo = mkl::uplo::upper`, the array `a` stores the upper triangular parts of the matrices A_i,
If `uplo = mkl::uplo::lower`, the array `a` stores the lower triangular parts of the matrices A_i.

n Order of the matrices A_i, ($0 \leq n$).

a Array containing batch of input matrices A_i, each of A_i being of size lda $\cdot n$ and holding either upper or lower triangular parts of the matrices A_i (see `uplo`).

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices A_i inside the batch.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *potrf_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Cholesky factors U_i or L_i, as specified by `uplo`.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then the leading minors of some of matrices (and therefore some matrices A_i themselves) are not positive-definite, and the factorizations could not be completed for these matrices from the batch. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The orders of corresponding not positive-definite leading minors of these matrices can be obtained by *exceptions()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

potrf_batch_scratchpad_size

Computes size of scratchpad memory required for the *potrf_batch* function.

Description

`potrf_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *potrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrf_batch_scratchpad_size(cl::sycl::queue &queue, mkl::uplo *uplo,␣
↪std::int64_t *n, std::int64_t *lda, std::int64_t group_count, std::int64_t *group_
↪sizes)
}
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

Array of `group_count` uplo_g parameters.
Each of uplo_g indicates whether the upper or lower triangular parts of the input matrices are provided:
If uplo_g is `mkl::uplo::upper`, input matrices from array `a` belonging to group g store the upper triangular parts,
If uplo_g is `mkl::uplo::lower`, input matrices from array `a` belonging to group g store the lower triangular parts.

n

Array of `group_count` n_g parameters.
Each ng specifies the order of the input matrices belonging to group g.

lda

Array of `group_count` lda_g parameters.
Each ldag specifies the leading dimensions of the matrices belonging to group g.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *potrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *potrf_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrf_batch_scratchpad_size(cl::sycl::queue &queue, mkl::uplo uplo,␣
↪std::int64_t n, std::int64_t lda, std::int64_t stride_a, std::int64_t batch_size)
};
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

Indicates whether the upper or lower triangular part of A_i is stored and how A_i is factored:
If `uplo = mkl::uplo::upper`, the array `a` stores the upper triangular parts of the matrices A_i,
If `uplo = mkl::uplo::lower`, the array `a` stores the lower triangular parts of the matrices A_i.

n Order of the matrices A_i, ($0 \leq n$).

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices A_i inside the batch.

batch_size Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *potrf_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

potrs_batch

Computes the LU factorizations of a batch of general matrices.

Description

`potrs_batch` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

potrs_batch (Buffer Version)

Description

The buffer version of `potrs_batch` supports only the strided API.

Strided API

The routine solves for X_i the systems of linear equations $A_i X_i = B_i$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, given the Cholesky factorization of A_i, $i \in \{1...batch\_size\}$:
$A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data if `uplo = mkl::uplo::upper`,
$A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data if `uplo = mkl::uplo::lower`,
where L_i is a lower triangular matrix and U_i is upper triangular.
The systems are solved with multiple right-hand sides stored in the columns of the matrices B_i.
Before calling this routine, matrices A_i should be factorized by call to the Strided API of the *potrf_batch (Buffer Version)* function.

Syntax

```
namespace oneapi::mkl::lapack {
  void potrs_batch(cl::sycl::queue &queue, mkl::uplo uplo, std::int64_t n, std::int64_
↪t nrhs, cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a,␣
↪cl::sycl::buffer<T> &b, std::int64_t ldb, std::int64_t stride_b, std::int64_t batch_
↪size, cl::sycl::buffer<T> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

Indicates how the input matrices have been factored:
If `uplo = mkl::uplo::upper`, the upper triangle U_i of A_i is stored, where $A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data.
If `uplo = mkl::uplo::lower`, the upper triangle L_i of A_i is stored, where $A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data.

n The order of matrices A_i ($0 \leq n$).

nrhs The number of right-hand sides ($0 \leq$ nrhs).

a Array containing batch of factorizations of the matrices A_i, as returned by the Strided API of the *potrf_batch (Buffer Version)* function.

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices inside the batch array `a`.

b Array containing batch of matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb Leading dimension of B_i.

stride_b Stride between the beginnings of matrices B_i inside the batch array `b`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *potrs_batch_scratchpad_size* function.

Output Parameters

b Solution matrices X_i.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then for some of the matrices diagonal element of the Cholesky factor is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these matrices can be obtained by *exceptions()* method of exception object.

potrs_batch (USM Version)

Description

The USM version of `potrs_batch` supports the group API and strided API.

Group API

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrs_batch(cl::sycl::queue &queue, mkl::uplo *uplo, std::int64_t␣
↪*n, std::int64_t *nrhs, T **a, std::int64_t *lda, T **b, std::int64_t *ldb,␣
↪std::int64_t group_count, std::int64_t *group_sizes, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

Array of `group_count` uplo_g parameters.
Each of uplo_g indicates whether the upper or lower triangular parts of the input matrices are provided:
If uplo_g is `mkl::uplo::upper`, input matrices from array `a` belonging to group g store the upper triangular parts,
If uplo_g is `mkl::uplo::lower`, input matrices from array `a` belonging to group g store the lower triangular parts.

n

Array of `group_count` n_g parameters.
Each n_g specifies the order of the input matrices from array `a` belonging to group g.

nrhs

Array of `group_count` nrhs_g parameters.
Each nrhs_g specifies the number of right-hand sides supplied for group g in corresponding part of array `b`.

a Array of `batch_size` pointers to Cholesky factored matrices A_i as returned by the Group API of the *potrf_batch (USM Version)* function.

lda

Array of `group_count` lda_g parameters.
Each lda_g specifies the leading dimensions of the matrices from `a` belonging to group g.

b Array of `batch_size` pointers to right-hand side matrices B_i, each of size $\text{ldb}_g \cdot \text{nrhs}_g$, where g is an index of group corresponding to B_i.

ldb

Array of `group_count` ldb_g parameters.

Each ldb_g specifies the leading dimensions of the matrices from `b` belonging to group g.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Group API of the *potrs_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b Solution matrices X_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then for some of the matrices diagonal element of the Cholesky factor is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these matrices can be obtained by *exceptions()* method of exception object.

Strided API

The routine solves for X_i the systems of linear equations $A_i X_i = B_i$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrices A_i, given the Cholesky factorization of A_i, $i \in \{1...batch\_size\}$:

$A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data if `uplo = mkl::uplo::upper`,
$A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data if `uplo = mkl::uplo::lower`,
where L_i is a lower triangular matrix and U_i is upper triangular.
The systems are solved with multiple right-hand sides stored in the columns of the matrices B_i.
Before calling this routine, matrices A_i should be factorized by call to the Strided API of the *potrf_batch (USM Version)* function.

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event potrs_batch(cl::sycl::queue &queue, mkl::uplo uplo, std::int64_t n,␣
↪std::int64_t nrhs, T *a, std::int64_t lda, std::int64_t stride_a, T *b, std::int64_
↪t ldb, std::int64_t stride_b, std::int64_t batch_size, T *scratchpad, std::int64_t␣
↪scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
};
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

Indicates how the input matrices have been factored:
If `uplo = mkl::uplo::upper`, the upper triangle U_i of A_i is stored, where $A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data.
If `uplo = mkl::uplo::lower`, the upper triangle L_i of A_i is stored, where $A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data.

n The order of matrices A_i ($0 \leq n$).

nrhs The number of right-hand sides ($0 \leq nrhs$).

a Array containing batch of factorizations of the matrices A_i, as returned by the Strided API of the *potrf_batch (USM Version)* function.

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices inside the batch array `a`.

b Array containing batch of matrices B_i whose columns are the right-hand sides for the systems of equations.

ldb Leading dimension of B_i.

stride_b Stride between the beginnings of matrices B_i inside the batch array `b`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by the Strided API of the *potrs_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

b Solution matrices X_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of problems during calculations. The info code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

If `info` is zero, then for some of the matrices diagonal element of the Cholesky factor is zero, and the solve could not be completed. The indices of such matrices in the batch can be obtained with *ids()* method of the exception object. The indices of first zero diagonal elements in these matrices can be obtained by *exceptions()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

potrs_batch_scratchpad_size

Computes size of scratchpad memory required for the *potrs_batch* function.

Description

`potrs_batch_scratchpad_size` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *potrs_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrs_batch_scratchpad_size(cl::sycl::queue &queue, mkl::uplo *uplo,␣
↪std::int64_t *n, std::int64_t *nrhs, std::int64_t *lda, std::int64_t *ldb,␣
↪std::int64_t group_count, std::int64_t *group_sizes)
}
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

Array of `group_count` uplo_g parameters.
Each of uplo_g indicates whether the upper or lower triangular parts of the input matrices are provided:
If uplo_g is `mkl::uplo::upper`, input matrices from array `a` belonging to group g store the upper triangular parts,
If uplo_g is `mkl::uplo::lower`, input matrices from array `a` belonging to group g store the lower triangular parts.

n

Array of `group_count` n_g parameters.
Each n_g specifies the order of the input matrices belonging to group g.

nrhs

Array of `group_count` nrhs_g parameters.
Each rhs_g specifies the number of right-hand sides supplied for group g.

lda

Array of `group_count` lda_g parameters.
Each lda_g specifies the leading dimensions of the matrices belonging to group g.

ldb

Array of `group_count` ldb_g parameters.
Each ldb_g specifies the leading dimensions of the matrices belonging to group g.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of group_count integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *potrs_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *potrs_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t potrs_batch_scratchpad_size(cl::sycl::queue &queue, mkl::uplo uplo,␣
↪std::int64_t n, std::int64_t nrhs, std::int64_t lda, std::int64_t stride_a,␣
↪std::int64_t ldb, std::int64_t stride_b, std::int64_t batch_size)
};
```

Input Parameters

queue Device queue where calculations will be performed.

uplo

> Indicates how the input matrices have been factored:
> If `uplo = mkl::uplo::upper`, the upper triangle U_i of A_i is stored, where $A_i = U_i^T U_i$ for real data, $A_i = U_i^H U_i$ for complex data.
> If `uplo = mkl::uplo::lower`, the upper triangle L_i of A_i is stored, where $A_i = L_i L_i^T$ for real data, $A_i = L_i L_i^H$ for complex data.

n Order of matrices A_i ($0 \leq n$).

nrhs Number of right-hand sides ($0 \leq$ nrhs).

lda Leading dimension of A_i.

stride_a Stride between the beginnings of matrices inside the batch array `a`.

ldb Leading dimensions of B_i.

stride_b Stride between the beginnings of matrices B_i inside the batch array `b`.

batch_size Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *potrs_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

ungqr_batch

Generates the complex unitary matrices Q_i of the batch of QR factorizations formed by the *geqrf_batch* function.

Description

`ungqr_batch` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

ungqr_batch (Buffer Version)

Description

The buffer version of `ungqr_batch` supports only the strided API.

Strided API

> The routine generates the wholes or parts of :math`m times m` unitary matrices Q_i of the batch of QR factorization formed by the Strided API of the *geqrf_batch (Buffer Version)*.
>
> Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with :math`m ge p`.
>
> To compute the whole matrices Q_i, use:
>
> `ungqr_batch(queue, m, m, p, a, ...)`
>
> To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):
>
> `ungqr_batch(queue, m, p, p, a, ...)`
>
> To compute the matrices Q_i of the QR factorizations of leading k columns of the matrices A_i:
>
> `ungqr_batch(queue, m, m, k, a, ...)`

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
ungqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  void ungqr_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_
↪t k, cl::sycl::buffer<T> &a, std::int64_t lda, std::int64_t stride_a,␣
↪cl::sycl::buffer<T> &tau, std::int64_t stride_tau, std::int64_t batch_size,␣
↪cl::sycl::buffer<T> &scratchpad, std::int64_t scratchpad_size)
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in the matrices A_i ($0 \leq m$).

n Number of columns in the matrices A_i ($0 \leq n$).

k Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

a Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

lda Leading dimension of A_i (lda $\leq m$).

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

tau Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

stride_tau Stride between the beginnings of arrays tau_i inside the array `tau`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by strided version of the Strided API of the *ungqr_batch_scratchpad_size* function.

Output Parameters

a Array data is overwritten by `a` batch of n leading columns of the $m \times m$ unitary matrices Q_i.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of problems during calculations. The info code of the problem can be obtained by info() method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

ungqr_batch (USM Version)

Description

The USM version of `ungqr_batch` supports the group API and strided API.

Group API

The routine generates the wholes or parts of :math`m times m` unitary matrices Q_i of the batch of QR factorization formed by the Group API of the *geqrf_batch (Buffer Version)*.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with :math`m ge p`.

To compute the whole matrices Q_i, use:

```
ungqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

```
ungqr_batch(queue, m, p, p, a, ...)
```

To compute the matrices Q_i of the QR factorizations of leading k columns of the matrices A_i:

```
ungqr_batch(queue, m, m, k, a, ...)
```

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
ungqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungqr_batch(cl::sycl::queue &queue, std::int64_t *m, std::int64_t␣
↪*n, std::int64_t *k, T **a, std::int64_t *lda, T **tau, std::int64_t group_count,␣
↪std::int64_t *group_sizes, T *scratchpad, std::int64_t scratchpad_size, const␣
↪cl::sycl::vector_class<cl::sycl::event> &events = {})
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Array of `group_count` m_g parameters as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.

n Array of `group_count` n_g parameters as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.

k

Array of `group_count` k_g parameters as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.

The number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k_g \leq n_g$).

a Array resulting after call to the Group API of the *geqrf_batch (USM Version)* function.

lda Array of leading dimensions of A_i as previously supplied to the Group API of the *geqrf_batch (USM Version)* function.

tau Array resulting after call to the Group API of the *geqrf_batch (USM Version)* function.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by Group API of the *ungqr_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Matrices pointed to by array `a` are overwritten by n_g leading columns of the $m_g \times m_g$ orthogonal matrices Q_i, where g is an index of group of parameters corresponding to Q_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value. If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Strided API

The routine generates the wholes or parts of :math`m times m` unitary matrices Q_i of the batch of QR factorization formed by the Strided API of the *geqrf_batch (USM Version)*.

Usually Q_i is determined from the QR factorization of an $m \times p$ matrix A_i with :math` m ge p` .

To compute the whole matrices Q_i, use:

```
ungqr_batch(queue, m, m, p, a, ...)
```

To compute the leading p columns of Q_i (which form an orthonormal basis in the space spanned by the columns of A_i):

```
ungqr_batch(queue, m, p, p, a, ...)
```

To compute the matrices Q_i of the QR factorizations of leading k columns of the matrices A_i:

```
ungqr_batch(queue, m, m, k, a, ...)
```

To compute the leading k columns of Q_i^k (which form an orthonormal basis in the space spanned by leading k columns of the matrices A_i):

```
ungqr_batch(queue, m, k, k, a, ...)
```

Syntax

```
namespace oneapi::mkl::lapack {
  cl::sycl::event ungqr_batch(cl::sycl::queue &queue, std::int64_t m, std::int64_t n,␣
↪std::int64_t k, T *a, std::int64_t lda, std::int64_t stride_a, T *tau, std::int64_t␣
↪stride_tau, std::int64_t batch_size, T *scratchpad, std::int64_t scratchpad_size,␣
↪const cl::sycl::vector_class<cl::sycl::event> &events = {})
};
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in the matrices A_i ($0 \leq m$).

n Number of columns in the matrices A_i ($0 \leq n$).

k Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

a Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

lda Leading dimension of A_i (lda $\leq m$).

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

tau Array resulting after call to the Strided API of the *geqrf_batch (USM Version)* function.

stride_tau Stride between the beginnings of arrays tau_i inside the array `tau`.

batch_size Number of problems in a batch.

scratchpad Scratchpad memory to be used by routine for storing intermediate results.

scratchpad_size Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less then the value returned by strided version of the Strided API of the *ungqr_batch_scratchpad_size* function.

events List of events to wait for before starting computation. Defaults to empty list.

Output Parameters

a Array data is overwritten by `a` batch of n leading columns of the $m \times m$ unitary matrices Q_i.

Return Values

Output event to wait on to ensure computation is complete.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::lapack::batch_error

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

The `info` code of the problem can be obtained by *info()* method of exception object:

If `info = -n`, the n-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and *detail()* returns non zero, then passed scratchpad is of insufficient size, and required size should be not less then value returned by *detail()* method of exception object.

If `info` is not zero and *detail()* returns zero, then there were some errors for some of the problems in the supplied batch and `info` code contains the number of failed calculations in a batch.

Parent topic: *LAPACK-like Extensions Routines*

ungqr_batch_scratchpad_size

Computes size of scratchpad memory required for the *ungqr_batch* function.

Description

`ungqr_batch_scratchpad_size` supports the following precisions.

T
`std::complex<float>`
`std::complex<double>`

Group API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *ungqr_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungqr_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t *m,␣
↪std::int64_t *n, std::int64_t *k, std::int64_t *lda, std::int64_t group_count,␣
↪std::int64_t *group_sizes)
}
```

Input Parameters

queue Device queue where calculations will be performed.

m Array of `group_count` m_g parameters.

n Array of `group_count` n_g parameters.

k

Array of `group_count` k_g parameters.
Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k_g \leq n_g$).

lda Array of leading dimensions of A_i.

group_count Number of groups of parameters. Must be at least 0.

group_sizes Array of `group_count` integers. Array element with index g specifies the number of problems to solve for each of the groups of parameters g. So the total number of problems to solve, `batch_size`, is a sum of all parameter group sizes.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Group API of the *ungqr_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Strided API

Computes the number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *ungqr_batch* function.

Syntax

```
namespace oneapi::mkl::lapack {
  template <typename T>
  std::int64_t ungqr_batch_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,␣
↪std::int64_t n, std::int64_t k, std::int64_t lda, std::int64_t stride_a, std::int64_
↪t stride_tau, std::int64_t batch_size)
};
```

Input Parameters

queue Device queue where calculations will be performed.

m Number of rows in the matrices A_i ($0 \leq m$).

n Number of columns in the matrices A_i ($0 \leq n$).

k Number of elementary reflectors whose product defines the matrices Q_i ($0 \leq k \leq n$).

lda Leading dimensions of A_i (lda $\leq m$).

stride_a Stride between the beginnings of matrices A_i inside the batch array `a`.

stride_tau Stride between the beginnings of arrays τ_i inside the array `tau`.

batch_size Number of problems in a batch.

Return Values

Number of elements of type `T` the scratchpad memory should able to hold to be passed to the Strided API of the *ungqr_batch* function.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::unimplemented

oneapi::mkl::unsupported_device

oneapi::mkl::lapack::invalid_argument

> Exception is thrown in case of incorrect supplied argument value. Position of wrong argument can be determined by *info()* method of exception object.

Parent topic: *LAPACK-like Extensions Routines*

Note

Different arrays used as parameters to oneMKL LAPACK routines must not overlap.

Warning

LAPACK routines assume that input matrices do not contain IEEE 754 special values such as INF or NaN values. Using these special values may cause LAPACK to return unexpected results or become unstable.

Parent topic: *Dense Linear Algebra*

12.2.2 Sparse Linear Algebra

The oneAPI Math Kernel Library provides a Data Parallel C++ interface to some of the Sparse Linear Algebra routines.

Sparse BLAS provides basic operations on sparse vectors and matrices, and separates them into two stages: analysis (also called inspector stage or optimize stage) and execution. For a given matrix, the analysis would typically be called one time and the execution may be called multiple times. During the analysis stage, the API inspects the matrix properties including size, sparsity pattern and available parallelism and can apply matrix format or structure changes to enable a more optimized algorithm. In the execution stage, multiple routine calls can take advantage of the analysis stage data in order to improve performance.

In order to save information in between calls to Sparse BLAS computation routines, the *matrix_handle_t* type is introduced, that is essentially an opaque pointer, used to store data related to initial sparse matrix and data obtained during analysis stage.

Sparse BLAS

Sparse BLAS Routines provide basic operations on sparse vectors and matrices

Routines	Description
init_matrix_handle	Initialize the sparse matrix handle
release_matrix_handle	Release the sparse matrix handle
set_csr_data	Fills the internal CSR data structure
optimize_gemv	Optimize routine for gemv
optimize_trmv	Optimize routine for trmv
optimize_trsv	Optimize routine for trsv
gemv	Sparse matrix-dense vector product using a general sparse matrix
gemvdot	Sparse matrix-dense vector product followed by dot product
symv	Sparse matrix-dense vector product using a symmetric sparse matrix
trmv	Sparse matrix-dense vector product using a triangular sparse matrix
trsv	Solving a linear system with a triangular sparse matrix
gemm	Sparse matrix-dense matrix product using a general sparse matrix

- *Sparse storage formats*

init_matrix_handle

Initializes a matrix_handle_t object to default values.

Description and Assumptions

The oneapi::mkl::sparse::init_matrix_handle function initializes the `matrix_handle_t` object with default values.

Syntax

```
namespace oneapi::mkl::sparse {

   void init_matrix_handle (oneapi::mkl::sparse::matrix_handle_t *handle);

}
```

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Sparse BLAS*

release_matrix_handle

Releases internal data and sets matrix_handle_t object to NULL.

Description and Assumptions

The oneapi::mkl::sparse::release_matrix_handle releases any internal data that the `matrix_handle_t` object holds and sets it with defaults values, otherwise it throws an exception. The routine also waits for the dependencies to be finished before releasing any data in case of USM.

Syntax

```
namespace oneapi::mkl::sparse {

   void release_matrix_handle (oneapi::mkl::sparse::matrix_handle_t  handle,
                               const sycl::vector_class<sycl::event> &dependencies =
↪{});

}
```

Input parameter

handle Handle to object containing sparse matrix and other internal data. Created using one of the oneapi::mkl::sparse::set_<sparse_matrix_type>_structure routines.

dependencies List of events that `handle` depends on. The call waits on the events(if any) before resetting the `handle` to default values.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Sparse BLAS*

set_csr_data

Takes a matrix handle and the input CSR matrix arrays and fills the internal CSR data structure.

Description and Assumptions

Refer to *Supported Types* for a list of supported `<fp>` and `<intType>`. The mkl::sparse::set_csr_data routine takes a matrix handle for a sparse matrix of dimensions *num_rows* -by- *num_cols* represented in the CSR format, and fills the internal CSR data structure.

set_csr_data (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void set_csr_data (oneapi::mkl::sparse::matrix_handle_t  handle,
                      intType                               num_rows,
                      intType                               num_cols,
                      oneapi::mkl::index_base               index,
                      sycl::buffer<intType, 1>              &row_ptr,
                      sycl::buffer<intType, 1>              &col_ind,
                      sycl::buffer<fp, 1>                   &val);

}
```

Input Parameters

handle Handle to object containing sparse matrix and other internal data for subsequent DPC++ Sparse BLAS operations.

num_rows Number of rows of the input matrix .

num_cols Number of columns of the input matrix .

index Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ptr SYCL memory object containing an array of length `num_rows+1`. Refer to *CSR* format for detailed description of `row_ptr`.

col_ind SYCL memory object which stores an array containing the column indices in `index`-based numbering. Refer to *CSR* format for detailed description of `col_ind`.

val SYCL memory object which stores an array containing non-zero elements of the input matrix. Refer to *CSR* format for detailed description of `val`.

Output Parameters

handle Handle to object containing sparse matrix and other internal data for subsequent SYCL Sparse BLAS operations.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

set_csr_data (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   void set_csr_data (oneapi::mkl::sparse::matrix_handle_t  handle,
                      intType                               num_rows,
                      intType                               num_cols,
                      oneapi::mkl::index_base               index,
                      intType                               *row_ptr,
                      intType                               *col_ind,
                      fp                                    *val);

}
```

Input Parameters

handle Handle to object containing sparse matrix and other internal data for subsequent DPC++ Sparse BLAS operations.

num_rows Number of rows of the input matrix .

num_cols Number of columns of the input matrix .

index Indicates how input arrays are indexed. The possible options are described in *index_base* enum class.

row_ptr USM object containing an array of length `num_rows+1`. Refer to *CSR* format for detailed description of `row_ptr`

col_ind USM object which stores an array containing the column indices in `index`-based numbering. Refer to *CSR* format for detailed description of `col_ind`

val USM object which stores an array containing non-zero elements of the input matrix. Refer to *CSR* format for detailed description of `val`

Output Parameters

handle Handle to object containing sparse matrix and other internal data for subsequent SYCL Sparse BLAS operations.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Parent topic: *Sparse BLAS*

gemm

Computes a sparse matrix times dense matrix product.

Description and Assumptions

Refer to *Supported Types* for a list of supported <fp> and <intType> types. The oneapi::mkl::sparse::gemm routine computes a sparse matrix-dense matrix product defined as

$$C \leftarrow \alpha \text{op}(A)B + \beta C$$

where α and β are scalars, B and C are dense matrices and A is a sparse matrix. Dense matrix storage is in row-major format. Sparse matrix formats are compressed sparse row (CSR) formats.

gemm (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void gemm (sycl::queue                           &queue,
              oneapi::mkl::transpose                transpose_val,
              const fp                              alpha,
              oneapi::mkl::sparse::matrix_handle_t  A_handle,
              sycl::buffer<fp, 1>                   &B,
              const std::int64_t                    columns,
              const std::int64_t                    ldb,
              const fp                              beta,
              sycl::buffer<fp, 1>                   &C,
              const std::int64_t                    ldc);

}
```

Input parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix, A. Created using the oneapi::mkl::sparse::set_csr_data routine.

B The dense matrix in the sparse-dense matrix product. A one dimensional SYCL memory object containing an array of size at least `cols*ldb`, where `cols` = the number of columns of matrix $op(A)$.

columns Number of columns of matrix, C.

ldb Specifies the leading dimension of matrix, B. Should be greater than or equal to the number of columns of B which is `columns`.

beta Specifies the scalar `beta`.

C The dense matrix input/output array. A one-dimensional SYCL memory object containing an array of size at least `rows*ldc`, where `rows` = the number of rows of matrix $op(A)$.

ldc Specifies the leading dimension of matrix C . Must be greater than or equal to the number of columns of C which is `columns`.

Output Parameters

C Dense matrix output is overwritten by the updated matrix, C.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

gemm (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   sycl::event gemm (sycl::queue                              &queue,
                     oneapi::mkl::transpose                   transpose_val,
                     const fp                                 alpha,
                     oneapi::mkl::sparse::matrix_handle_t     A_handle,
                     const fp                                 *B,
                     const std::int64_t                       columns,
                     const std::int64_t                       ldb,
                     const fp                                 beta,
                     const fp                                 *C,
                     const std::int64_t                       ldc,
                     const sycl::vector_class<sycl::event>    &dependencies = {});

}
```

Input parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix, A. Created using the oneapi::mkl::sparse::set_csr_data routine.

B The dense matrix in the sparse-dense matrix product. A device accessible USM object containing an array of size at least `cols*ldb`, where `cols` = the number of columns of matrix $op(A)$.

columns Number of columns of matrix C.

ldb Specifies the leading dimension of matrix B. Should be greater than or equal to the number of columns of B.

beta Specifies the scalar `beta`.

C The dense matrix input/output array. A device accessible USM object containing an array of size at least `rows*ldc`, where `rows` = the number of rows of matrix $op(A)$.

ldc Specifies the leading dimension of matrix C . Must be greater than or equal to `columns`.

dependencies List of events that oneapi::mkl::sparse::gemm routine depends on. If omitted, defaults to no dependencies.

Output Parameters

C Dense matrix output is overwritten by the updated matrix C.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Return Values

Output event that can be waited upon or added as a dependency for the completion of gemm routine.

Parent topic: *Sparse BLAS*

gemv

Computes a sparse matrix-dense vector product.

Description and Assumptions

Refer to *Supported Types* for a list of supported `<fp>` and `<intType>`. The oneapi::mkl::sparse::gemv routine computes a sparse matrix-dense vector product defined as

$$y \leftarrow \alpha \text{op}(A)x + \beta y$$

where α and β are scalars, x and y are dense vectors, `A` is a sparse matrix.

gemv (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void gemv (sycl::queue                            &queue,
              oneapi::mkl::transpose                 transpose_val,
              const fp                               alpha,
              oneapi::mkl::sparse::matrix_handle_t   A_handle,
              sycl::buffer<fp, 1>                    &x,
              const fp                               beta,
              sycl::buffer<fp, 1>                    &y);

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix, A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x SYCL memory object containing an array of size at least equal to the number of columns of matrix op(A).

beta Specifies the scalar β.

y SYCL memory object containing an array of size at least equal to the number of rows of matrix op(A).

Output Parameters

y Overwritten by the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

gemv (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   sycl::event gemv (sycl::queue                              &queue,
                     oneapi::mkl::transpose                   transpose_val,
                     const fp                                 alpha,
                     oneapi::mkl::sparse::matrix_handle_t     A_handle,
                     const fp                                 *x,
                     const fp                                 beta,
                     const fp                                 *y,
                     const sycl::vector_class<sycl::event>    &dependencies = {});

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix, A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x Device-accessible USM object containing an array of size at least equal to the number of columns of matrix op(A).

beta Specifies the scalar β.

y Device-accessible USM object containing an array of size at least equal to the number of rows of matrix op(A).

dependencies List of events that oneapi::mkl::sparse::gemv routine depends on. If omitted, defaults to no dependencies.

Output Parameters

y Overwritten by the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Return Values

Output event that can be waited upon or added as a dependency for the completion of gemv routine.

Parent topic: *Sparse BLAS*

gemvdot

Computes a sparse matrix-dense vector product with dot product.

Description and Assumptions

Refer to *Supported Types* for a list of supported `<fp>` and `<intType>`. The oneapi::mkl::sparse::gemvdot routine computes a sparse matrix-dense vector product and dot product defined as

$$y \leftarrow \alpha \text{op}(A)x + \beta y$$

$$d \leftarrow xy$$

where:

`A` is a general sparse matrix, α, β, and `d` are scalars, x and y are dense vectors.

gemvdot (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void gemvdot (sycl::queue                              &queue,
                 oneapi::mkl::transpose                   transpose_val,
```

(continues on next page)

(continued from previous page)

```
                fp                                      alpha,
                oneapi::mkl::sparse::matrix_handle_t    A_handle,
                sycl::buffer<fp, 1>                     &x,
                fp                                      beta,
                sycl::buffer<fp, 1>                     &y,
                sycl::buffer<fp, 1>                     &d);

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x SYCL memory object containing an array of size at least equal to the number of columns of matrix op(A).

beta Specifies the scalar β.

y SYCL memory object containing an array of size at least equal to the number of rows of matrix op(A).

d SYCL scalar memory object used to store the result of dot product.

Output Parameters

y Overwritten by the updated vector `y`.

d Overwritten by the dot product of `x` and `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

gemvdot (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   sycl::event gemvdot (sycl::queue                              &queue,
                        oneapi::mkl::transpose                   transpose_val,
                        fp                                       alpha,
                        oneapi::mkl::sparse::matrix_handle_t     A_handle,
                        fp                                       *x,
                        fp                                       beta,
                        fp                                       *y,
                        fp                                       *d,
                        const sycl::vector_class<sycl::event> &dependencies = {});

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x Device-accessible USM object containing an array of size at least equal to the number of columns of matrix op(A).

beta Specifies the scalar β.

y Device-accessible USM object containing an array of size at least equal to the number of rows of matrix op(A)

d Device-accessible USM scalar object used to store the result of dot product.

dependencies List of events that oneapi::mkl::sparse::gemvdot routine depends on. If omitted, defaults to no dependencies.

Output Parameters

y Overwritten by the updated vector `y`.

d Overwritten by the dot product of `x` and `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented

oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Return Values

Output event that can be waited upon or added as a dependency for the completion of gemvdot routine.

Parent topic: *Sparse BLAS*

optimize_gemv

Performs internal optimizations for oneapi::mkl::sparse::gemv by analyzing the matrix structure.

Description and Assumptions

The oneapi::mkl::sparse::optimize_gemv routine analyzes matrix structure and performs optimizations. Optimized data is then stored in the handle.

optimize_gemv (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void optimize_gemv (sycl::queue                           &queue,
                       oneapi::mkl::transpose                transpose_val,
                       oneapi::mkl::sparse::matrix_handle_t handle);

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

handle Handle to object containing sparse matrix and other internal data. Created using the oneapi::mkl::sparse::set_csr_data routine.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument

oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

optimize_gemv (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   sycl::event optimize_gemv (sycl::queue                          &queue,
                              oneapi::mkl::transpose               transpose_val,
                              oneapi::mkl::sparse::matrix_handle_t handle,
                              sycl::vector_class<sycl::event>      &dependencies);

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

handle Handle to object containing sparse matrix and other internal data. Created using the oneapi::mkl::sparse::set_csr_data routine.

dependencies List of events that oneapi::mkl::sparse::optimize_gemv routine depends on.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Return Values

Output event that can be waited upon or added as a dependency for the completion of optimize_gemv routine.

Parent topic: *Sparse BLAS*

symv

Computes a sparse matrix-dense vector product for a symmetric part of the sparse matrix.

Description and Assumptions

Refer to *Supported Types* for a list of supported `<fp>` and `<intType>`. The oneapi::mkl::sparse::symv routine computes a sparse matrix-dense vector product over a symmetric part defined as

$$y \leftarrow \alpha Ax + \beta y$$

where:

α and β are scalars, x and y are dense vectors, `A` is a sparse matrix.

symv (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void symv (sycl::queue                           &queue,
              oneapi::mkl::uplo                     uplo_val,
              fp                                    alpha,
              oneapi::mkl::sparse::matrix_handle_t  A_handle,
              sycl::buffer<fp, 1>                   &x,
              fp                                    beta,
              sycl::buffer<fp, 1>                   &y);

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x SYCL memory object containing an array of size at least equal to the number of columns of A matrix.

beta Specifies the scalar β.

y SYCL memory object containing an array of size at least equal to the number of rows of A matrix.

Output Parameters

y Overwritten by the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

symv (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   sycl::event symv (sycl::queue                              &queue,
                     oneapi::mkl::uplo                        uplo_val,
                     fp                                       alpha,
                     oneapi::mkl::sparse::matrix_handle_t     A_handle,
                     fp                                       *x,
                     fp                                       beta,
                     fp                                       *y,
                     const sycl::vector_class<sycl::event> &dependencies = {});

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x Device-accessible USM object containing an array of size at least equal to the number of columns of A matrix.

beta Specifies the scalar β.

y Device-accessible USM object containing an array of size at least equal to the number of rows of A matrix.

dependencies List of events that oneapi::mkl::sparse::symv routine depends on. If omitted, defaults to no dependencies.

Output Parameters

y Overwritten by the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Return Values

Output event that can be waited upon or added as a dependency for the completion of symv routine.

Parent topic: *Sparse BLAS*

trmv

Computes a sparse matrix-dense vector product over upper or lower triangular matrix parts.

Description and Assumptions

Refer to *Supported Types* for a list of supported `<fp>` and `<intType>`. The oneapi::mkl::sparse::trmv routine computes a sparse matrix-dense vector product over a triangular part defined as

$$y \leftarrow \alpha \text{op}(A)x + \beta y$$

where: $alpha$ and $beta$ are scalars, x and y are dense vectors, `A` is a sparse matrix.

trmv (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void trmv (sycl::queue                          &queue,
              oneapi::mkl::uplo                    uplo_val
              oneapi::mkl::transpose               transpose_val,
              oneapi::mkl::diag                    diag_val
              fp                                   alpha,
              oneapi::mkl::sparse::matrix_handle_t A_handle,
```

(continues on next page)

(continued from previous page)

```
                sycl::buffer<fp, 1>                 &x,
                fp                                  beta,
                sycl::buffer<fp, 1>                 &y);

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

diag_val Specifies if the diagonal is unit or not. The possible options are described in *diag* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x SYCL memory object containing an array of size at least equal to the number of columns of matrix op(A).

beta Specifies the scalar β.

y SYCL memory object containing an array of size at least equal to the number of rows of matrix op(A).

Output Parameters

y Overwritten by the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

trmv (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   sycl::event trmv (sycl::queue                              &queue,
                     oneapi::mkl::uplo                        uplo_val
                     oneapi::mkl::transpose                   transpose_val,
                     oneapi::mkl::diag                        diag_val
                     fp                                       alpha,
                     oneapi::mkl::sparse::matrix_handle_t     A_handle,
                     fp                                       *x,
                     fp                                       beta,
                     fp                                       *y
                     const sycl::vector_class<sycl::event>    &dependencies = {});

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

diag_val Specifies if the diagonal is unit or not. The possible options are described in *diag* enum class.

alpha Specifies the scalar α.

A_handle Handle to object containing sparse matrix A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x Device-accessible USM object containing an array of size at least equal to the number of columns of matrix op(A).

beta Specifies the scalar β.

y Device-accessible USM object containing an array of size at least equal to the number of rows of matrix op(A).

dependencies List of events that oneapi::mkl::sparse::trmv routine depends on. If omitted, defaults to no dependencies.

Output Parameters

y Overwritten by the updated vector `y`.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument

oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Return Values

Output event that can be waited upon or added as a dependency for the completion of trmv routine.

Parent topic: *Sparse BLAS*

optimize_trmv

Performs internal optimizations for oneapi::mkl::sparse::trmv by analyzing the matrix structure.

Description and Assumptions

The oneapi::mkl::sparse::optimize_trmv routine analyzes matrix structure and performs optimizations. Optimized data is then stored in the handle.

optimize_trmv (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void optimize_trmv (sycl::queue                           &queue,
                       oneapi::mkl::uplo                     uplo_val,
                       oneapi::mkl::transpose                transpose_val,
                       oneapi::mkl::diag                     diag_val,
                       oneapi::mkl::sparse::matrix_handle_t handle);
}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

diag_val Specifies if the diagonal is unit or not. The possible options are described in *diag* enum class.

handle Handle to object containing sparse matrix and other internal data. Created using the oneapi::mkl::sparse::set_csr_data routine.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

optimize_trmv (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   sycl::event optimize_trmv (sycl::queue                           &queue,
                              oneapi::mkl::uplo                     uplo_val,
                              oneapi::mkl::transpose                transpose_val,
                              oneapi::mkl::diag                     diag_val,
                              oneapi::mkl::sparse::matrix_handle_t  handle,
                              sycl::vector_class<sycl::event>       &dependencies);
}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

diag_val Specifies if the diagonal is unit or not. The possible options are described in *diag* enum class.

handle Handle to object containing sparse matrix and other internal data. Created using the oneapi::mkl::sparse::set_csr_data routine.

dependencies List of events that oneapi::mkl::sparse::optimize_trmv routine depends on.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Return Values

Output event that can be waited upon or added as a dependency for the completion of optimize_trmv routine.

Parent topic: *Sparse BLAS*

trsv

Solves a system of linear equations for a triangular sparse matrix.

Description and Assumptions

Refer to *Supported Types* for a list of supported <fp> and <intType>. The oneapi::mkl::sparse::trsv routine solves a system of linear equations for a square matrix:

$$\text{op}(A)y \leftarrow x$$

where: A is a triangular sparse matrix of size m rows by m columns, op is a matrix modifier for matrix A, x and y are dense vectors of length at least m.

trsv (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void trsv (sycl::queue                          &queue,
              oneapi::mkl::uplo                    uplo_val
              oneapi::mkl::transpose               transpose_val,
              oneapi::mkl::diag                    diag_val
              oneapi::mkl::sparse::matrix_handle_t A_handle,
              sycl::buffer<fp, 1>                  &x,
              sycl::buffer<fp, 1>                  &y);

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

diag_val Specifies if the diagonal is unit or not. The possible options are described in *diag* enum class.

A_handle Handle to object containing sparse matrix A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x SYCL memory object containing an array of size at least equal to the number of columns of matrix op(A).

y SYCL memory object containing an array of size at least equal to the number of rows of matrix op(A).

Output Parameters

y SYCL memory object containing an array of size at least `nRows` filled with the solution to the system of linear equations.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

trsv (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   sycl::event trsv (sycl::queue                              &queue,
                     oneapi::mkl::uplo                        uplo_val
                     oneapi::mkl::transpose                   transpose_val,
                     oneapi::mkl::diag                        diag_val
                     oneapi::mkl::sparse::matrix_handle_t     A_handle,
                     fp                                       *x,
                     fp                                       *y
                     const sycl::vector_class<sycl::event> &dependencies = {});

}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

diag_val Specifies if the diagonal is unit or not. The possible options are described in *diag* enum class.

A_handle Handle to object containing sparse matrix A. Created using the oneapi::mkl::sparse::set_csr_data routine.

x Device-accessible USM object containing an array of size at least equal to the number of columns of matrix op(A).

y Device-accessible USM object containing an array of size at least equal to the number of rows of matrix op(A).

dependencies List of events that oneapi::mkl::sparse::trmv routine depends on. If omitted, defaults to no dependencies.

Output Parameters

y Device-accessible USM object containing an array of size at least `nRows` filled with the solution to the system of linear equations.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Return Values

Output event that can be waited upon or added as a dependency for the completion of trmv routine.

Parent topic: *Sparse BLAS*

optimize_trsv

Performs internal optimizations for oneapi::mkl::sparse::trsv by analyzing the matrix structure.

Description and Assumptions

The oneapi::mkl::sparse::optimize_trsv routine analyzes matrix structure and performs optimizations. Optimized data is then stored in the handle.

optimize_trsv (Buffer version)

Syntax

```
namespace oneapi::mkl::sparse {

   void optimize_trsv (sycl::queue                          &queue,
                       oneapi::mkl::uplo                    uplo_val,
                       oneapi::mkl::transpose               transpose_val,
                       oneapi::mkl::diag                    diag_val,
                       oneapi::mkl::sparse::matrix_handle_t handle);
}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

diag_val Specifies if the diagonal is unit or not. The possible options are described in *diag* enum class.

handle Handle to object containing sparse matrix and other internal data. Created using the oneapi::mkl::sparse::set_csr_data routine.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

optimize_trmv (USM version)

Syntax

```
namespace oneapi::mkl::sparse {

   sycl::event optimize_trsv (sycl::queue                          &queue,
                              oneapi::mkl::uplo                    uplo_val,
                              oneapi::mkl::transpose               transpose_val,
                              oneapi::mkl::diag                    diag_val,
                              oneapi::mkl::sparse::matrix_handle_t handle,
                              sycl::vector_class<sycl::event>      &dependencies);
}
```

Input Parameters

queue Specifies the SYCL command queue which will be used for SYCL kernels execution.

uplo_val Specifies which part is to be processed. The possible options are described in *uplo* enum class.

transpose_val Specifies operation `op()` on input matrix. The possible options are described in *transpose* enum class.

diag_val Specifies if the diagonal is unit or not. The possible options are described in *diag* enum class.

handle Handle to object containing sparse matrix and other internal data. Created using the oneapi::mkl::sparse::set_csr_data routine.

dependencies List of events that oneapi::mkl::sparse::optimize_trsv routine depends on.

Throws

This routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here.

oneapi::mkl::computation_error
oneapi::mkl::device_bad_alloc
oneapi::mkl::host_bad_alloc
oneapi::mkl::invalid_argument
oneapi::mkl::unimplemented
oneapi::mkl::uninitialized
oneapi::mkl::unsupported_device

Return Values

Output event that can be waited upon or added as a dependency for the completion of optimize_trsv routine.

Parent topic: *Sparse BLAS*

Supported Types

Data Types <fp>	Integer Types <intType>
`float`	`std::int32_t`
`double`	`std::int64_t`
`std::complex<float>`	
`std::complex<double>`	

General descriptions

matrix_handle_t

Type for the `handle` that can be used to store information about the initial sparse matrix (represented in a sparse format) and data created/obtained during the analysis stage to be used in the execution stage.

Sparse storage formats

CSR

There are a variety of matrix storage formats available for representing the sparse matrix. One of the most popular is compressed sparse row (CSR) format, that is represented by three arrays: *row_ptr*, *col_ind* and *val*, and *index* parameter.

num_rows	Number of rows in the sparse matrix.
num_cols	Number of columns in the sparse matrix.
index	Parameter that is used to specify whether the matrix has zero or one-based indexing.
val	An array that contains the non-zero elements of the sparse matrix stored row by row.
col_ind	An integer array of column indices for non-zero elements stored in the *val* array, such that *col_ind[i]* is the column number (using zero- or one-based indexing) of the element of the sparse matrix stored in *val[i]*.
row_ptr	An integer array of size equal to `num_rows + 1`. Element j of this integer array gives the position of the element in the *val* array that is first non-zero element in a row j of A. Note that this position is equal to *row_ptr[j] - index*. Last element of the *row_ptr* array (*row_ptr[num_rows]*) stores the sum of, number of nonzero elements and *index*(number of nonzero elements + *index*).

A sparse matrix can be represented in a CSR format in a following way (assuming zero-based indexing):

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & -1 & 4 \\ 3 & 0 & 0 \end{pmatrix}$$

num_rows	3				
num_cols	3				
index	0				
val	1	2	-1	4	3
col_ind	0	2	1	2	0
row_ptr	0	2	4	5	

Parent topic: *Sparse BLAS*

Parent topic: *Sparse Linear Algebra*

12.2.3 Discrete Fourier Transforms

The *Discrete Fourier Transform Functions* offer several options for computing Discrete Fourier Transforms (DFTs).

Discrete Fourier Transform Functions

The general form of the d-dimensional discrete Fourier transform(DFT) is

$$z_{k_1,k_2,\ldots,k_d} = \sigma \sum_{j_d=0}^{n_d-1} \cdots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1,j_2,\ldots,j_d} \exp\left[\delta 2\pi i \left(\sum_{\ell=1}^{d} \frac{j_\ell k_\ell}{n_\ell}\right)\right].$$

for $k_\ell = 0, \ldots, n_\ell - 1$ and $\ell \in \{1, \ldots, d\}$, where σ is a scale factor, $\delta = -1$ for the forward transform, and $\delta = +1$ for the backward(inverse) transform. In the forward transform, the input sequence $(w_{j_1,j_2,\ldots,j_d})$ belongs to the set of complex-valued sequences or real-valued sequences. Respective domains for the backward transform are represented by complex-valued sequences or complex conjugate-even sequences.

The discrete Fourier transform to be performed is defined by the creation of a *descriptor* class, with the associated configuration parameters, described in *Configuration Parameters and Enums*. Once the descriptor class is defined and *commit* is called and provided with a `sycl::queue` to define the device and context, it can be used for computing the forward and/or backward transformations. The available data storage formats for the various configurations are described in *Storage Formats*.

The routines and objects associated with computing a discrete Fourier transform.

Routines and Objects	Description
descriptor	A class to define the specific discrete Fourier transform problem to be applied.
descriptor::set_value	A member function of descriptor class to set non-default configuration parameters and define the DFT transformation to be applied.
descriptor::get_value	A member function of descriptor class to query configuration parameters that define the DFT transformation to be applied.
descriptor::commit	A member function of descriptor class to finalize the DFT descriptor before computations.
compute_forward	Computes the in-place/out-of-place forward transformation.
compute_backward	Computes the in-place/out-of-place backward transformation.

Parent topic: *oneMKL Domains*

Configuration Parameters and Enums

The following enum classes are defined in the `oneapi::mkl::dft` namespace which are used for configuring the discrete Fourier transform problem in the *descriptor* class prior to a call to *commit*.

enum class	Description
precision	The floating-point precision in which the transform is carried out. Used as a template argument for *descriptor* class.
domain	The forward domain data type for dft transformation. Used as a template argument for *descriptor* class.
config_param	The configuration parameters to specify the DFT transformation desired. These can be set and retrieved via the *set_value* and *get_value* functions.
config_value	Some possible enum values that the *config_param* configuration parameters can take on.

precision

The floating-point precision in which the transform is to be carried out. The data must be presented in this precision, the computation is carried out in this precision, and the result is delivered in this precision.

Syntax

```
enum class precision {
   SINGLE,
   DOUBLE
};
```

Value	Description
SINGLE	data and transforms are executed using single(fp32) precision
DOUBLE	data and transforms are executed using double(fp64) precision

domain

The discrete Fourier transform supports forward transformations on input sequences of two domains, from the forward domain to the backward domain. The backward transformation operates on input sequences from the backward domain to the forward domain. This `domain` value defines the forward domain and the backward domain is always implied to be complex-valued.

Syntax

```
enum class domain {
   REAL,
   COMPLEX
};
```

Value	Forward domain	Backward domain	Description
REAL	real-valued	complex-valued	Forward transformation is real-to-complex, backward transform is complex-to-real.
COMPLEX	complex-valued	complex-valued	Forward and backward transformations are complex-to-complex.

config_param

```
enum class config_param {

   FORWARD_DOMAIN,
   DIMENSION,
   LENGTHS,
   PRECISION,

   FORWARD_SCALE,
   BACKWARD_SCALE,

   NUMBER_OF_TRANSFORMS,

   COMPLEX_STORAGE,
   REAL_STORAGE,
   CONJUGATE_EVEN_STORAGE,

   PLACEMENT,

   INPUT_STRIDES,
   OUTPUT_STRIDES,

   FWD_DISTANCE,
   BWD_DISTANCE,

   WORKSPACE,
   ORDERING,
   TRANSPOSE,
   PACKED_FORMAT,
   COMMIT_STATUS
};
```

Many of the config_param enum's will take values in *config_value* or other `std::int64_t`, `std::vector<std::int64_t>`, or floating-point *precision* values as specified in the following table.

Value	Description
FORWARD_DOMAIN	Read-only value of forward *domain* set at *descriptor* construction time.
DIMENSION	Read-only value of the dimension of the transformation. Value is a positive integer of type `std::int64_t` set at *descriptor* construction.
LENGTHS	For a one-dimensional transform, the transform length is specified by a positive integer value represented in an integer scalar (`std::int64_t`). For multi-dimensional (≥ 2) transform, the lengths of each of the dimensions are supplied in an integer vector (`std::vector<std::int64_t>`) at *descriptor* construction time.
PRECISION	Read-only value of *precision* set at *descriptor* construction time.
FORWARD_SCALE	The forward transform is associated with a scale factor, σ, of real floating-point type *precision*, the default value is 1.0.
BACKWARD_SCALE	The backward transform is associated with a scale factor, σ, of real floating-point type *precision*, the default value is 1.0.
NUMBER_OF_TRANSFORMS	If you need to perform a large number of identical DFTs, you can do this in a single call to a compute forward function with the value of this equal to the actual number of the transforms. Takes a value of `std::int64_t` with default value of 1.
COMPLEX_STORAGE	Specifies the data storage format for *domain* with value of `COMPLEX`.
REAL_STORAGE	Specifies the data storage format for *domain* with value of `REAL`.
CONJUGATE_EVEN_STORAGE	Specifies the data storage format using conjugate-even symmetry of the data which allows to store only half of the mathematical results.
PLACEMENT	Choose between in-place(value is `config_value::INPLACE`) and out-of-place (value is `config_value::NOT_INPLACE`) transformations. For in-place transformation, the computational functions overwrite the input data with the output results. The default is `config_value::INPLACE`. When the configuration parameter is set to `config_value::NOT_INPLACE`, the input and output data sets must have no common elements.
INPUT_STRIDES	Defines the layout of multi-dimensional input data in computer memory. The value for a d-dimensional dataset is a d-dimensional vector of type `std::vector<std::int64_t>` representing offsets of elements of the appropriate data type as specified in *INPUT_STRIDES and OUTPUT_STRIDES*.
OUTPUT_STRIDES	Defines the layout of multi-dimensional output data in computer memory. The value for a d-dimensional dataset is a d-dimensional vector of type `std::vector<std::int64_t>` representing offsets of elements of the appropriate data type as specified in *INPUT_STRIDES and OUTPUT_STRIDES*.
FWD_DISTANCE	If computing multiple(batched) transforms, this parameter specifies the distance (in elements) between the first data elements of consecutive data sets in the forward domain. Provided in type `std::int64_t`, the default value is 1.
BWD_DISTANCE	If computing multiple(batched) transforms, this parameter specifies the distance (in elements) between the first data elements of consecutive data sets in the backward domain. Provided in type `std::int64_t`, the default value is 1.
WORKSPACE	Some FFT algorithm computation steps require a scratch space for permutations or other purposes. To manage the use of auxiliary storage, set to `config_value::ALLOW` to permit the use of auxiliary storage and `config_value::AVOID` to avoid using auxiliary storage if possible.
ORDERING	Some FFT algorithms apply an explicit permutation stage that can be time consuming. The value of `config_value::ORDERED` (default) applies the data ordering for all transformations. The value of `config_value::BACKWARD_SCRAMBLED` applies ordering for forward transform, but allows backward transform to have scrambled data if it gives a performance advantage.
TRANSPOSE	A boolean value to indicate providing the transposition of output results (for multi-dimensional transforms). Default value is `false`.
PACKED_FORMATS	Packing format for complex domain data storage of finite conjugate-even sequences from real-to-complex or complex-to-real transformations.
COMMIT_STATUS	Read-only value indicates whether the descriptor is ready for computation after a successful *commit*. Value of `config_value::COMMITTED` indicates a successful call to *commit*. A value of `config_value::UNCOMMITTED` (default) is set after descriptor constructor call and before successful call to *commit*.

config_value

These are some of the non-integer/floating-point values that the *config_param* configuration parameters can take on.

```
enum class config_value {

   // for config_param::COMMIT_STATUS
   COMMITTED,
   UNCOMMITTED,

   // for config_param::COMPLEX_STORAGE,
   //     config_param::REAL_STORAGE and
   //     config_param::CONJUGATE_EVEN_STORAGE
   COMPLEX_COMPLEX,
   REAL_COMPLEX,
   REAL_REAL,

   // for config_param::PLACEMENT
   INPLACE,
   NOT_INPLACE,

   // for config_param::ORDERING
   ORDERED,
   BACKWARD_SCRAMBLED,

   // Allow/avoid certain usages
   ALLOW,
   AVOID,
   NONE,

   // for config_param::PACKED_FORMAT for storing conjugate-even finite sequence in␣
↪real containers
   CCE_FORMAT

};
```

Parent topic: *Discrete Fourier Transform Functions*

Forward and Backward Scale

The forward and backward transformations are each associated with a scale factor, σ, having the default value of 1. For example, for a one-dimensional transform of length n , you can use the default scale of 1 for the forward transform and set the scale factor for the backward transform to be 1/ n , thus making the backward transform the inverse of the forward transform. Use real floating point data type corresponding to *precision*.

Parent topic *Configuration Parameters and Enums*

Number of Transforms

If you need to perform a large number of identical DFTs, you can do this in a single call to a compute* function with the value of this configuration parameter equal to the actual number of the transforms. The default value is 1. You can set this parameter to a positive integer value using the `std::int64_t` data type.

When setting the number of transforms to a value greater than one, you also need to specify the distance between the forward data sets and the distance between the backward data sets using the `config_param::FWD_DISTANCE` and `config_param::BWD_DISTANCE` configuration parameters corresponding to the specified *domain*.

Note:

- The data sets must not have common elements
- All the sets of data in each domain must be located within the same memory block.

Parent topic *Configuration Parameters and Enums*

Storage Formats

Depending on the value of the *domain* template value, the implementation of the DFT supports several storage schemes for input and output data. (See `Charles Van Loan, Computational Frameworks for the Fast Fourier Transform , SIAM, Philadelphia, 1992` for motivation of these schemes).

The data elements are placed within contiguous memory blocks, defined with generalized strides (see *INPUT_STRIDES and OUTPUT_STRIDES*). For multiple transforms, all sets of data should be located within the same memory block, and the data sets should be placed at the same distance from each other (see *NUMBER_OF TRANSFORMS* and `:config_param::FWD_DISTANCE`, `config_param::BWD_DISTANCE`).

The input data and strides sizes are stored and offsets counted in terms of elements of the data type (complex or real) based on the storage format and *forward domain* as seen in *Element types for complex-to-complex transformation and COMPLEX_STORAGE*, *Element types for real-to-complex transformations and REAL_STORAGE* and *Element types for real-to-complex transformations and CONJUGATE_EVEN_STORAGE*.

COMPLEX_STORAGE

For the *domain* template parameter with value `COMPLEX`, both input and output sequences belong to the complex domain. In this case, the configuration parameter COMPLEX_STORAGE can have one of the two values:`COMPLEX_COMPLEX` (default) or `REAL_REAL` .

COMPLEX_COMPLEX

With the `config_value::COMPLEX_COMPLEX` storage, complex-valued data sequences are stored in a single complex container (array/`sycl::buffer`), AZ, so that a complex-valued element $z_{k_1,k_2,\ldots,k_d}$ of the m-th d-dimensional sequence is accessed at $AZ[m{*}\text{distance}{+}\text{stride}_0{+}k_1{*}\text{stride}_1{+}k_2{*}\text{stride}_2{+}\cdots{+}k_d{*}\text{stride}_d]$ as a structure consisting of the real and imaginary parts. This code illustrates the use of `config_value::COMPLEX_COMPLEX` storage with three dimensions (n_1, n_2, n_3) and m batches:

```
std::complex<datatype> * AZ; // 2*sizeof(datatype)*n1*n2*n3*m
std::vector<std::int64_t> ios; // length 4 of sizes
std::int64_t iodist;
```

(continues on next page)

(continued from previous page)

```
// ...

// on input: Z(k1,k2,k3,m)
//   = AZ[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
compute_forward(descr, AZ); // complex-to-complex in-place FFT
// on output: Z{k1,k2,k3,m}
//   = AZ[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
```

REAL_REAL

With the `config_value::REAL_REAL` storage, complex-valued data sequences are stored by two real containers (arrays/`sycl::buffer`'s), AR and AI, so that a complex-valued element $z_{k_1,k_2,\ldots,k_d}$ of the m-th d-dimensional sequence has real part $AR[m * \text{distance} + \text{stride}_0 + k_1 * \text{stride}_1 + k_2 * \text{stride}_2 + \cdots + k_d * \text{stride}_d]$ and imaginary part $AI[m * \text{distance} + \text{stride}_0 + k_1 * \text{stride}_1 + k_2 * \text{stride}_2 + \cdots + k_d * \text{stride}_d]$. This code illustrates the use of `config_value::REAL_REAL` storage with three dimensions (n_1, n_2, n_3) and m batches:

```
datatype * AR; // sizeof(datatype)*n1*n2*n3*m
datatype * AI; // sizeof(datatype)*n1*n2*n3*m
std::vector<std::int64_t> ios; // length 4 of strides
std::int64_t iodist;

// ...

// on input: Z(k1,k2,k3,m)
//   =   AR[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
//   + i*AI[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
compute_forward(descr, AR, AI); // complex-to-complex in-place FFT
// on output: Z{k1,k2,k3,m}
//   =   AR[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
//   + i*AI[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
```

REAL_STORAGE

For the *domain* template parameter with value `REAL`, only the value of `REAL_REAL` is supported.

REAL_REAL

With the `REAL_REAL` storage, real-valued data sequences in a real domain are stored by one real container (array/`sycl::buffer`), AR, so that a real-valued element $r_{k_1,k_2,\ldots,k_d}$ of the m-th d-dimensional sequence is accessed as $AR[m * \text{distance} + \text{stride}_0 + k_1 * \text{stride}_1 + k_2 * \text{stride}_2 + \cdots + k_d * \text{stride}_d]$. This code illustrates the use of `config_value::REAL_REAL` storage with three dimensions (n_1, n_2, n_3) and m batches:

```
datatype * AR; // sizeof(datatype)*n1*n2*n3*m
datatype * AI; // sizeof(datatype)*n1*n2*n3*m
std::vector<std::int64_t> ios; // length 4 of strides
std::int64_t iodist;

// ...

// on input: R(k1,k2,k3,m)
//   =   AR[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
```

(continues on next page)

(continued from previous page)

```
compute_forward(descr, AR, AI); // real-to-complex in-place FFT
// on output: Z{k1,k2,k3,m}
//   =   AR[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
//   + i*AI[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
```

CONJUGATE_EVEN_STORAGE

For the *domain* template parameter with value `REAL` and considered as a conjugate-even domain, the value of `config_value::COMPLEX_COMPLEX` is supported. The conjugate-even symmetry of the data enables storing only about a half of the whole mathematical result, so that one part of it can be directly referenced in the memory while the other part can be reconstructed depending on the selected storage configuration. The `config_param::PACKED_FORMAT` configuration parameter defines how the data is packed. Possible values for `config_param::PACKED_FORMAT` depend on the values of the `config_param::CONJUGATE_EVEN_STORAGE` configuration parameter.

CONJUGATE_EVEN_STORAGE	Supported PACKED_FORMATS
COMPLEX_COMPLEX	`config_value::CCE_FORMAT` can be used with transforms of any dimension.

COMPLEX_COMPLEX

There is only one `config_param::PACKED_FORMAT` supported by the `config_value::COMPLEX_COMPLEX` value for `config_param::CONJUGATE_EVEN_STORAGE`, mainly the `config_value::CCE_FORMAT`. The complex-valued data sequence consists of one complex container (array/`sycl::buffer`), AZ, so that a complex-valued element $z_{k_1,k_2,\ldots,k_d}$ of the m-th d-dimensional sequence can be accessed or reconstructed as follows:

Consider a d-dimensional real-to-complex transform.

Because the input sequence, R, is real-valued, the mathematical result, Z, has conjugate-even symmetry: $z_{k_1,k_2,\ldots,k_d} = \text{conjugate}(z_{n_1-k_1,n_2-k_2,\ldots,n_d-k_d})$, where index arithmetic is performed modulo the length of the respective dimension. Obviously, the first element of the result is real-valued: $z_{0,0,\ldots,0} = \text{conjugate}(z_{0,0,\ldots,0})$.

For dimensions with even lengths, some of the other elements are real-valued as well. For example, if n_s is even, then $z_{0,0,\ldots,\frac{n_s}{2},0,\ldots,0} = \text{conjugate}(z_{0,0,\ldots,\frac{n_s}{2},0,\ldots,0})$. With the conjugate-even symmetry, approximately a half of the result suffices to fully reconstruct it. For an arbitrary dimension, h , it suffices to store elements $z_{k_1,\ldots,k_h,\ldots,k_d}$ for the following indices:

- $k_h = 0, \ldots, \left[\frac{n_h}{2}\right]$
- $k_i = 0, \ldots, n_i - 1$, where $i = 1, \ldots, d$ and $i \neq h$

and assuming that integer division rounds down.

The symmetry property enables reconstructing the remaining elements: for $k_h = \left[\frac{n_h}{2}\right] + 1, \ldots, n_h - 1$. The halved dimension is always assumed to be the dimension for which storage is contiguous in memory (see strides), for example in a 2D row-major format, it is the last dimension and for 2D column-major format it is the first dimension.

Packed complex domain formats for a 1D real-to-complex transformation considered as a conjugate-even-domain with COMPLEX_COMPLEX storage and $n = 2L$ (even size) or $n = 2L + 1$ (odd size).

$k =$	0	1	2	...	L-2	L-1	L
CCE	Z_0	Z_1	Z_2	...	Z_{L-2}	Z_{L-1}	Z_L

Packed complex domain formats for a 2D $n_1 \times n_2$ real-to-complex transformations considered as a conjugate-even-domain with COMPLEX_COMPLEX storage and $n_1 = 2K$ (even size) and $n_2 = 2L$ (even size) using row-major input data.

$k_1 \backslash k_2$	0	1	2	...	L-1	L
0	$Z_{0,0}$	$Z_{0,1}$	$Z_{0,2}$	...	$Z_{0,L-1}$	$Z_{0,L}$
1	$Z_{1,0}$	$Z_{1,1}$	$Z_{1,2}$	...	$Z_{1,L-1}$	$Z_{1,L}$
2	$Z_{2,0}$	$Z_{2,1}$	$Z_{2,2}$	...	$Z_{2,L-1}$	$Z_{2,L}$
...	...	...	...	...	...	...
$n_1 - 2$	$Z_{n_1-2,0}$	$Z_{n_1-2,1}$	$Z_{n_1-2,2}$	...	$Z_{n_1-2,L-1}$	$Z_{n_1-2,L}$
$n_1 - 1$	$Z_{n_1-1,0}$	$Z_{n_1-1,1}$	$Z_{n_1-1,2}$	...	$Z_{n_1-1,L-1}$	$Z_{n_1-1,L}$

The following code illustrates usage of the `config_value::COMPLEX_COMPLEX` storage for a two-dimensional conjugate-even domain with row-major input data:

```
datatype * AR;                 // sizeof(datatype)*n1*n2*m
std::complex<datatype> * AZ;   // sizeof(datatype)*n1*n2*m
std::vector<std::int64_t> is; // length 3 of input strides
std::vector<std::int64_t> os; // length 3 of output strides
std::int64_t idist, odist;

// ...

// on input: R(k1,k2,m)
//    =   AR[ is[0] + k1*is[1] + k2*is[2] + m*idist ]
compute_forward(descr, AR, AZ); // real-to-complex out-of-place FFT
// on output:
// for k2=0,n2/2:      Z{k1,k2,m} = AZ[os[0] + k1*os[1] + k2*os[2] + m*odist]
// for k2=n2/2+1,n2-1: Z{k1,k2,m} = conj(AZ[os[0] + (n1-k1)%n1*os[1]
//                                          + (n2-k2)%n2*os[2] + m*odist])
```

For the backward transform, the input and output parameters and layouts exchange roles. Set the strides describing the layout in the backward/forward domain as input/output strides, respectively. For example:

```
// ...
descr.set_value(config_param::INPUT_STRIDES,  fwd_domain_strides);
descr.set_value(config_param::OUTPUT_STRIDES, bwd_domain_strides);
descr.commit(queue);
compute_forward(descr, ...);
// ...
descr.set_value(config_param::INPUT_STRIDES,  bwd_domain_strides);
descr.set_value(config_param::OUTPUT_STRIDES, fwd_domain_strides);
descr.commit(queue);
compute_backward(descr, ...);
```

Parent topic *Configuration Parameters and Enums*

INPUT_STRIDES and OUTPUT_STRIDES

The FFT interface provides configuration parameters that define the layout of multidimensional data in the computer memory. For d-dimensional data set, X, defined by dimensions $n_1 \times n_2 \times \cdots \times n_d$, the layout describes where a particular element $X(k_1, k_2, \ldots, k_d)$ of the data set is located. The memory address of the element $X(k_1, k_2, \ldots, k_d)$ is expressed by the formula: $X(k_1, k_2, \ldots, k_d)$ = the $+s_0 + k_1 * s_1 + k_2 * s_2 + \cdots + k_d * s_d$-th element of the container (`sycl::buffer` or USM pointer) provided to the compute function, where s_0 is the displacement and $s_1, \ldots, s_d$ are generalized strides. The configuration parameters `config_param::INPUT_STRIDES` and `config_param::OUTPUT_STRIDES` enable you to get and set these values. The configuration value is a $d + 1$ lengthed `std::vector<std::int64_t>` of values $(s_0, s_1, \ldots, s_d)$.

The offset is counted in elements of the data type (complex or real) defined by the descriptor configuration as tabulated below.

The computation functions take containers(`sycl::buffer` or USM pointer) which are typed according to the descriptor configuration parameters. Specifically, the *forward domain* which defines the type of transformation and the *storage format* configuration parameters: `config_param::COMPLEX_STORAGE`, `config_param::REAL_STORAGE` and `config_param::CONJUGATE_EVEN_STORAGE` define the type of the elements as shown here:

Assumed Element Types using complex-to-complex transform and `config_param::COMPLEX_STORAGE`:

COMPLEX_STORAGE	Element type of forward data	Element type of backward data
COMPLEX_COMPLEX	Complex	Complex
REAL_REAL	Real	Real

Assumed Element Types using real-to-complex transform and `config_param::REAL_STORAGE`:

REAL_STORAGE	Element type of forward data	Element type of backward data
REAL_REAL	Real	Real

Assumed Element Types using real-to-complex transform and `config_param::CONJUGATE_EVEN_STORAGE`:

CONJUGATE_EVEN_STORAGE	Element type of forward data	Element type of backward data
COMPLEX_COMPLEX	Real	Complex

The `config_param::INPUT_STRIDES` configuration parameter defines the layout of the input data, while the element type is defined by the forward domain for the *compute_forward* function and by the backward domain for the *compute_backward* function. The `config_param::OUTPUT_STRIDES` configuration parameter defines the layout of the output data, while the element type is defined by the backward domain for the *compute_forward* function and by the forward domain for *compute_backward* function.

For in-place transforms (`config_param::PLACEMENT=config_value::INPLACE`), the configuration set by `config_param::OUTPUT_STRIDES` is ignored when the element types in the forward and backward domains are the same. If they are different, set `config_param::OUTPUT_STRIDES` explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the first elements on input and output must coincide in each dimension.

Parent topic *Configuration Parameters and Enums*

FORWARD_DISTANCE and BACKWARD_DISTANCE

The FFT interface enables computation of multiple transforms. To compute multiple transforms, you need to specify the data distribution of the multiple sets of data. The distance between the first data elements of consecutive data sets, `FORWARD_DISTANCE` for forward *domain* data or `BACKWARD_DISTANCE` for backward *domain* data, specifies the distribution. The configuration setting is a value of `std::int64_t` data type.

The default value for both configuration settings is one. You must set this parameter explicitly if the number of transforms is greater than one (see *Number of Transforms*).

The distance is counted in elements of the data type defined by the descriptor configuration (rather than by the type of the variable passed to the computation functions). Specifically, the *domain* template parameter, and the `COMPLEX_STORAGE`, `REAL_STORAGE` and `CONJUGATE_EVEN_STORAGE` configuration parameters described in *Storage Formats* define the type of the elements as shown in the *complex_storage*, *real_storage* and *conjugate_even_storage* tables.

For in-place transforms (`PLACEMENT=INPLACE`), the configuration set by `FORWARD_DISTANCE` and `BACKWARD_DISTANCE` should be consistent, that is, the locations of the data sets for input and output must coincide.

Parent topic: *Configuration Parameters and Enums*

descriptor

The descriptor class defines a discrete Fourier transform problem to be computed.

Description

The discrete Fourier transform problem is defined through the use of the `oneapi::mkl::dft::descriptor` class which lives in the `oneapi::mkl::dft::` namespace. The enum and config_param values associated with the descriptor class can be found in *Configuration Parameters and Enums* including *precision*, *domain* and *config_param*. The descriptor class allows to set several configuration parameters using set_value (and query using get_value) and then upon call to *commit* with a `sycl::queue`, is ready to be used in computations on the specified device.

This class is then passed to a *compute_forward* or *compute_backward* function along with the data for the actual transformation to be applied.

Note: The *compute_forward* and *compute_backward* functions may need to be able to access the internals of the descriptor to apply the transform, this could be done for instance, by labeling them as friend functions of the descriptor class.

---

descriptor class

Syntax

The descriptor class lives in the `oneapi::mkl::dft` namespace.

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   class descriptor {
    public:

        // Syntax for 1-dimensional DFT
        descriptor(std::int64_t length);

        // Syntax for d-dimensional DFT
        descriptor(std::vector<std::int64_t> dimensions);

        ~descriptor();


        void set_value(config_param param, ...);

        void get_value(config_param param, ...);

        void commit(sycl::queue &queue);

    };


}
```

Descriptor class template parameters

precision **prec** Specifies the floating-point precision in which the transform is to be carried out.

domain **dom** Specifies the forward domain for the transformations.

Descriptor class member functions

Routines	Description
constructors	Initialize descriptor for 1-dimensional or N-dimensional transformations
set_value	Sets one particular configuration parameter with the specified configuration value.
get_value	Gets the configuration value of one particular configuration parameter.
commit	Performs all initialization for the actual FFT computation.

Descriptor class constructors

The constructors for the discrete Fourier transform `descriptor` class with default configuration settings for a given precision, forward *domain* type and dimension of the transform.

The constructors allocate memory for the descriptor data structure and instantiate it with all the default configuration settings for the precision, (forward) *domain*, and dimensions of the transform. The constructors do not perform any significant computational work, such as computation of twiddle factors. The function *commit* does this work after use of the function *set_value* to set values of all necessary parameters.

Syntax (one-dimensional transform)

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   descriptor<prec,dom>(std::int64_t length);

}
```

Syntax (multi-dimensional transform)

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   descriptor<prec,dom>(std::vector<std::int64_t> dimensions);

}
```

Input Parameters

length dimension(length) of data for a 1-dimensional transform.

dimensions vector of $d \geq 0$ dimensions(lengths) of data for a d-dimensional transform.

Throws

The *descriptor()* constructor shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::host_bad_alloc() If any memory allocations on host have failed, for instance due to insufficient memory.

oneapi::mkl::unimplemented() If length of `dimensions` vector is larger than is supported by the library implementation.

Descriptor class member table: *Descriptor class member functions*

set_value

Sets DFT configuration values before *commit*.

Description

This function sets one particular configuration parameter with the specified configuration value. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see *config_param*. All calls to `set_param` must be done before *commit*.

Syntax

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   void descriptor<prec,dom>::set_value(config_param param, ...);

}
```

Input Parameters

param The enum value of *config_param* to be set.

... The corresponding value or container corresponding to the specific parameter. Defined in *config_param*.

Throws

The *descriptor::set_value()* routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument() If the provided *config_param* or config_value is not valid.

oneapi::mkl::unimplemented() If the provided *config_param* or config_value is valid, but not supported by the library implementation.

Descriptor class member table: *Descriptor class member functions*

get_value

Retrieves current DFT configuration values.

Description

This function gets one particular configuration parameter with the specified configuration value. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see *config_param*.

Syntax

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   void descriptor<prec,dom>::get_value(config_param param, ...);

}
```

Input Parameters

param The enum value of *config_param* to be retrieved.

... The corresponding value or container corresponding to the specific parameter. Defined in *config_param*.

Throws

The *descriptor::get_value()* routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument() If the requested *config_param* is not correct.

Descriptor class member table: *Descriptor class member functions*

commit

Finalizes DFT descriptor after all configuration parameters have been set.

Description

This function completes initialization of a previously created descriptor, which is required before the descriptor can be used for FFT computations. Typically, committing the descriptor performs all initialization that is required for the actual FFT computation on the device specified through input queue. The initialization performed by the function may involve exploring different factorizations of the input length to find the optimal computation method.

All calls to the *set_value* function to change configuration parameters of a descriptor need to happen after the constructor call for the *descriptor* class and before a call to *commit*. Typically, a commit function call is immediately followed by a computation function call (see *compute_forward* or *compute_backward*)

Syntax

```
namespace oneapi::mkl::dft {

   template <oneapi::mkl::dft::precision prec, oneapi::mkl::dft::domain dom>
   void descriptor<prec,dom>::commit(sycl::queue& queue);

}
```

Input Parameters

queue Valid DPC++ queue specifying the device and context on which the transformation will be executed.

Throws

The following oneMKL exceptions may be thrown in this function:

The *descriptor::commit()* routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument() If the queue is found to be invalid in any way.

oneapi::mkl::host_bad_alloc() If any host side only memory allocations fail, for instance due to lack of memory.

oneapi::mkl::device_bad_alloc() If any device or shared memory allocation fail.

Descriptor class member table: *Descriptor class member functions*

Parent topic: *Discrete Fourier Transform Functions*

compute_forward

This function computes the forward transform defined by an instantiation of the *descriptor* class.

Description

The compute_forward function accepts the *descriptor* and one or more data parameters and in the case of USM data, any `sycl::event` dependencies. Given a successfully configured and committed descriptor, this function computes the forward transform, that is, the *transform* with the minus sign, $\delta = -1$, in the exponent.

The configuration parameters `config_param::COMPLEX_STORAGE`, `config_param::REAL_STORAGE` and `config_param::CONJUGATE_EVEN_STORAGE` define the layout of the input and output data and must be properly set in a call to *set_value*.

Note: The compute_forward function may need to access the internals and private/protected members of the *descriptor* class. This could be done, for instance, by labeling it as a friend function to the descriptor class.

---

compute_forward (Buffer version)

Syntax (In-place transform)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   void compute_forward( descriptor_type            &desc,
                         sycl::buffer<data_type, 1> &inout);

}
```

Syntax (In-place transform, using `config_param::COMPLEX_STORAGE=config_value::REAL_REAL` data format)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type typename data_type>
   void compute_forward( descriptor_type            &desc,
                         sycl::buffer<data_type, 1> &inout_re,
                         sycl::buffer<data_type, 1> &inout_im);
}
```

Syntax (Out-of-place transform)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   void compute_forward( descriptor_type              &desc,
                         sycl::buffer<input_type, 1>  &in,
                         sycl::buffer<output_type, 1> &out);
}
```

Syntax (Out-of-place transform, using `config_param::COMPLEX_STORAGE=config_value::REAL_REAL` data format)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   void compute_forward( descriptor_type              &desc,
                         sycl::buffer<input_type, 1>  &in_re,
                         sycl::buffer<input_type, 1>  &in_im,
                         sycl::buffer<output_type, 1> &out_re,
                         sycl::buffer<output_type, 1> &out_im);

}
```

Input Parameters

desc A fully configured and committed discrete Fourier transform descriptor class object, defining the type of transformation and data layout to be applied. At commit time, the `sycl::queue` has already been provided.

inout Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house both the input and output data sequences for the in-place transformation. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_re Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_im Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the input data sequence for the out-of-place transformation. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in_re Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of input data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in_im Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of input data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

Output Parameters

inout Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house both the input and output data sequences for the in-place transformation. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_re Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_im Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the output data sequence for the out-of-place transformation. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out_re Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of output data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out_im Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of output data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

Throws

The *oneapi::mkl::dft::compute_forward* routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument() If the provided *descriptor* class is invalid, for instance, if it is a nullptr or if the value of `config_param::COMMIT_STATUS` in descriptor is not `config_param::COMMITTED`.

compute_forward (USM version)

Syntax (In-place transform)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   sycl::event compute_forward( descriptor_type                                  &desc,
                                data_type                                        *inout,
                                const cl::sycl::vector_class<cl::sycl::event> &
↪dependencies = {});
}
```

Syntax (In-place transform, using `config_param::COMPLEX_STORAGE=config_value::REAL_REAL` data format)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   sycl::event compute_forward(descriptor_type                                   &desc,
                               data_type                                         *inout_
↪re,
                               data_type                                         *inout_
↪im,
                               const cl::sycl::vector_class<cl::sycl::event> &
↪dependencies = {});

}
```

Syntax (Out-of-place transform)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   sycl::event compute_forward( descriptor_type                                  &desc,
                                input_type                                       *in,
                                output_type                                      *out,
                                const cl::sycl::vector_class<cl::sycl::event> &
↪dependencies = {});

}
```

Syntax (Out-of-place transform, using `config_param::COMPLEX_STORAGE=config_value::REAL_REAL` data format)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   sycl::event compute_forward( descriptor_type                                  &desc,
                                input_type                                       *in_re,
                                input_type                                       *in_im,
                                output_type                                      *out_re,
                                output_type                                      *out_im,
                                const cl::sycl::vector_class<cl::sycl::event> &
↪dependencies = {});

}
```

Input Parameter

desc A fully configured and committed discrete Fourier transform descriptor class object, defining the type of transformation and data layout to be applied. At commit time, the `sycl::queue` has already been provided.

inout USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house both the input and output data sequences for the in-place transformation. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_re USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_im USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the input data sequence for the out-of-place transformation. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in_re USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of the input data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in_im USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of the input data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

dependencies A vector of `sycl::event`'s that represent the previously enqueued tasks that must be finished before this transformation can be started.

Output Parameters

inout USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house both the input and output data sequences for the in-place transformation. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_re USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_im USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the output data sequence for the out-of-place transformation. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out_re USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of the output data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out_im USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of the output data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

Throws

The *oneapi::mkl::dft::compute_forward()* routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument() If the provided *descriptor* class is invalid, for instance, if it is a nullptr or if the value of `config_param::COMMIT_STATUS` in descriptor is not `config_param::COMMITTED`. It will also be thrown if the input/output pointers are NULL.

Return Values

This function returns a `sycl::event` that allows to track progress of this transformation, and can be passed as a dependency to other routines that may depend on the results of this transformation to be finished before proceeding with the other operations.

Parent topic: *Discrete Fourier Transform Functions*

compute_backward

This function computes the backward transform defined by an instantiation of the *descriptor* class.

Description

The compute_backward function accepts the *descriptor* and one or more data parameters and in the case of USM data, any `syc::event` dependencies. Given a successfully configured and committed descriptor, this function computes the backward transform, that is, the *transform* with the plus sign, $\delta = +1$, in the exponent.

The configuration parameters `config_param::COMPLEX_STORAGE`, `config_param::REAL_STORAGE` and `config_param::CONJUGATE_EVEN_STORAGE` define the layout of the input and output data and must be properly set in a call to *set_value*.

Note: The compute_backward function may need to access the internals and private/protected members of the *descriptor* class. This could be done, for instance, by labeling it as a friend function to the descriptor class.

---

compute_backward (Buffer version)

Syntax (In-place transform)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   void compute_backward( descriptor_type            &desc,
                          sycl::buffer<data_type, 1> &inout );

}
```

Syntax (In-place transform, using `config_param::COMPLEX_STORAGE=config_value::REAL_REAL` data format)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   void compute_backward( descriptor_type            &desc,
                          sycl::buffer<data_type, 1> &inout_re,
                          sycl::buffer<data_type, 1> &inout_im);
}
```

Syntax (Out-of-place transform)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   void compute_backward( descriptor_type              &desc,
                          sycl::buffer<input_type, 1>  &in,
                          sycl::buffer<output_type, 1> &out);
}
```

Syntax (Out-of-place transform, using `config_param::COMPLEX_STORAGE=config_value::REAL_REAL` data format)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   void compute_backward( descriptor_type              &desc,
                          sycl::buffer<input_type, 1>  &in_re,
                          sycl::buffer<input_type, 1>  &in_im,
                          sycl::buffer<output_type, 1> &out_re,
                          sycl::buffer<output_type, 1> &out_im);

}
```

Input Parameters

desc A fully configured and committed discrete Fourier transform descriptor class object, defining the type of backward transformation and data layout to be applied. At commit time, the `sycl::queue` has already been provided.

inout Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house both the input and output data sequences for the in-place transformation. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_re Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_im Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of both the input and output data sequences for the in-place transformation when

using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the input data sequence for the out-of-place transformation. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in_re Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of input data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in_im Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of input data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

Output Parameters

inout Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house both the input and output data sequences for the in-place transformation. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_re Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_im Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the output data sequence for the out-of-place transformation. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out_re Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of output data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out_im Sycl buffer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of output data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

Throws

The *oneapi::mkl::dft::compute_backward()* routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument() If the provided *descriptor* class is invalid, for instance, if it is a nullptr or if the value of `config_param::COMMIT_STATUS` in descriptor is not `config_param::COMMITTED`.

compute_backward (USM version)

Syntax (In-place transform)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   sycl::event compute_backward( descriptor_type                                  &desc,
                                 data_type                                        *inout,
                                 const cl::sycl::vector_class<cl::sycl::event> &
↪dependencies = {});
}
```

Syntax (In-place transform, using `config_param::COMPLEX_STORAGE=config_value::REAL_REAL` data format)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename data_type>
   sycl::event compute_backward(descriptor_type                                  &desc,
                                data_type                                        *inout_
↪re,
                                data_type                                        *inout_
↪im,
                                const cl::sycl::vector_class<cl::sycl::event> &
↪dependencies = {});

}
```

Syntax (Out-of-place transform)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   sycl::event compute_backward( descriptor_type                                  &desc,
                                 input_type                                       *in,
                                 output_type                                      *out,
                                 const cl::sycl::vector_class<cl::sycl::event> &
↪dependencies = {});

}
```

Syntax (Out-of-place transform, using `config_param::COMPLEX_STORAGE=config_value::REAL_REAL` data format)

```
namespace oneapi::mkl::dft {

   template <typename descriptor_type, typename input_type, typename output_type>
   sycl::event compute_backward( descriptor_type                                  &desc,
                                 input_type                                       *in_re,
                                 input_type                                       *in_im,
                                 output_type                                      *out_
↪re,
                                 output_type                                      *out_
↪im,
                                 const cl::sycl::vector_class<cl::sycl::event> &
↪dependencies = {});

}
```

Input Parameters

desc A fully configured and committed discrete Fourier transform descriptor class object, defining the type of backward transformation and data layout to be applied. At commit time, the `sycl::queue` has already been provided.

inout USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house both the input and output data sequences for the in-place transformation. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_re USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_im USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the input data sequence for the out-of-place transformation. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in_re USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of the input data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

in_im USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of the input data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

dependencies A vector of `sycl::event`'s that represent the previously enqueued tasks that must be finished before this transformation can be started.

Output Parameters

inout USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house both the input and output data sequences for the in-place transformation. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_re USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

inout_im USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of both the input and output data sequences for the in-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the output data sequence for the out-of-place transformation. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out_re USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the real part of the output data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

out_im USM pointer containing an array of length no less than is specified at the *descriptor construction* time to house the imaginary part of the output data sequence for the out-of-place transformation when using the `config_value::REAL_REAL` format for the `config_param::COMPLEX_STORAGE` configuration parameter. Corresponds to the choice of `config_value::NOT_INPLACE` for the configuration parameter `config_param::PLACEMENT`.

Throws

The *oneapi::mkl::dft::compute_backward()* routine shall throw the following exceptions if the associated condition is detected. An implementation may throw additional implementation-specific exception(s) in case of error conditions not covered here:

oneapi::mkl::invalid_argument() If the provided *descriptor* class is invalid, for instance, if it is a nullptr or if the value of `config_param::COMMIT_STATUS` in descriptor is not `config_param::COMMITTED`. It will also be thrown if the input/output pointers are NULL.

Return Values

This function returns a `sycl::event` that allows to track progress of this transformation, and can be passed as a dependency to other routines that may depend on the results of this transformation to be finished before proceeding with the other operations.

Parent topic: *Discrete Fourier Transform Functions*

12.2.4 Random Number Generators

The oneAPI Math Kernel Library Random Number Generators provides a *set of routines* implementing commonly used *pseudorandom, quasi-random, and non-deterministic generators* with *continuous and discrete distributions*.

Random Number Generators (RNG) Overview

Definitions

The pseudo-random number generator is defined by a structure(S, μ, f, U, g), where:

- S is a finite set of states (the state space)
- μ is a probability distribution on S for the initial state (or seed) s_0
- $f : S \to S$ is the transition function
- U – a finite set of output symbols
- $g : S \to U$ an output function

The generation of random numbers is as follows:

1. Generate the initial state (called the seed) s_0 according to μ and compute $u_0 = g(s_0)$.
2. Iterate for $i = 1, ...,$ and $u_i = g(s_i)$. Output values u_i are the so-called random numbers produced by the PRNG.

In computational statistics, random variate generation is usually made in two steps:

1. Generating imitations of independent and identically distributed (i.i.d.) random variables having the uniform distribution over the interval $(0, 1)$
2. Applying transformations to these i.i.d. $U(0, 1)$ random variates in order to generate (or imitate) random variates and random vectors from arbitrary distributions.

Structure

RNG domain contains two classes types:

- Engines (basic random number generators) classes, which holds the state of generator and is a source of i.i.d. random. Refer to *Engines (Basic Random Number Generators)* for a detailed description.
- Distribution classes templates (transformation classes) for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These classes contain all of the distribution's parameters (including generation method). Refer to *Distributions* for a detailed description of the distributions.

The RNG domain also contains two types of free functions:

- Generation routines. The current routines are used to obtain random numbers from a given engine with proper statistics defined by a given distribution. Refer to the *Generate Routine* section for a detailed description.

- Service routines. The routines are used to modify the engine state. Refer to *Service Routines* for a description of these routines.

Engine classes work with both generation and service routines. Distribution classes are used in generation routines only. Refer to the *oneMKL RNG Usage Model* section for the description of typical RNG scenario.

oneMKL RNG Usage Model

Description

A typical algorithm for random number generators is as follows:

1. **Create and initialize the object for basic random number** generator.
   - **Use the skip_ahead or leapfrog function if it is required (used** in parallel with random number generation for Host and CPU devices).
2. Create and initialize the object for distribution generator.
3. **Call the generate routine to get random numbers with appropriate** statistical distribution.

The following example demonstrates generation of random numbers that is output of basic generator (engine) PHILOX4X32X10. The seed is equal to 777. The generator is used to generate 10,000 normally distributed random numbers with parameters `a` = 5 and `sigma`= 2. The purpose of the example is to calculate the sample mean for normal distribution with the given parameters.

Buffer-based example

```
#include <iostream>
#include <vector>

#include "CL/sycl.hpp"
#include "mkl_rng_sycl.hpp"

int main() {
    sycl::queue queue;
    const size_t n = 10000;
    const std::uint64_t seed = 777;
    std::vector<double> r(n);

    oneapi::mkl::rng::philox4x32x10 engine(queue, seed); // basic random number␣
↪generator object
    oneapi::mkl::rng::gaussian<double, oneapi::mkl::rng::gaussian_method::box_muller2>
↪ distr(5.0, 2.0); //  distribution object

    {
        //create buffer for random numbers
        sycl::buffer<double, 1> r_buf(r.data(), r.size());
        oneapi::mkl::rng::generate(distr, engine, n, r_buf); // perform generation
    }

    double s = 0.0;
    for(int i = 0; i < n; i++) {
        s += r[i];
    }
    s /= n;
```

(continues on next page)

(continued from previous page)

```
    std::cout << "Average = " << s << std::endl;
    return 0;
}
```

USM-based example

```
#include <iostream>
#include <vector>

#include "CL/sycl.hpp"
#include "mkl_rng_sycl.hpp"

int main() {
    sycl::queue queue;
    const size_t n = 10000;
    const std::uint64_t seed = 777;

    // create USM allocator
    sycl::usm_allocator<double, sycl::usm::alloc::shared> allocator(queue.get_
↪context(), queue.get_device());

    // create vector with USM allocator
    std::vector<double, sycl::usm_allocator<double, sycl::usm::alloc::shared>> r(n,␣
↪allocator);

    oneapi::mkl::rng::philox4x32x10 engine(queue, seed); // basic random number␣
↪generator object
    oneapi::mkl::rng::gaussian<double, oneapi::mkl::rng::gaussian_method::box_muller2>
↪ distr(5.0, 2.0); // distribution object

    auto event = oneapi::mkl::rng::generate(distr, engine, n, r.data()); // perform␣
↪generation
    // sycl::event object is returned by generate function for synchronization
    event.wait(); // synchronization can be also done by queue.wait()

    double s = 0.0;
    for(int i = 0; i < n; i++) {
        s += r[i];
    }
    s /= n;

    std::cout << "Average = " << s << std::endl;
    return 0;
}
```

USM usage

You can also use USM with raw pointers by using the sycl::malloc_shared/malloc_device functions.

Parent topic: *Random Number Generators*

Generate Routine

- *generate* Entry point to obtain random numbers from a given engine with proper statistics of a given distribution.

Parent topic: *Random Number Generators*

generate

Entry point to obtain random numbers from a given engine with proper statistics of a given distribution.

Description and Assumptions

oneapi::mkl::rng::generate function produces random numbers sequence from the given engine object and applied transformation from a given distribution object.

generate (Buffer version)

Syntax

```
namespace oneapi::mkl::rng {
template<typename DistrType, typename EngineType> void generate (const DistrType&␣
↪distr, EngineType& engine, std::int64_t n, cl::sycl::buffer<typename␣
↪DistrType::result_type, 1>& r);
}
```

Template Parameters

DistrType Type of distribution which is used for random number generation.

EngineType Type of engine which is used for random number generation.

Input Parameters

distr Distribution object. See *Distributions* for details.

engine Engine object. See *Engines (Basic Random Number Generators)* for details.

n Number of random values to be generated.

Output Parameters

r sycl::buffer of generated values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when n > r.get_count(), or n < 0

generate (USM version)

Syntax

```
namespace oneapi::mkl::rng {
template<typename DistrType, typename EngineType> cl::sycl::event generate (const␣
↪DistrType& distr, EngineType& engine, std::int64_t n, typename DistrType::result_
↪type* r, const cl::sycl::vector_class<cl::sycl::event> & dependencies);
}
```

Template Parameters

DistrType Type of distribution which is used for random number generation.

EngineType Type of engine which is used for random number generation.

Input Parameters

distr Distribution object. See *Distributions* for details.

engine Engine object. See *Engines (Basic Random Number Generators)* for details.

n Number of random values to be generated.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

r pointer to generated values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when r == nullptr, or n < 0

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Generate Routine*

Engines (Basic Random Number Generators)

oneMKL RNG provides pseudorandom, quasi-random, and non-deterministic random number generators for Data Parallel C++:

Routine	Description
default_engine	The default random engine
mrg32k3a	The combined multiple recursive pseudorandom number generator `MRG32k3a` *[L'Ecuyer99a]*
philox4x32x10	Philox4x32-10 counter-based pseudorandom number generator with a period of 2^{128} `PHILOX4X32X10` *[Salmon11]*
mcg31m1	The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1) *[L'Ecuyer99]*
r250	The 32-bit generalized feedback shift register pseudorandom number generator `GFSR(250, 103)` *[Kirkpatrick81]*
mcg59	The 59-bit multiplicative congruential pseudorandom number generator `MCG(13`^{13}`, 2`^{59}`)` from NAG Numerical Libraries *[NAG]*
wichmann_hill	Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries *[NAG]*
mt19937	Mersenne Twister pseudorandom number generator `MT19937` *[Matsumoto98]* with period length 2^{19937}-1 of the produced sequence
mt2203	Set of 6024 Mersenne Twister pseudorandom number generators `MT2203` *[Matsumoto98] <onemkl_rng_bibliography>*, *[Matsumoto00]*. Each of them generates a sequence of period length equal to 2^{2203}-1. Parameters of the generators provide mutual independence of the corresponding sequences.
sfmt19937	SIMD-oriented Fast Mersenne Twister pseudorandom number generator `SFMT19937` *[Saito08]* with a period length equal to 2^{19937}-1 of the produced sequence.
sobol	Sobol quasi-random number generator *[Sobol76]*, *[Bratley88]*, which works in arbitrary dimension.
niederreiter	Niederreiter quasi-random number generator *[Bratley92]*, which works in arbitrary dimension.
ars5	ARS-5 counter-based pseudorandom number generator with a period of 2^{128}, which uses instructions from the AES-NI set `ARS5` *[Salmon11]*.
nondeterministic	Non-deterministic random number generator

For some basic generators, oneMKL RNG provides two methods of creating independent states in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo. The description of these functions can be found in the *Service Routines* section.

In addition, the MT2203 pseudorandom number generator is a set of 6024 generators designed to create up to 6024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [*Coddington94*].

Parent topic: *Random Number Generators*

- *default_engine* The default random engine (implementation defined)
- *mrg32k3a* The combined multiple recursive pseudorandom number generator MRG32k3a [L'Ecuyer99a]
- *philox4x32x10* A Philox4x32-10 counter-based pseudorandom number generator. [Salmon11].
- *mcg31m1* The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1) [L'Ecuyer99]
- *mcg59* The 59-bit multiplicative congruential pseudorandom number generator MCG(1313, 259) from NAG Numerical Libraries [NAG].
- *r250* The 32-bit generalized feedback shift register pseudorandom number generator GFSR(250,103)[Kirkpatrick81].
- *wichmann_hill* Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG].
- *mt19937* Mersenne Twister pseudorandom number generator MT19937 [Matsumoto98] with period length 2^{19937}-1 of the produced sequence.
- *sfmt19937* SIMD-oriented Fast Mersenne Twister pseudorandom number generator SFMT19937 [Saito08] with a period length equal to 2^{19937}-1 of the produced sequence.
- *mt2203* Set of 6024 Mersenne Twister pseudorandom number generators MT2203 [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to 2^{2203}-1. Parameters of the generators provide mutual independence of the corresponding sequences..
- *ars5* ARS-5 counter-based pseudorandom number generator with a period of 2^{128}, which uses instructions from the AES-NI set ARS5[Salmon11].
- *sobol* Sobol quasi-random number generator [Sobol76], [Bratley88], which works in arbitrary dimension.
- *niederreiter* Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension.
- *nondeterministic* Non-deterministic random number generator.

default_engine

Default random engine.

Description

The choice of engine type named by default_engine is implementation-defined. The implementation may select this type on the basis of performance, size, quality, or any combination of such factors.

type alias default_engine

Syntax

```
using default_engine = implementation-defined;
```

Parent topic: *Engines (Basic Random Number Generators)*

mrg32k3a

The combined multiple recursive pseudorandom number generator MRG32k3a.

Description

MRG32k3a engine is a 32-bit combined multiple recursive generator with two components of order 3 *[L'Ecuyer99a]*. MRG32k3a combined generator meets the requirements for modern RNGs, such as good multidimensional uniformity, or a long period ($p \approx 2^{191}$).

Generation algorithm

$x_n = a_{11}x_{n-1} + a_{12}x_{n-2} + a_{13}x_{n-3} (mod\ m_1)$

$y_n = a_{21}y_{n-1} + a_{22}y_{n-2} + a_{23} (mod\ m_2)$

$z_n = x_n - y_n (mod\ m_1)$

$u_n = z_n / m_1$

$a_{11} = 0, a_{12} = 1403580, a_{13} = -810728, m_1 = 2^{32} - 209$

$a_{21} = 527612, a_{22} = 0, a_{23} = -1370589, m_2 = 2^{32} - 22853$

class mrg32k3a

Syntax

```
namespace oneapi::mkl::rng {
class mrg32k3a {
public:
    static constexpr std::uint32_t default_seed = 1;

    mrg32k3a(sycl::queue queue, std::uint32_t seed = default_seed);

    mrg32k3a(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    mrg32k3a(const mrg32k3a& other);

    mrg32k3a(mrg32k3a&& other);

    mrg32k3a& operator=(const mrg32k3a& other);

    mrg32k3a& operator=(mrg32k3a&& other);

    ~mrg32k3a();
};
}
```

Class Members

Routine	Description
sycl::queue queue, std::uint32_t seed = default_seed	Constructor for common seed initialization of the engine
mrg32k3a(sycl::queue queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine
mrg32k3a(const mrg32k3a& other)	Copy constructor
mrg32k3a(mrg32k3a&& other)	Move constructor
mrg32k3a& operator=(const mrg32k3a& other)	Copy assignment operator
mrg32k3a& operator=(mrg32k3a&& other)	Move assignment operator

Constructors

```
mrg32k3a::sycl::queue queue, std::uint32_t seed = default_seed
```

Input Parameters

queue Valid `sycl::queue object`, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume $x_{-3} = seed \ mod \ m_1, x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$.

```
mrg32k3a::mrg32k3a(sycl::queue queue, std::initializer_list<std::uint32_t> seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume if $n = 0 : x_{-3} = x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 1 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 2 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 3 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = y_{-2} = y_{-1} = 1$

if $n = 4 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = y_{-1} = 1$

if $n = 5 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = seed[4] \ mod \ m_2, y_{-1} = 1$

if $n \geqslant 6 : x_{-3} = seed[0] \ mod \ m_1, x_{-2} = seed[1] \ mod \ m_1, x_{-1} = seed[2] \ mod \ m_1$

$y_{-3} = seed[3] \ mod \ m_2, y_{-2} = seed[4] \ mod \ m_2, y_{-1} = seed[5] \ mod \ m_2$

if the values prove to be $x_{-3} = x_{-2} = x_{-1} = 0$, assume $x_{-3} = 1$

if the values prove to be $y_{-3} = y_{-2} = y_{-1} = 0$, assume $y_{-3} = 1$

```
mrg32k3a::mrg32k3a(const mrg32k3a& other)
```

Input Parameters

other Valid `mrg32k3a` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mrg32k3a::mrg32k3a(mrg32k3a&& other)
```

Input Parameters

other Valid `mrg32k3a` object. The `queue` and state of the other engine is moved to the current engine.

```
mrg32k3a::mrg32k3a& operator=(const mrg32k3a& other)
```

Input Parameters

other Valid `mrg32k3a` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mrg32k3a::mrg32k3a& operator=(mrg32k3a&& other)
```

Input Parameters

other Valid `mrg32k3a` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

philox4x32x10

The Philox4x32x10 counter-based pseudorandom number generator.

Description

The Philox4x32x10 engine is a keyed family of generator of counter-based BRNG. The state consists of 128-bit integer counter c and two 32-bits keys k_0 and k_1.

Generation algorithm

The generator has 32-bit integer output obtained in the following way [*Salmon11*]:

1. $c_n = c_{n-1} + 1$
2. $\omega_n = f(c_n)$**, where f is a function that takes 128-bit argument and returns a 128-bit number. The returned number is obta**
   2.1. The argument c is interpreted as four 32-bit numbers $c = \overline{L_1 R_1 L_0 R_0}$, where $\overline{ABCD} = A \cdot 2^{96} + B \cdot 2^{64} + C \cdot 2^{32} + D$, put $k_0^0 = k_0, k_1^0 = k_1$.

   2.2. The following recurrence is calculated:

   $L_1^{i+1} = mullo(R_1^i, 0xD2511F53)$

   $R_1^{i+1} = mulhi(R_0^i, 0xCD9E8D57) \oplus k_0 \oplus L_0$

$L_0^{i+1} = mullo(R_0^i, 0xCD9E8D57)$

$R_0^{i+1} = mulhi(R_1^i, 0xD2511F53) \oplus k_1 \oplus L_1$

$k_0^{i+1} = k_0^i + 0xBB67AE85$

$k_1^{i+1} = k_1^i + 0x9E3779B9$, where $mulhi(a, b)$ and $mullo(a, b)$ are high and low parts of the $a \cdot b$ product respectively.

2.3. Put $f(c) = \overline{L_1^N R_1^N L_0^N R_0^N}$, where $N = 10$

3. Integer output: $r_{4n+k} = \omega_n(k)$, where $\omega_n(k)$ is the k-th 32-bit integer in quadruple $\omega_n, k = 0, 1, 2, 3$
4. Real output: $u_n = (int)r_n/2^{32} + 1/2$

class philox4x32x10

Syntax

```
namespace oneapi::mkl::rng {
class philox4x32x10 {
public:
static constexpr std::uint64_t default_seed = 0;

philox4x32x10(sycl::queue queue, std::uint64_t seed = default_seed);

philox4x32x10(sycl::queue queue, std::initializer_list<std::uint64_t> seed);

philox4x32x10(const philox4x32x10& other);

philox4x32x10(philox4x32x10&& other);

philox4x32x10& operator=(const philox4x32x10& other);

philox4x32x10& operator=(philox4x32x10&& other);

~philox4x32x10();
};
}
```

Class Members

Routine	Description
philox4x32x10(sycl::queue queue, std::uint64_t seed = default_seed)	Constructor for common seed initialization of the engine
philox4x32x10(sycl::queue queue, std::initializer_list<std::uint64_t> seed)	Constructor for extended seed initialization of the engine
philox4x32x10(const philox4x32x10& other)	Copy constructor
philox4x32x10(philox4x32x10&& other)	Move constructor
philox4x32x10& operator=(const philox4x32x10& other)	Copy assignment operator
philox4x32x10& operator=(philox4x32x10&& other)	Move assignment operator

Constructors

```
philox4x32x10::philox4x32x10(sycl::queue queue, std::uint64_t seed = default_seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume $k = seed, c = 0$, where k is a 64-bit key, c is a 128-bit counter.

```
philox4x32x10::philox4x32x10(sycl::queue queue, std::initializer_list<std::uint64_t>␣
↪seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume if $n = 0 : k = 0, c = 0$

if $n = 1 : k = seed[0], c = 0$

if $n = 2 : k = seed[0], c = seed[1]$

if $n = 3 : k = seed[0], c = seed[1] + seed[2] \cdot 2^{64}$

for $n > 3$ following arguments are ignored

```
philox4x32x10::philox4x32x10(const philox4x32x10& other)
```

Input Parameters

other Valid `philox4x32x10` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
philox4x32x10::philox4x32x10(philox4x32x10&& other)
```

Input Parameters

other Valid `philox4x32x10` r-value object. The `queue` and state of the other engine is moved to the current engine.

```
philox4x32x10::philox4x32x10& operator=(const philox4x32x10& other)
```

Input Parameters

other Valid `philox4x32x10` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
philox4x32x10::philox4x32x10& operator=(philox4x32x10&& other)
```

Input Parameters

other Valid `philox4x32x10` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

mcg31m1

The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1).

Description

The mcg31m1 engine is a 31-bit multiplicative congruential generator *[L'Ecuyer99]*. The mcg31m1 generator belongs to linear congruential generators with the period length of approximately 2^{31}. Such generators are still used as default random number generators in various software systems, mainly due to the simplicity of the portable versions implementation, speed, and compatibility with the earlier systems versions. However, their period length does not meet the requirements for modern basic generators. Still, the mcg31m1 generator possesses good statistic properties and you may successfully use it to generate random numbers of different distributions for small samplings.

Generation algorithm

$x_n = ax_{n-1}(mod\ m)$

$u_n = x_n/m$

$a = 1132489760, m = 2^{31} - 1$

class mcg31m1

Syntax

```
namespace oneapi::mkl::rng {
class mcg31m1 {
public:
    static constexpr std::uint32_t default_seed = 1;

    mcg31m1(sycl::queue queue, std::uint32_t seed = default_seed);

    mcg31m1(const mcg31m1& other);

    mcg31m1(mcg31m1&& other);

    mcg31m1& operator=(const mcg31m1& other);
```

(continues on next page)

(continued from previous page)

```
    mcg31m1& operator=(mcg31m1&& other);

    ~mcg31m1();
};
}
```

Class Members

Routine	Description
mcg31m1(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine
mcg31m1(const mcg31m1& other)	Copy constructor
mcg31m1(mcg31m1&& other)	Move constructor
mcg31m1& operator=(const mcg31m1& other)	Copy assignment operator
mcg31m1& operator=(mcg31m1&& other)	Move assignment operator

Constructors

```
mcg31m1::mcg31m1(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue Valid sycl::queue object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume $x_0 = seed \ mod \ 0x7FFFFFFF$, if $x_0 = 0$, assume $x_0 = 1$.

```
mcg31m1::mcg31m1(const mcg31m1& other)
```

Input Parameters

other Valid `mcg31m1` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mcg31m1::mcg31m1(mcg31m1&& other)
```

Input Parameters

other Valid `mcg31m1` object. The `queue` and state of the other engine is moved to the current engine.

```
mcg31m1::mcg31m1& operator=(const mcg31m1& other)
```

Input Parameters

other Valid `mcg31m1` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mcg31m1::mcg31m1& operator=(mcg31m1&& other)
```

Input Parameters

other Valid `mcg31m1` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

mcg59

The 59-bit multiplicative congruential pseudorandom number generator.

Description

The mcg59 engine is a 59-bit multiplicative congruential generator from NAG Numerical Libraries *NAG*. The mcg59 generator belongs to linear congruential generators with the period length of approximately 2^{57}.

Generation algorithm

$x_n = ax_{n-1}(mod\ m)$

$u_n = x_n/m$

$a = 13^{13}, m = 2^{59}$

class mcg59

Syntax

```
namespace oneapi::mkl::rng {
class mcg59 {
public:
    static constexpr std::uint64_t default_seed = 1;

    mcg59(sycl::queue queue, std::uint64_t seed = default_seed);

    mcg59(const mcg59& other);

    mcg59(mcg59&& other);

    mcg59& operator=(const mcg59& other);

    mcg59& operator=(mcg59&& other);

    ~mcg59();
};
}
```

Class Members

Routine	Description
mcg59(sycl::queue queue, std::uint64_t seed = default_seed)	Constructor for common seed initialization of the engine
mcg59(const mcg59& other)	Copy constructor
mcg59(mcg59&& other)	Move constructor
mcg59& operator=(const mcg59& other)	Copy assignment operator
mcg59& operator=(mcg59&& other)	Move assignment operator

Constructors

```
mcg59::mcg59(sycl::queue queue, std::uint64_t seed = default_seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume $x_0 = seed \ mod \ 2^{59}$, if $x_0 = 0$, assume $x_0 = 1$.

```
mcg59::mcg59(const mcg59& other)
```

Input Parameters

other Valid `mcg59` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mcg59::mcg59(mcg59&& other)
```

Input Parameters

other Valid `mcg59` object. The `queue` and state of the other engine is moved to the current engine.

```
mcg59::mcg59& operator=(const mcg59& other)
```

Input Parameters

other Valid `mcg59` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mcg59::mcg59& operator=(mcg59&& other)
```

Input Parameters

other Valid `mcg59` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

r250

The 32-bit generalized feedback shift register pseudorandom number generator GFSR(250,103) *[Kirkpatrick81]*.

Description

Feedback shift register generators possess ample theoretical foundation and were initially intended for cryptographic and communication applications. The stream state is the array of 250 32-bit integers.

Generation algorithm

$x_n = x_{n-103} \oplus x_{n-250}$

$u_n = x_n / 2^{32}$

class r250

Syntax

```
namespace oneapi::mkl::rng {
class r250 {
public:
    static constexpr std::uint32_t default_seed = 1;

    r250(sycl::queue queue, std::uint32_t seed = default_seed);

    r250(sycl::queue queue, std::vector<std::uint32_t> seed);

    r250(const r250& other);

    r250(r250&& other);

    r250& operator=(const r250& other);

    r250& operator=(r250&& other);

    ~r250();
};
}
```

Class Members

Routine	Description
r250(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine
r250(sycl::queue queue, std::vector<std::uint32_t> seed)	Constructor for extended seed initialization of the engine
r250(const r250& other)	Copy constructor
r250(r250&& other)	Move constructor
r250& operator=(const r250& other)	Copy assignment operator
r250& operator=(r250&& other)	Move assignment operator

Constructors

```
r250::r250(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue Valid sycl::queue object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume $x_{-250} = seed$. If $seed = 0$, assume $seed = 1$. Other values in state are initialized according to recurrent correlation $x_{n+1} = 69069 x_n (mod\ 2^{32})$. Then the values $x_{7k-247}, k = 0, 1, ..., 31$ are interpreted as a binary matrix of size 32 x 32 and diagonal bits are set to 0, the under-diagonal bits to 0.

```
r250::r250(sycl::queue queue, std::vector<std::uint32_t> seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state if $n \geqslant 0 : x_{k-250} = seed[k], k = 0, 1, ..., 249$

```
r250::r250(const r250& other)
```

Input Parameters

other Valid `r250` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
r250::r250(r250&& other)
```

Input Parameters

other Valid `r250` object. The `queue` and state of the other engine is moved to the current engine.

```
r250::r250& operator=(const r250& other)
```

Input Parameters

other Valid `r250` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
r250::r250& operator=(r250&& other)
```

Input Parameters

other Valid `r250` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

wichmann_hill

The wichmann_hill engine is the set of 273 Wichmann-Hill's combined multiplicative congruential generators from NAG Numerical Libraries *[NAG]*.

Description

The set of 372 different basic pseudorandom number generators wichmann_hill is the second basic generator in the NAG libraries.

Generation algorithm

$x_n = a_{1,j} x_{n-1} (mod\ m_{1,j})$

$y_n = a_{2,j} y_{n-1} (mod\ m_{2,j})$

$z_n = a_{3,j} z_{n-1} (mod\ m_{3,j})$

$w_n = a_{4,j} w_{n-1} (mod\ m_{4,j})$

$u_n = (x_n/m_{1,j} + y_n/m_{2,j} + z_n/m_{3,j} + w_n/m_{4,j}) mod\ 1$

The constants $a_{i,j}$ range from 112 to 127, the constants $m_{i,j}$ are prime numbers ranging from 16718909 to 16776917, close to 2^{24}.

class wichmann_hill

Syntax

```
namespace oneapi::mkl::rng {
class wichmann_hill {
public:
    static constexpr std::uint32_t default_seed = 1;

    wichmann_hill(sycl::queue queue, std::uint32_t seed = default_seed);

    wichmann_hill(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx);

    wichmann_hill(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    wichmann_hill(sycl::queue queue, std::initializer_list<std::uint32_t> seed,␣
↪std::uint32_t engine_idx);

    wichmann_hill(const wichmann_hill& other);

    wichmann_hill(wichmann_hill&& other);

    wichmann_hill& operator=(const wichmann_hill& other);

    wichmann_hill& operator=(wichmann_hill&& other);

    ~wichmann_hill();
};
}
```

Class Members

Routine	Description
wichmann_hill(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine (for this case multiple generators of the set would be used)
wichmann_hill(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx)	Constructor for common seed initialization of the engine (for this case single generator of the set would be used)
wichmann_hill(sycl::queue& queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine (for this case multiple generators of the set would be used)
wichmann_hill(sycl::queue& queue, std::initializer_list<std::uint32_t> seed, std::uint32_t engine_idx)	Constructor for extended seed initialization of the engine (for this case single generator of the set would be used)
wichmann_hill(const wichmann_hill& other)	Copy constructor
wichmann_hill(wichmann_hill&& other)	Move constructor
wichmann_hill& operator=(const wichmann_hill& other)	Copy assignment operator
wichmann_hill& operator=(wichmann_hill&& other)	Move assignment operator

Constructors

```
wichmann_hill::wichmann_hill(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. Assume $x_0 = seed \ mod \ m_1, y_0 = z_0 = w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.

```
wichmann_hill::wichmann_hill(sycl::queue queue, std::uint32_t seed, std::uint32_t␣
↪engine_idx)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. Assume $x_0 = seed \ mod \ m_1, y_0 = z_0 = w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.

engine_idx The index of the set $1, \ldots, 273$.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $idx > 273$

```
wichmann_hill::wichmann_hill(sycl::queue& queue, std::initializer_list<std::uint32_t>␣
↪seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume: if $n = 0 : x_0 = y_0 = z_0 = w_0 = 1$

if $n = 1 : x_0 = seed[0] \ mod \ m_1, y_0 = z_0 = w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.

if $n = 2 : x_0 = seed[0] \ mod \ m_1, y_0 = seed[1] \ mod \ m_2, z_0 = w_0 = 1$.

if $n = 3 : x_0 = seed[0] \ mod \ m_1, y_0 = seed[1] \ mod \ m_2, z_0 = seed[2] \ mod \ m_3, w_0 = 1$.

if $n \geqslant 4 : x_0 = seed[0] \ mod \ m_1, y_0 = seed[1] \ mod \ m_2$

$z_0 = seed[2] \ mod \ m_3, w_0 = seed[3] \ mod \ m_4$.

```
wichmann_hill::wichmann_hill(sycl::queue& queue, std::initializer_list<std::uint32_t>␣
↪seed, std::uint32_t engine_idx)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume: if $n = 0 : x_0 = y_0 = z_0 = w_0 = 1$

if $n = 1 : x_0 = seed[0] \ mod \ m_1, y_0 = z_0 = w_0 = 1$. If $x_0 = 0$, assume $x_0 = 1$.

if $n = 2 : x_0 = seed[0] \ mod \ m_1, y_0 = seed[1] \ mod \ m_2, z_0 = w_0 = 1$.

if $n = 3 : x_0 = seed[0] \ mod \ m_1, y_0 = seed[1] \ mod \ m_2, z_0 = seed[2] \ mod \ m_3, w_0 = 1$.

if $n \geqslant 4 : x_0 = seed[0] \ mod \ m_1, y_0 = seed[1] \ mod \ m_2$

$z_0 = seed[2] \ mod \ m_3, w_0 = seed[3] \ mod \ m_4$.

engine_idx The index of the set 1, . . . , 273.

```
wichmann_hill::wichmann_hill(const wichmann_hill& other)
```

Input Parameters

other Valid `wichmann_hill` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
wichmann_hill::wichmann_hill(wichmann_hill&& other)
```

Input Parameters

other Valid `wichmann_hill` object. The `queue` and state of the other engine is moved to the current engine.

```
wichmann_hill::wichmann_hill& operator=(const wichmann_hill& other)
```

Input Parameters

other Valid `wichmann_hill` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
wichmann_hill::wichmann_hill& operator=(wichmann_hill&& other)
```

Input Parameters

other Valid `wichmann_hill` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

mt19937

Mersenne Twister pseudorandom number generator.

Description

The Mersenne Twister pseudorandom number generator, mt19937, is a modification of twisted generalized feedback shift register generator [*Matsumoto98*]. MT19937 has the period length of $2^{19937} - 1$ and is 623-dimensionally equidistributed with up to 32-bit accuracy. These properties make the generator applicable for simulations in various fields of science and engineering. The state of the generator is represented by 624 32-bit unsigned integer numbers.

Generation algorithm

$x_n = x_{n-(624-397)} \oplus ((x_{n-624} \& 0x80000000) | (x_{n-624+1} \& 0x7FFFFFFF))A$

$y_n = x_n$

$y_n = y_n \oplus (y_n >> 11)$

$y_n = y_n \oplus ((y_n << 7) \& 0x9D2C5680)$

$y_n = y_n \oplus ((y_n << 15) \& 0xEFC60000)$

$y_n = y_n \oplus (y_n >> 18)$

$u_n = y_n / 2^{32}$

Matrix $A_j (32x32)$ has the following format:

$$\begin{bmatrix} 0 & 1 & 0 & ... & ... \\ 0 & 0 & ... & 0 & ... \\ ... & ... & ... & ... & ... \\ 0 & ... & 0 & 0 & 1 \\ a_{31} & a_{30} & ... & ... & a_0 \end{bmatrix}$$

Where the 32-bit vector $a = a_{31}..a_0$ has the value $a = 0x9908B0DF$.

class mt19937

Syntax

```
namespace oneapi::mkl::rng {
class mt19937 {
public:
    static constexpr std::uint32_t default_seed = 1;

    mt19937(sycl::queue queue, std::uint32_t seed = default_seed);

    mt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    mt19937(const mt19937& other);

    mt19937(mt19937&& other);

    mt19937& operator=(const mt19937& other);
```

(continues on next page)

(continued from previous page)

```
    mt19937& operator=(mt19937&& other);

    ~mt19937();
};
}
```

Class Members

Routine	Description
mt19937(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine
mt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine
mt19937(const mt19937& other)	Copy constructor
mt19937(mt19937&& other)	Move constructor
mt19937& operator=(const mt19937& other)	Copy assignment operator
mt19937& operator=(mt19937&& other)	Move assignment operator

Constructors

```
mt19937::mt19937(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

```
mt19937::mt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

```
mt19937::mt19937(const mt19937& other)
```

Input Parameters

other Valid `mt19937` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mt19937::mt19937(mt19937&& other)
```

Input Parameters

other Valid `mt19937` object. The `queue` and state of the other engine is moved to the current engine.

```
mt19937::mt19937& operator=(const mt19937& other)
```

Input Parameters

other Valid `mt19937` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mt19937::mt19937& operator=(mt19937&& other)
```

Input Parameters

other Valid `mt19937` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

sfmt19937

The SIMD-oriented Mersenne Twister pseudorandom number generator.

Description

SIMD-oriented Fast Mersenne Twister pseudorandom number generator SFMT19937 [*Saito08*] with a period length equal to $2^{19937} - 1$ of the produced sequence. The state of the engine contains the array of 156 128-bit integers.

Generation algorithm

$w_n = w_0 A \oplus w_M B \oplus w_{n-2} C \oplus w_{n-1} D$

Where $w_0, w_M, w_{n-2}, ...$ are the 128-bit integers, and wA, wB, wC, wD operations are defined as follows:

$wA = (w << 8) \oplus w$, left shift of 128-bit integer w by a followed by exclusive-or operation

$wB = (w >> 8) \& mask$, right shift of each 32-bit integer in quadruple w by and-operator with quadruple of 32-bit masks $mask = (0xBFFFFFF6, 0xDFFAFFFF, 0xDDFECB7F, 0xDFFFFFEF)$

$wC = (w >> 8) \oplus w$, right shift of 128-bit integer w

$wD = (w << 8)$, left shift of each 32-bit integer in quadruple w

Integer output: $r_{4n+k} = w_n(k)$, where $w_n(k)$ is the k-th 32-bit integer in quadruple $w_n, k = 0, 1, 2, 3$

$u_n = (int)r_n/2^{32} + 1/2$

class sfmt19937

Syntax

```
namespace oneapi::mkl::rng {
class sfmt19937 {
public:
    static constexpr std::uint32_t default_seed = 1;

    sfmt19937(sycl::queue queue, std::uint32_t seed = default_seed);

    sfmt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    sfmt19937(const sfmt19937& other);

    sfmt19937(sfmt19937&& other);

    sfmt19937& operator=(const sfmt19937& other);

    sfmt19937& operator=(sfmt19937&& other);

    ~sfmt19937();
};
}
```

Class Members

Routine	Description
sfmt19937(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine
sfmt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine
sfmt19937(const sfmt19937& other)	Copy constructor
sfmt19937(sfmt19937&& other)	Move constructor
sfmt19937& operator=(const sfmt19937& other)	Copy assignment operator
sfmt19937& operator=(sfmt19937&& other)	Move assignment operator

Constructors

```
sfmt19937::sfmt19937(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. The initialization algorithm described in [*Saito08*].

```
sfmt19937::sfmt19937(sycl::queue queue, std::initializer_list<std::uint32_t> seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. The initialization algorithm described in [*Saito08*].

```
sfmt19937::sfmt19937(const sfmt19937& other)
```

Input Parameters

other Valid `sfmt19937` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
sfmt19937::sfmt19937(sfmt19937&& other)
```

Input Parameters

other Valid `sfmt19937` object. The `queue` and state of the other engine is moved to the current engine.

```
sfmt19937::sfmt19937& operator=(const sfmt19937& other)
```

Input Parameters

other Valid `sfmt19937` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
sfmt19937::sfmt19937& operator=(sfmt19937&& other)
```

Input Parameters

other Valid `sfmt19937` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

mt2203

The mt2203 engine is the set of 6024 Mersenne Twister pseudorandom number generators MT2203 [*Matsumoto98*], [*Matsumoto00*].

Description

The set of 6024 basic pseudorandom number generators MT2203 is a natural addition to the MT19937 generator. MT2203 generators are intended for use in large scale Monte Carlo simulations performed on multi-processor computer systems.

Generation algorithm

For $j = 1, ..., 6024$:

$x_{n,j} = x_{n-(69-34),j} \oplus ((x_{n-69,j}\&0xFFFFFFE0)|(x_{n+69+1,j}\&0x1F))A_j$

$y_{n,j} = x_{n,j}$

$y_{n,j} = y_{n,j} \oplus (y_{n,j} >> 12)$

$y_{n,j} = y_{n,j} \oplus ((y_{n,j} << 7)\&b_j)$

$y_{n,j} = y_{n,j} \oplus ((y_{n,j} << 15)\&c_j)$

$y_{n,j} = y_{n,j} \oplus (y_{n,j} >> 18)$

Matrix $A_j(32x32)$ has the following format:

$$\begin{bmatrix} 0 & 1 & 0 & ... & ... \\ 0 & 0 & ... & 0 & ... \\ ... & ... & ... & ... & ... \\ 0 & ... & 0 & 0 & 1 \\ a_{31,j} & a_{30,j} & ... & ... & a_{0,1} \end{bmatrix}$$

class mt2203

Syntax

```
namespace oneapi::mkl::rng {
class mt2203 {
public:
    static constexpr std::uint32_t default_seed = 1;

    mt2203(sycl::queue queue, std::uint32_t seed = default_seed);

    mt2203(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx);

    mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed);

    mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed, std::uint32_
↪t engine_idx);

    mt2203(const mt2203& other);
```

(continues on next page)

(continued from previous page)

```
    mt2203(mt2203&& other);

    mt2203& operator=(const mt2203& other);

    mt2203& operator=(mt2203&& other);

    ~mt2203();
};
}
```

Class Members

Routine	Description
mt2203(sycl::queue queue, std::uint32_t seed = default_seed)	Constructor for common seed initialization of the engine (for this case multiple generators of the set would be used)
mt2203(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx)	Constructor for common seed initialization of the engine (for this case single generator of the set would be used)
mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed)	Constructor for extended seed initialization of the engine (for this case multiple generators of the set would be used)
mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed, std::uint32_t engine_idx)	Constructor for extended seed initialization of the engine (for this case single generator of the set would be used)
mt2203(const mt2203& other)	Copy constructor
mt2203(mt2203&& other)	Move constructor
mt2203& operator=(const mt2203& other)	Copy assignment operator
mt2203& operator=(mt2203&& other)	Move assignment operator

Constructors

```
mt2203::mt2203(sycl::queue queue, std::uint32_t seed = default_seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

```
mt2203::mt2203(sycl::queue queue, std::uint32_t seed, std::uint32_t engine_idx)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

engine_idx The index of the set 1, . . . , 6024.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $idx > 6024$

```
mt2203::mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

```
mt2203::mt2203(sycl::queue queue, std::initializer_list<std::uint32_t> seed,␣
↪std::uint32_t engine_idx)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state. The initialization algorithm described in [*MT2203*].

engine_idx The index of the set 1, . . . , 6024.

```
mt2203::mt2203(const mt2203& other)
```

Input Parameters

other Valid `mt2203` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mt2203::mt2203(mt2203&& other)
```

Input Parameters

other Valid `mt2203` object. The `queue` and state of the other engine is moved to the current engine.

```
mt2203::mt2203& operator=(const mt2203& other)
```

Input Parameters

other Valid `mt2203` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
mt2203::mt2203& operator=(mt2203&& other)
```

Input Parameters

other Valid `mt2203` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

ars5

The ars5 counter-based pseudorandom number generator.

Description

The ars5 engine is a keyed family of counter-based BRNG. The state consists of a 128-bit integer counter c and a 128-bit key k. The BRNG is based on the AES encryption algorithm [*FIPS-197*].

Generation algorithm

The generator has a 32-bit integer output obtained in the following way [*Salmon11*]:

1. The i-th number is defined by the following formula $r_i = (f(i/4) >> ((i \ mod \ 4) * 32)) \ mod \ 2^{32}$
2. **Function $f(c)$ takes a 128-bit argument and returns a 128-bit number. The returned number is obtained as follows:**
   2.1. $c_0 = c \oplus k$ and $k_0 = k$.

   2.2. The following recurrence is calculated N = 5 times:

   $c_{i+1} = SubBytes(c)$

   $c_{i+1} = ShiftRows(c_{i+1})$

   $c_{i+1} = MixColumns(c_{i+1})$, this step is omitted if $i + 1 = N$

   $c_{i+1} = AddRoundKey(c_{i+1}, k_j)$

   $Lo(k_{i+1}) = Lo(k) + 0x9E3779B97F4A7C15$

   $Hi(k_{i+1}) = Hi(k) + 0xBB67AE8584CAA73B$

   Specification for $SubBytes, ShiftRows, MixColumns, AddRoundKey$ functions can be found in [*FIPS-197*].

   2.3. Put $f(c) = c_N$, where $N = 10$
3. Real output: $u_n = (int)r_n/2^{32} + 1/2$

class ars5

Syntax

```
namespace oneapi::mkl::rng {
class ars5 {
public:
        static constexpr std::uint64_t default_seed = 0;

        ars5(sycl::queue queue, std::uint64_t seed = default_seed);

        ars5(sycl::queue queue, std::initializer_list<std::uint64_t> seed);

        ars5(const ars5& other);

        ars5(ars5&& other);

        ars5& operator=(const ars5& other);

        ars5& operator=(ars5&& other);

        ~ars5();
};
}
```

Class Members

Routine	Description
ars5(sycl::queue queue, std::uint64_t seed)	Constructor for common seed initialization of the engine
ars5(sycl::queue queue, std::initializer_list<std::uint64_t> seed)	Constructor for extended seed initialization of the engine
ars5(const ars5& other)	Copy constructor
ars5(ars5&& other)	Move constructor
ars5& operator=(const ars5& other)	Copy assignment operator
ars5& operator=(ars5&& other)	Move assignment operator

Constructors

```
ars5::ars5(sycl::queue queue, std::uint64_t seed)
```

Input Parameters

queue Valid sycl::queue object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume $k = seed, c = 0$, where k is 128-bit key, c is 128-bit counter.

```
ars5::ars5(sycl::queue queue, std::initializer_list<std::uint64_t> seed)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

seed The initial conditions of the generator state, assume if $n = 0 : k = 0, c = 0$

if $n = 1 : k = seed[0], c = 0$

if $n = 2 : k = seed[0] + seed[1] \cdot 2^{64}, c = 0$

if $n = 3 : k = seed[0] + seed[1] \cdot 2^{64}, c = seed[2]$

if $n = 4 : k = seed[0] + seed[1] \cdot 2^{64}, c = seed[2] + seed[3] \cdot 2^{64}$

for $n > 4$ following arguments are ignored

```
ars5::ars5(const ars5& other)
```

Input Parameters

other Valid `ars5` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
ars5::ars5(ars5&& other)
```

Input Parameters

other Valid `ars5` r-value object. The `queue` and state of the other engine is moved to the current engine.

```
ars5::ars5& operator=(const ars5& other)
```

Input Parameters

other Valid `ars5` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
ars5::ars5& operator=(ars5&& other)
```

Input Parameters

other Valid `ars5` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

sobol

The sobol is a 32-bit Gray code-based quasi-random number generator.

Description

Bratley and Fox *[Bratley88]* provide an implementation of the SOBOL quasi-random number generator. The default dimensions of quasi-random vectors can vary from 1 to 40 inclusive. It is also allowed to register user-defined parameters (direction numbers).

Generation algorithm

$x_n = x_{n_1} \oplus v_c$

$u_n = x_n / 2^{32}$

The value c is the right-most zero bit in $n-1$; x_n is s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during engine initialization) $v_i, i = 1, 32$ are called direction numbers. The vector u_n is the generator output normalized to the unit hypercube $(0, 1)^s$.

class sobol

Syntax

```
namespace oneapi::mkl::rng {
class sobol {
public:
    static constexpr std::uint32_t default_dimensions_number = 1;

    sobol(sycl::queue queue, std::uint32_t dimensions = default_dimensions_number);

    sobol(sycl::queue queue, std::vector<std::uint32_t>& direction_numbers);

    sobol(const sobol& other);

    sobol(sobol&& other);

    sobol& operator=(const sobol& other);

    sobol& operator=(sobol&& other);

    ~sobol();
};
}
```

Class Members

Routine	Description
sobol(sycl::queue queue, std::uint32_t dimensions = default_dimensions_number)	Constructor with specified number of dimensions. The value should be 1..40.
sobol(sycl::queue queue, std::vector<std::uint32_t>& direction_numbers)	Constructor for extended use-case, when it's needed to use the number of dimensions greater than 40 or obtain another sequence.
sobol(const sobol& other)	Copy constructor
sobol(sobol&& other)	Move constructor
sobol& operator=(const sobol& other)	Copy assignment operator
sobol& operator=(sobol&& other)	Move assignment operator

Constructors

```
sobol::sobol(sycl::queue queue, std::uint32_t dimensions = default_dimensions_number)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

dimensions Number of dimensions. If $dimen < 1$ or $dimen > 40$, assume $dimen = 1$.

```
sobol::sobol(sycl::queue queue, std::vector<std::uint32_t>& direction_numbers)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

direction_numbers If you want to generate quasi-random vectors of greater dimension or obtain another sequence, you can register a set of your own direction_numbers. The number of dimensions corresponds to direction_numbers.size() / 32.

```
sobol::sobol(const sobol& other)
```

Input Parameters

other Valid `sobol` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
sobol::sobol(sobol&& other)
```

Input Parameters

other Valid `sobol` object. The `queue` and state of the other engine is moved to the current engine.

```
sobol::sobol& operator=(const sobol& other)
```

Input Parameters

other Valid `sobol` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
sobol::sobol& operator=(sobol&& other)
```

Input Parameters

other Valid `sobol` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

niederreiter

The niederreiter generator is a 32-bit Gray code-based quasi-random number generator.

Description

According to results of Bratley, Fox and Niederreiter *[Bratley92]* Niederreiter sequences have the best known theoretical asymptotic properties. The default dimension of quasi-random vectors can vary from 1 to 318 inclusive. It is also allowed to register user-defined parameters (irreducible polynomials).

Generation algorithm

$x_n = x_{n_1} \oplus v_c$

$u_n = x_n / 2^{32}$

The value c is the right-most zero bit in $n-1$; x_n is s-dimensional vector of 32-bit values. The s-dimensional vectors (calculated during engine initialization) $v_i, i = 1, 32$ are called direction numbers. The vector u_n is the generator output normalized to the unit hypercube $(0, 1)^s$.

class niederreiter

Syntax

```
namespace oneapi::mkl::rng {
class niederreiter {
public:
    static constexpr std::uint32_t default_dimensions_number = 1;

    niederreiter(sycl::queue queue, std::uint32_t dimensions = default_dimensions_
↪number);

    niederreiter(sycl::queue queue, std::vector<std::uint32_t>& irred_polynomials);

    niederreiter(const niederreiter& other);

    niederreiter(niederreiter&& other);

    niederreiter& operator=(const niederreiter& other);

    niederreiter& operator=(niederreiter&& other);

    ~niederreiter();
};
}
```

Class Members

Routine	Description
niederreiter(sycl::queue queue, std::uint32_t dimensions = default_dimensions_number)	Constructor with specified number of dimensions. The value should be 1..318.
niederreiter(sycl::queue queue, std::vector<std::uint32_t>& irred_polynomials)	Constructor for extended use-case, when it's needed to use the number of dimensions greater than 318 or obtain another sequence.
niederreiter(const niederreiter& other)	Copy constructor
niederreiter(niederreiter&& other)	Move constructor
niederreiter& operator=(const niederreiter& other)	Copy assignment operator
niederreiter& operator=(niederreiter&& other)	Move assignment operator

Constructors

```
niederreiter::niederreiter(sycl::queue queue, std::uint32_t dimensions = default_
↪dimensions_number)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

dimensions Number of dimensions. If $dimen < 1$ or $dimen > 318$, assume $dimen = 1$.

```
niederreiter::niederreiter(sycl::queue queue, std::vector<std::uint32_t>& irred_
↪polynomials)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

irred_polynomials If you want to generate quasi-random vectors of greater dimension or obtain another sequence, you can register a set of your own irreducible polynomials. The number of dimensions corresponds to the length of the vector.

```
niederreiter::niederreiter(const niederreiter& other)
```

Input Parameters

other Valid `niederreiter` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
niederreiter::niederreiter(niederreiter&& other)
```

Input Parameters

other Valid `niederreiter` object. The `queue` and state of the other engine is moved to the current engine.

```
niederreiter::niederreiter& operator=(const niederreiter& other)
```

Input Parameters

other Valid `niederreiter` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
niederreiter::niederreiter& operator=(niederreiter&& other)
```

Input Parameters

other Valid `niederreiter` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

nondeterministic

Non-deterministic random number generator.

Description

Implementation defined generator with non-deterministic source of randomness (for example, a hardware device).

class nondeterministic

Syntax

```
namespace oneapi::mkl::rng {
class nondeterministic {
public:
    nondeterministic(sycl::queue queue);

    nondeterministic(const nondeterministic& other);

    nondeterministic(nondeterministic&& other);

    nondeterministic& operator=(const nondeterministic& other);

    nondeterministic& operator=(nondeterministic&& other);

    ~nondeterministic();
};
}
```

Class Members

Routine	Description
nondeterministic(sycl::queue queue)	Constructor for the particular device
nondeterministic(const nondeterministic& other)	Copy constructor
nondeterministic(nondeterministic&& other)	Move constructor
nondeterministic& operator=(const nondeterministic& other)	Copy assignment operator
nondeterministic& operator=(nondeterministic&& other)	Move assignment operator

Constructors

```
nondeterministic::nondeterministic(sycl::queue queue)
```

Input Parameters

queue Valid `sycl::queue` object, calls of the *oneapi::mkl::rng::generate()* routine submits kernels in this queue to obtain random numbers from a given engine.

```
nondeterministic::nondeterministic(const nondeterministic& other)
```

Input Parameters

other Valid `nondeterministic` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
nondeterministic::nondeterministic(nondeterministic&& other)
```

Input Parameters

other Valid `nondeterministic` object. The `queue` and state of the other engine is moved to the current engine.

```
nondeterministic::nondeterministic& operator=(const nondeterministic& other)
```

Input Parameters

other Valid `nondeterministic` object. The `queue` and state of the other engine is copied and applied to the current engine.

```
nondeterministic::nondeterministic& operator=(nondeterministic&& other)
```

Input Parameters

other Valid `nondeterministic` r-value object. The `queue` and state of the other engine is moved to the current engine.

Parent topic: *Engines (Basic Random Number Generators)*

Service Routines

Routine	Description
leapfrog	Proceed state of engine by the leapfrog method to generate a subsequence of the original sequence
skip_ahead	Proceed state of engine by the skip-ahead method to skip a given number of elements from the original sequence

Parent topic: *Random Number Generators*

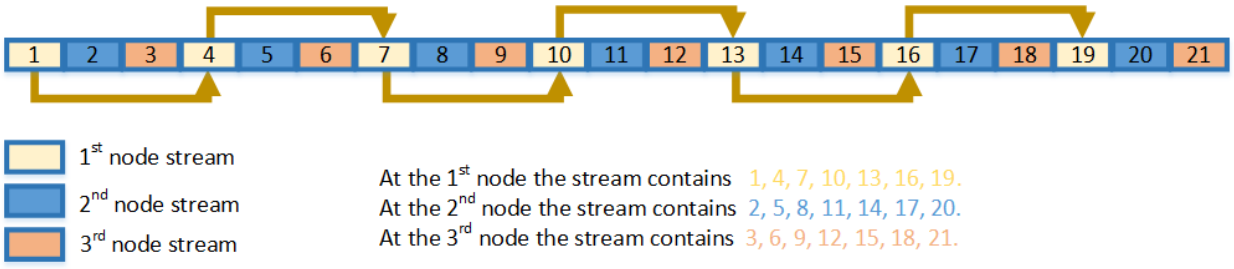
leapfrog

Proceed state of engine by the leapfrog method.

Description and Assumptions

oneapi::mkl::rng::leapfrog function generates random numbers in an engine with non-unit stride. This feature is particularly useful in distributing random numbers from the original stream across the stride buffers without generating the original random sequence with subsequent manual distribution. see *Figure "Leapfrog Method"*.

Leapfrog Method



leapfrog

Syntax

```
namespace oneapi::mkl::rng {
template<typename EngineType> \
                void oneapi::mkl::rng::leapfrog(EngineType& engine, std::uint64_t idx,␣
↪std::uint64_t stride);
}
```

Template Parameters

EngineType Type of engine. Note: may not be supported by all available engine classes.

Input Parameters

engine Engine which state would be skipped.

idx Index of the computational node.

stride Largest number of computational nodes, or stride.

Example

```
// Creating 3 identical engines
mkl::rng::mcg31m1 engine_1(queue, seed);

mkl::rng::mcg31m1 engine_2(queue, engine_1);
mkl::rng::mcg31m1 engine_3(queue, engine_1);


// Leapfrogging the states of engines
mkl::rng::leapfrog(engine_1, 0 , 3);
mkl::rng::leapfrog(engine_2, 1 , 3);
mkl::rng::leapfrog(engine_3, 2 , 3);
// Generating random numbers
```

Parent topic: *Service Routines*

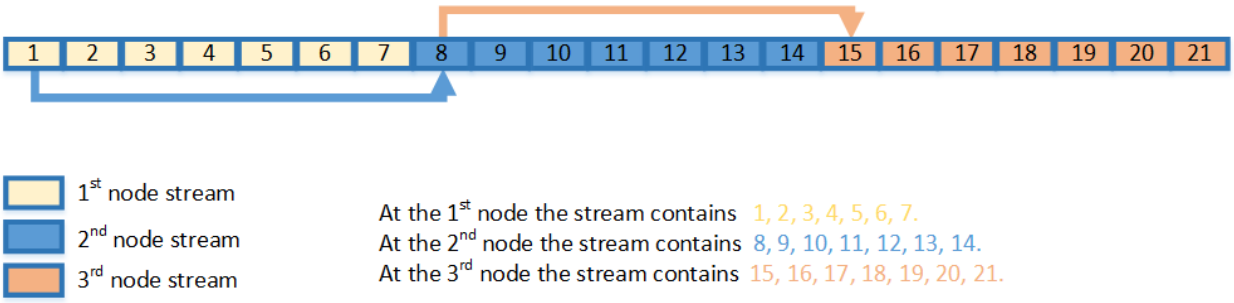
skip_ahead

Proceed state of engine by the skip-ahead method.

Description and Assumptions

oneapi::mkl::rng::skip_ahead function changes the current state of the engine so that with the further call of the generator the output subsequence begins with the specified offset see *Figure "Block-Splitting Method"*.

Block-Splitting Method



skip_ahead

Syntax

```
namespace oneapi::mkl::rng {
template<typename EngineType> \
                void oneapi::mkl::rng::skip_ahead(EngineType& engine, std::uint64_t␣
↪num_to_skip);
}
```

Template Parameters

EngineType Type of engine. Note: may not be supported by all available engine classes.

Input Parameters

engine Engine which state would be skipped.

num_to_skip Number of elements to skip in the engine's sequence.

Example

```
// Creating 3 identical engines
oneapi::mkl::rng::mcg31m1 engine_1(queue, seed);
oneapi::mkl::rng::mcg31m1 engine_2(queue, engine_1);
oneapi::mkl::rng::mcg31m1 engine_3(queue, engine_2);

// Skipping ahead by 7 elements the 2nd engine
oneapi::mkl::rng::skip_ahead(engine_2, 7);

// Skipping ahead by 14 elements the 3rd engine
oneapi::mkl::rng::skip_ahead(engine_3, 14);
```

skip_ahead (Interface with a partitioned number of skipped elements)

Syntax

```
namespace oneapi::mkl::rng {
template<typename EngineType> \
               void oneapi::mkl::rng::skip_ahead(EngineType& engine, std::initializer_
↪list<std::uint64_t> num_to_skip);
}
```

Template Parameters

EngineType Type of engine. Note: may not be supported by all available engine classes.

Input Parameters

engine Engine which state would be skipped.

num_to_skip Partitioned number of elements to skip in the engine's sequence. The total number of skipped elements would be: $num\_to\_skip[0] + num\_to\_skip[1] \cdot 2^{64} + ... + num\_to\_skip[1] \cdot 2^{64(n-1)}$, where n is a number of elements in num_to_skip list.

Example with Partitioned Numer of Elements

```
// Creating the first engine
oneapi::mkl::rng::mrg32k3a engine_1(queue, seed);

// To skip 2^64 elements in the random stream number of skipped elements should be
/represented as num_to_skip = 2^64 = 0 + 1 * 2^64
std::initializer_list<std::uint64_t> num_to_skip = {0, 1};

// Creating the 2nd engine based on 1st. Skipping by 2^64
oneapi::mkl::rng::mrg32k3a engine_2(queue, engine_1);
oneapi::mkl::rng::skip_ahead(engine_2, num_to_skip);
```

Parent topic: *Service Routines*

Distributions

oneMKL RNG routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence and the explanation of input and output parameters. *Table Continuous Distribution Generators* and *Table Discrete Distribution Generators* list the random number generator routines with data types and output distributions, and sets correspondence between data types of the generator routines and the basic random number generators.

Table Continuous Distribution Generators

Routine	Description
uniform (continuous)	Uniform continuous distribution on the interval [a,b)
gaussian	Normal (Gaussian) distribution
exponential	Exponential distribution
laplace	Laplace distribution (double exponential distribution)
weibull	Weibull distribution
cauchy	Cauchy distribution
rayleigh	Rayleigh distribution
lognormal	Lognormal distribution
gumbel	Gumbel (extreme value) distribution
gamma	Gamma distribution
beta	Beta distribution
chi_square	Chi-Square distribution
gaussian_mv	Normal Multivariate (Gaussian Multivariate) distribution

Table Discrete Distribution Generators

Type of Distribution	Description
uniform (discrete)	Uniform discrete distribution on the interval [a,b)
uniform_bits	Uniformly distributed bits in 32/64-bit chunks
bits	Bits of underlying BRNG integer recurrence
bernoulli	Bernoulli distribution
geometric	Geometric distribution
binomial	Binomial distribution
hypergeometric	Hypergeometric distribution
poisson	Poisson distribution
poisson_v	Poisson distribution with varying mean
negative_binomial	Negative binomial distribution, or Pascal distribution
multinomial	Multinomial distribution

Modes of random number generation

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within the definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval [a,b] belong to this interval irrespective of what a and b values may be. Fast mode provides high performance generation and also guarantees that generated random numbers belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying the relevant value of the method parameter in generator routines. The list of distributions that support accurate mode of generation is given in the table below.

Table Distribution Generators with Accurate Method

Distribution	Method
uniform (continuous)	*oneapi::mkl::rng::unform_method::standard_accurate*
exponential	*oneapi::mkl::rng::exponential_method::icdf_accurate*
weibull	*oneapi::mkl::rng::weibull_method::icdf_accurate*
rayleigh	*oneapi::mkl::rng::rayleigh_method::icdf_accurate*
lognormal	*oneapi::mkl::rng::lognormal_method::box_muller2_accurate*, *oneapi::mkl::rng::lognormal_method::icdf_accurate*
gamma	*oneapi::mkl::rng::gamma_method::marsaglia_accurate*
beta	*oneapi::mkl::rng::beta_method::cja_accurate*

Parent topic: *Random Number Generators*

Distributions Template Parameter Method

Method	Distributions	Math Description
uniform_method::standard uniform_method::standard_accurate	uniform uniform(i)	Standard method. uniform_method::standard_accurate supported for uniform(i) only.
gaussian_method::box_muller2	gaussian	Generates normally distributed random number x thru the pair of uniformly distributed numbers u_1 and u_2 according to the formula: $x = \sqrt{-2lnu_1}\sin(2\pi u_2)$
gaussian_method::box_muller2	gaussian lognormal	Generates normally distributed random numbers x_1 and x_2 thru the pair of uniformly distributed numbers u_1 and u_2 according to the formulas: $x_1 = \sqrt{-2lnu_1}\sin 2\pi u_2$ $x_2 = \sqrt{-2lnu_1}\cos 2\pi u_2$
gaussian_method::icdf	gaussian	Inverse cumulative distribution function (ICDF) method.
exponential_method::icdf exponential_method::icdf_accurate	exponential	Inverse cumulative distribution function (ICDF) method.
weibull_method::icdf weibull_method::icdf_accurate	weibull	Inverse cumulative distribution function (ICDF) method.
cauchy_method::icdf	cauchy	Inverse cumulative distribution function (ICDF) method.
rayleigh_method::icdf rayleigh_method::icdf_accurate	rayleigh	Inverse cumulative distribution function (ICDF) method.
bernoulli_method::icdf	bernoulli	Inverse cumulative distribution function (ICDF) method.
geometric_method::icdf	geometric	Inverse cumulative distribution function (ICDF) method.
gumbel_method::icdf	gumbel	Inverse cumulative distribution function (ICDF) method.
lognormal_method::icdf lognormal_method::icdf_accurate	lognormal	Inverse cumulative distribution function (ICDF) method.
lognormal_method::box_muller2 lognormal_method::box_muller2_accurate	lognormal	Generated normally distributed random numbers x_1 and x_2 by box_muller2 method are converted to lognormal distribution.
gamma_method::mt_gaa gamma_method::marsaglia	gamma	For $\alpha > 1$, a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \leq \alpha < 1$, a gamma distributed random number is generated using rejection from Weibull distribution; for $\alpha < 0.6$, a gamma distributed random number is obtained using transformation of exponential power distribution; for $\alpha = 1$, gamma distribution is reduced to exponential distribution.
beta_method::cja beta_method::cja_accurate	beta	Cheng-Jonhnk-Atkinson method. For $min(p,q) > 1$, Cheng method is used; for $min(p,q) < 1$, Johnk method is used, if $q + K * p2 + C \leq 0(K = 0.852..., C = -0.956...)$ otherwise, Atkinson switching algorithm is used; for $max(p,q) < 1$, method of Johnk is used; for $min(p,q) < 1, max(p,q) > 1$, Atkinson switching algorithm is used (CJA stands for Cheng, Johnk, Atkinson); for $p = 1$ or $q = 1$, inverse cumulative distribution function method is used; for $p = 1$ and $q = 1$, beta distribution is reduced to uniform distribution.
chi_square_method::gamma_based	chi_square	(gamma_based): If $\nu \geq 17$ or ν is odd and $5 \leq \nu \leq 15$, a chi-square distribution is reduced to a Gamma distribution with these parameters: Shape $\alpha = \nu/2$Offset $a = 0$ Scale factor $\beta = 2$ The random numbers of the Gamma distribution are generated.
binomial_method::btpe	binomial	Acceptance/rejection method for $ntrial * min(p, 1-p) \geq 30$ with decomposition into four regions: two parallelograms, triangle, left exponential tail, right exponential tail.
poisson_method::pspace	poisson	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into four regions: two parallelograms, triangle, left exponential tail, right exponential tail.
poisson_method::gaussian_icdf	poisson	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.
poisson_v_method::gaussian_icdf	poisson_v	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.
hypergeometric_method::h2pe	hypergeometric	Acceptance/rejection method for large mode of distribution with decomposition into three regions: rectangular, left exponential tail, right exponential tail.
negative_binomial_method::nbar	negative_binomial	Acceptance/rejection method for: $\frac{(a-1)(1-p)}{p} \geq 100$ with decomposition into five regions: rectangular, 2 trapezoid, left exponential tail, right exponential tail.
multinomial_method::poisson_approx	multinomial	Multinomial distribution with parameters m, k, and a probability vector p. Random numbers of the multinomial distribution are generated by Poisson Approximation

Parent topic: *Distributions*

uniform (continuous)

Class is used for generation of uniformly distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers uniformly distributed over the interval $[a, b)$, where a, b are the left and right bounds of the interval, respectively, and $a, b \in R; a < b$

The probability distribution is given by:

$$f_{a,b}(x) = \begin{cases} \frac{1}{b-a}, x \in [a, b) \\ 0, x \notin [a, b) \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, x < a \\ \frac{x-a}{b-a}, a \leq x < b, x \in R \\ 1, x \geq b \end{cases}$$

class uniform

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = uniform_method::by_default>
class uniform {
public:
    using method_type = Method;
    using result_type = RealType;
    uniform();
    explicit uniform(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::uniform_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::uniform_method::by_default`
- `oneapi::mkl::rng::uniform_method::standard`

- `oneapi::mkl::rng::uniform_method::standard_accurate`

See description of the methods in *Distributions methods template parameter*

Class Members

Routine	Description
uniform()	Default constructor
explicit uniform(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain left bound a
RealType b() const	Method to obtain right bound b

Member types

```
uniform::method_type = Method
```

Description

The type which defines transformation method for generation.

```
uniform::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
uniform::uniform()
```

Description

Default constructor for distribution, parameters set as $a = 0.0$, $b = 1.0$.

```
explicit uniform::uniform(RealType a, RealType b)
```

Description

Constructor with parameters. a is a left bound, b is a right bound, assume $a < b$.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $a \geq b$

Characteristics

```
RealType uniform::a() const
```

Return Value

Returns the distribution parameter a - left bound.

```
RealType uniform::b() const
```

Return Value

Returns the distribution parameter b - right bound.

Parent topic: *Distributions*

gaussian

Class is used for generation of normally distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers normally distributed with mean $(mean, a)$ and standard deviation $(stddev, \sigma)$, where $a, \sigma \in R; \sigma > 0$.

The probability distribution is given by:

$$f_{a,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} exp(-\frac{(x-a)^2}{2*\sigma^2}), x \in R.$$

The cumulative distribution function is as follows:

$$F_{a,\sigma}(x) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} exp(-\frac{(y-a)^2}{2*\sigma^2}) dy, x \in R.$$

class gaussian

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = gaussian_method::by_default>
class gaussian {
public:
    using method_type = Method;
    using result_type = RealType;
```

(continues on next page)

(continued from previous page)

```
    gaussian();
    explicit gaussian(RealType mean, RealType stddev);
    RealType mean() const;
    RealType stddev() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::gaussian_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::gaussian_method::by_default`
- `oneapi::mkl::rng::gaussian_method::box_muller`
- `oneapi::mkl::rng::gaussian_method::box_muller2`
- `oneapi::mkl::rng::gaussian_method::icdf`

See description of the methods in *Distributions methods template parameter*

Class Members

Routine	Description
gaussian()	Default constructor
explicit gaussian(RealType mean, RealType stddev)	Constructor with parameters
RealType mean() const	Method to obtain mean value
RealType stddev() const	Method to obtain standard deviation value

Member types

```
gaussian::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gaussian::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
gaussian::gaussian()
```

Description

Default constructor for distribution, parameters set as *mean* = 0.0, *stddev* = 1.0.

```
explicit gaussian::gaussian(RealType mean, RealType stddev)
```

Description

Constructor with parameters. *mean* is a mean value, *stddev* is a standard deviation value.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $stddev \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType gaussian::mean() const
```

Return Value

Returns the distribution parameter *mean* - mean value.

```
RealType gaussian::stddev() const
```

Return Value

Returns the distribution parameter *stddev* - standard deviation value.

Parent topic: *Distributions*

exponential

Class is used for generation of exponentially distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers exponentially distributed with displacement a and scalefactor β, where $a, \beta \in R; \beta > 0$.

The probability distribution is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} exp(-\frac{x-a}{\beta}), x \geq a \\ 0, x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - exp(-\frac{x-a}{\beta}), x \geq a \\ 0, x < a \end{cases}$$

class exponential

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = exponential_method::by_default>
class exponential {
public:
    using method_type = Method;
    using result_type = RealType;
    exponential();
    explicit exponential(RealType a, RealType beta);
    RealType a() const;
    RealType beta() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::exponential_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::exponential_method::by_default`
- `oneapi::mkl::rng::exponential_method::icdf`
- `oneapi::mkl::rng::exponential_method::icdf_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
exponential()	Default constructor
explicit exponential(RealType a, RealType beta)	Constructor with parameters
RealType a() const	Method to obtain displacement value
RealType beta() const	Method to obtain scalefactor

Member types

```
exponential::method_type = Method
```

Description

The type which defines transformation method for generation.

```
exponential::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
exponential::exponential()
```

Description

Default constructor for distribution, parameters set as a = 0.0, $beta$ = 1.0.

```
explicit exponential::exponential(RealType a, RealType beta)
```

Description

Constructor with parameters. a is a displacement, $beta$ is a scalefactor.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $beta \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType exponential::a() const
```

Return Value

Returns the distribution parameter a - displacement.

```
RealType exponential::beta() const
```

Return Value

Returns the distribution parameter *beta* - scalefactor value.

Parent topic: *Distributions*

laplace

Class is used for generation of Laplace distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Laplace distributed with mean value (or average) a, and scalefactor (b, β), where $a, \beta \in R; \beta > 0$. The scalefactor value determines the standard deviation as $\sigma = \beta\sqrt{2}$.

The probability distribution is given by:

$$f_{a,\beta}(x) = \frac{1}{\sqrt{2\beta}} exp(-\frac{|x-a|}{\beta}), x \in R.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} \frac{1}{2} exp(-\frac{|x-a|}{\beta}), x \geq a \\ 1 - \frac{1}{2} exp(-\frac{|x-a|}{\beta}), x < a \end{cases}$$

class laplace

Syntax

```
template<typename RealType = float, typename Method = laplace_method::by_default>
class laplace {
public:
    using method_type = Method;
    using result_type = RealType;
    laplace();
    explicit laplace(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::laplace_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::laplace_method::by_default`
- `oneapi::mkl::rng::laplace_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
laplace()	Default constructor
explicit laplace(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain mean value
RealType b() const	Method to obtain scalefactor value

Member types

```
laplace::method_type = Method
```

Description

The type which defines transformation method for generation.

```
laplace::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
laplace::laplace()
```

Description

Default constructor for distribution, parameters set as *a* = 0.0, and *beta* = 1.0.

```
explicit laplace::laplace(RealType a, RealType b)
```

Description

Constructor with parameters. *a* is a mean value, *beta* is a scalefactor value.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $b \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType laplace::a() const
```

Return Value

Returns the distribution parameter *a* - mean value.

```
RealType laplace::b() const
```

Return Value

Returns the distribution parameter *b* - scalefactor value.

Parent topic: *Distributions*

weibull

Class is used for generation of Weibull distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Weibull distributed with displacement a, scalefactor β, and shape α, where $a, \beta, \alpha \in R; \alpha > 0; \beta > 0$.

The probability distribution is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{\alpha}{\beta^{\alpha}}(x-a)^{\alpha-1} exp((-\frac{x-a}{\beta})^{\alpha}), x \geq a \\ 0, x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} 1 - exp((-\frac{x-a}{\beta})^{\alpha}), x \geq a \\ 0, x < a \end{cases}$$

class weibull

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = weibull_method::by_default>
class weibull {
public:
    using method_type = Method;
    using result_type = RealType;
    weibull();
    explicit weibull(RealType alpha, RealType a, RealType b);
    RealType alpha() const;
    RealType a() const;
    RealType beta() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::weibull_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::weibull_method::by_default`
- `oneapi::mkl::rng::weibull_method::icdf`
- `oneapi::mkl::rng::weibull_method::icdf_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
weibull()	Default constructor
explicit weibull(RealType alpha, RealType a, RealType beta)	Constructor with parameters
RealType alpha() const	Method to obtain shape value
RealType a() const	Method to obtain displacement value
RealType beta() const	Method to obtain scalefactor value

Member types

```
weibull::method_type = Method
```

Description

The type which defines transformation method for generation.

```
weibull::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
weibull::weibull()
```

Description

Default constructor for distribution, parameters set as *alpha* = 1.0, *a* = 0.0, and *b* = 1.0.

```
explicit weibull::weibull(RealType alpha, RealType a, RealType beta)
```

Description

Constructor with parameters. *alpha* is a shape value, *a* is a displacement value, *beta* is a scalefactor value.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $alpha \leq$ static_cast<RealType>(0.0), or $beta \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType weibull::alpha() const
```

Return Value

Returns the distribution parameter *alpha* - shape value.

```
RealType weibull::a() const
```

Return Value

Returns the distribution parameter *a* - displacement value.

```
RealType weibull::beta() const
```

Return Value

Returns the distribution parameter *beta* - scalefactor value.

Parent topic: *Distributions*

cauchy

Class is used for generation of Cauchy distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Cauchy distributed with displacement a, and scale parameter (b, β), where $a, \beta \in R; \beta > 0$.

The probability distribution is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta(1 + (\frac{x-a}{\beta})^2)}, x \in R.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi}\arctan{(\frac{x-a}{\beta})}, x \in R.$$

class cauchy

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = cauchy_method::by_default>
class cauchy {
public:
    using method_type = Method;
    using result_type = RealType;
    cauchy();
    explicit cauchy(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::cauchy_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::cauchy_method::by_default`
- `oneapi::mkl::rng::cauchy_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
cauchy()	Default constructor
explicit cauchy(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain displacement value
RealType b() const	Method to obtain scalefactor value

Member types

```
cauchy::method_type = Method
```

Description

The type which defines transformation method for generation.

```
cauchy::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
cauchy::cauchy()
```

Description

Default constructor for distribution, parameters set as a = 0.0, and b = 1.0.

```
explicit cauchy::cauchy(RealType a, RealType b)
```

Description

Constructor with parameters. a is a displacement value, b is a scalefactor value.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $b \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType cauchy::a() const
```

Return Value

Returns the distribution parameter a - displacement value.

```
RealType cauchy::b() const
```

Return Value

Returns the distribution parameter b - scalefactor value.

Parent topic: *Distributions*

rayleigh

Class is used for generation of Rayleigh distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Rayleigh distributed with displacement a, and scalefactor (b, β), where $a, \beta \in R; \beta > 0$.

The Rayleigh distribution is a special case of the *weibull* distribution, where the shape parameter *alpha* = 2 .

The probability distribution is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x-a)}{\beta^2} exp(-\frac{(x-a)^2}{\beta^2}), x \geq a \\ 0, x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - exp(-\frac{(x-a)^2}{\beta^2}), x \geq a \\ 0, x < a \end{cases}$$

class rayleigh

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = rayleigh_method::by_default>
class rayleigh {
public:
    using method_type = Method;
    using result_type = RealType;
    rayleigh();
    explicit rayleigh(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::rayleigh_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::rayleigh_method::by_default`
- `oneapi::mkl::rng::rayleigh_method::icdf`
- `oneapi::mkl::rng::rayleigh_method::icdf_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
rayleigh()	Default constructor
explicit rayleigh(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain displacement value
RealType b() const	Method to obtain scalefactor value

Member types

```
rayleigh::method_type = Method
```

Description

The type which defines transformation method for generation.

```
rayleigh::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
rayleigh::rayleigh()
```

Description

Default constructor for distribution, parameters set as a = 0.0, and b = 1.0.

```
explicit rayleigh::rayleigh(RealType a, RealType b)
```

Description

Constructor with parameters. a is a displacement value, b is a scalefactor value.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $b \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType rayleigh::a() const
```

Return Value

Returns the distribution parameter a - displacement value.

```
RealType rayleigh::b() const
```

Return Value

Returns the distribution parameter b - scalefactor value.

Parent topic: *Distributions*

lognormal

Class is used for generation of lognormally distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers lognormally distributed with mean (m, a) and standard deviation (s, σ) of subject normal distribution, displacement $(displ, b)$, and scalefactor $(scale, \beta)$, where $a, \sigma, b, \beta \in R; \sigma > 0; \beta > 0$.

The probability distribution is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \frac{1}{\sigma(x-b)\sqrt{2\pi}} exp(-\frac{[ln((x-b)/\beta)-a]^2}{2*\sigma^2}), x > b \\ 0, x \leq b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi(\frac{(ln((x-b)/\beta)-a)}{\sigma}), x > b \\ 0, x \leq b \end{cases}$$

class lognormal

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = lognormal_method::by_default>
class lognormal {
public:
    using method_type = Method;
    using result_type = RealType;
    lognormal();
    explicit lognormal(RealType m, RealType s, RealType displ = static_cast<RealType>
↪(0.0), RealType scale = static_cast<RealType>(1.0));
    RealType m() const;
    RealType s() const;
    RealType displ() const;
    RealType scale() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::lognormal_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::lognormal_method::by_default`
- `oneapi::mkl::rng::lognormal_method::box_muller2`
- `oneapi::mkl::rng::lognormal_method::icdf`
- `oneapi::mkl::rng::lognormal_method::box_muller2_accurate`
- `oneapi::mkl::rng::lognormal_method::icdf_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
lognormal()	Default constructor
explicit lognormal(RealType m, RealType s, RealType displ = static_cast<RealType>(0.0), RealType scale = static_cast<RealType>(1.0))	Constructor with parameters
RealType m() const	Method to obtain mean value
RealType s() const	Method to obtain standard deviation value
RealType displ() const	Method to obtain displacement value
RealType scale() const	Method to obtain scale-factor value

Member types

```
lognormal::method_type = Method
```

Description

The type which defines transformation method for generation.

```
lognormal::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
lognormal::lognormal()
```

Description

Default constructor for distribution, parameters set as *m* = 0.0, *s* = 1.0, *displ* = 0.0, *scale* = 1.0.

```
explicit lognormal::lognormal(RealType m, RealType s, RealType displ = static_cast
↪<RealType>(0.0), RealType scale = static_cast<RealType>(1.0))
```

Description

Constructor with parameters. *m* is a mean value, *s* is a standard deviation value, *displ* is a displacement value, *scale* is a scalefactor value.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $s \leq$ static_cast<RealType>(0.0), or $scale \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType lognormal::m() const
```

Return Value

Returns the distribution parameter *m* - mean value.

```
RealType lognormal::s() const
```

Return Value

Returns the distribution parameter *s* - standard deviation value.

```
RealType lognormal::displ() const
```

Return Value

Returns the distribution parameter *displ* - displacement value.

```
RealType lognormal::scale() const
```

Return Value

Returns the distribution parameter *scale* - scalefactor value.

Parent topic: *Distributions*

gumbel

Class is used for generation of Gumbel distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Gumbel distributed with displacement a, and scalefactor (b, β), where $a, \beta \in R; \beta > 0$.

The probability distribution is given by:

$$f_{a,\beta}(x) = \frac{1}{\beta} exp(-\frac{x-a}{\beta}) exp(-exp(\frac{x-a}{\beta})), x \in R.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = 1 - exp(-exp(\frac{x-a}{\beta})), x \in R.$$

class gumbel

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = gumbel_method::by_default>
class gumbel {
public:
    using method_type = Method;
    using result_type = RealType;
    gumbel();
    explicit gumbel(RealType a, RealType b);
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::gumbel_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::gumbel_method::by_default`
- `oneapi::mkl::rng::gumbel_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
gumbel()	Default constructor
explicit gumbel(RealType a, RealType b)	Constructor with parameters
RealType a() const	Method to obtain displacement value
RealType b() const	Method to obtain scalefactor value

Member types

```
gumbel::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gumbel::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
gumbel::gumbel()
```

Description

Default constructor for distribution, parameters set as *a* = 0.0, and *beta* = 1.0.

```
explicit gumbel::gumbel(RealType a, RealType b)
```

Description

Constructor with parameters. *a* is a displacement value, *beta* is a scalefactor value.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $b \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType gumbel::a() const
```

Return Value

Returns the distribution parameter *a* - displacement value.

```
RealType gumbel::b() const
```

Return Value

Returns the distribution parameter *b* - scalefactor value.

Parent topic: *Distributions*

gamma

Class is used for generation of gamma distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers gamma distributed with shape α, displacement a, and scale parameter β, where $a, \alpha, \beta \in R; \alpha > 0; \beta > 0$.

The probability distribution is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^{\alpha}}(x-a)^{\alpha-1}e^{-(x-a)/\beta}, x \geq a \\ 0, x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} \int_a^x \frac{1}{\Gamma(\alpha)\beta^{\alpha}}(y-a)^{\alpha-1}e^{-(y-a)/\beta}dy, x \geq a \\ 0, x < a \end{cases}$$

class gamma

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = gamma_method::by_default>
class gamma {
public:
    using method_type = Method;
    using result_type = RealType;
    gamma();
    explicit gamma(RealType alpha, RealType a, RealType beta);
    RealType alpha() const;
    RealType a() const;
    RealType beta() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::gamma_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::gamma_method::by_default`
- `oneapi::mkl::rng::gamma_method::marsaglia`
- `oneapi::mkl::rng::gamma_method::marsaglia_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
gamma()	Default constructor
explicit gamma(RealType alpha, RealType a, RealType beta)	Constructor with parameters
RealType alpha() const	Method to obtain shape value
RealType a() const	Method to obtain displacement value
RealType beta() const	Method to obtain scale value

Member types

```
gamma::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gamma::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
gamma::gamma()
```

Description

Default constructor for distribution, parameters set as *alpha* = 1.0, *a* = 0.0, and *beta* = 1.0.

```
explicit gamma::gamma(RealType alpha, RealType a, RealType beta)
```

Description

Constructor with parameters. *alpha* is a shape value, *a* is a displacement value, *beta* is a scale parameter.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $alpha \leq$ static_cast<RealType>(0.0), or $beta \leq$ static_cast<RealType>(0.0)

Characteristics

```
RealType gamma::alpha() const
```

Return Value

Returns the distribution parameter *alpha* - shape value.

```
RealType gamma::a() const
```

Return Value

Returns the distribution parameter *a* - displacement value.

```
RealType gamma::beta() const
```

Return Value

Returns the distribution parameter *beta* - scale parameter.

Parent topic: *Distributions*

beta

Class is used for generation of beta distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers beta distributed with shape parameters p and q, displacement α and scale parameter (b, β), where p, q. α, $\beta \in R; p > 0; q > 0; \beta > 0$.

The probability distribution is given by:

$$f_{p,q,\alpha,\beta}(x) = \begin{cases} \frac{1}{B(p,q)*\beta^{p+q-1}}(x-a)^{p-1} * (\beta+\alpha-x)^{q-1}, \alpha \le x < \alpha+\beta \\ 0, x < \alpha, x \ge \alpha+\beta \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, x < \alpha \\ \int_{\alpha}^{x} \frac{1}{B(p,q)*\beta^{p+q-1}}(y-\alpha)^{p-1} * (\beta+\alpha-y)^{q-1} dy, \alpha \le x < \alpha+\beta, x \in R \\ 1, x \ge \alpha+\beta \end{cases}$$

Where $B(p, 1)$ is the complete beta function.

class beta

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = beta_method::by_default>
class beta {
public:
    using method_type = Method;
    using result_type = RealType;
    beta();
```

(continues on next page)

(continued from previous page)

```
    explicit beta(RealType p, RealType q, RealType a, RealType b);
    RealType p() const;
    RealType q() const;
    RealType a() const;
    RealType b() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::beta_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::beta_method::by_default`
- `oneapi::mkl::rng::beta_method::cja`
- `oneapi::mkl::rng::beta_method::cja_accurate`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
beta()	Default constructor
explicit beta(RealType p, RealType q, RealType a, RealType b)	Constructor with parameters
RealType p() const	Method to obtain shape p
RealType q() const	Method to obtain shape q
RealType a() const	Method to obtain displacement α
RealType b() const	Method to obtain scalefactor β

Member types

```
beta::method_type = Method
```

Description

The type which defines transformation method for generation.

```
beta::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
beta::beta()
```

Description

Default constructor for distribution, parameters set as p = 1.0, q = 0.0, α = 1.0, β = 1.0.

```
explicit beta::beta(RealType p, RealType q, RealType a, RealType b)
```

Description

Constructor with parameters. p and q are shapes, α is a displacement, β is a scalefactor.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $p \leq 0.0f$, or $q \leq 0.0f$, or $\beta \leq 0.0f$

Characteristics

```
RealType beta::p() const
```

Return Value

Returns the distribution parameter p - shape.

```
RealType beta::q() const
```

Return Value

Returns the distribution parameter q - shape.

```
RealType beta::a() const
```

Return Value

Returns the distribution parameter α - displacement.

```
RealType beta::b() const
```

Return Value

Returns the distribution parameter β - scalefactor.

Parent topic: *Distributions*

chi_square

Class is used for generation of chi-square distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers chi-square distributed with n degrees of freedom, $n \in N; n > 0$.

The probability distribution is given by:

$$f_n(x) = \begin{cases} \frac{x^{\frac{n-2}{2}} e^{-\frac{x}{2}}}{2^{n/2}\Gamma(n/2)}, x \geq 0 \\ 0, x < 0 \end{cases}$$

The cumulative distribution function is as follows:

$$F_n(x) = \begin{cases} \int_0^x \frac{y^{\frac{n-2}{2}} e^{-\frac{x}{2}}}{2^{n/2}\Gamma(n/2)} dy, x \geq 0 \\ 0, x < 0 \end{cases}$$

class chi_square

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = float, typename Method = chi_square_method::by_default>
class chi_square {
public:
    using method_type = Method;
    using result_type = RealType;
    chi_square();
    explicit chi_square(std::int32_t n);
    std::int32_t n() const;
```

(continues on next page)

(continued from previous page)

```
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

typename Method = oneapi::mkl::rng::chi_square_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::chi_square_method::by_default`
- `oneapi::mkl::rng::chi_square_method::gamma_based`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
chi_square()	Default constructor
explicit chi_square(std::int32_t n)	Constructor with parameters
std::int32_t n() const	Method to obtain number of degrees of freedom *n*

Member types

```
chi_square::method_type = Method
```

Description

The type which defines transformation method for generation.

```
chi_square::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
chi_square::chi_square()
```

Description

Default constructor for distribution, parameters set as $n = 5$.

```
explicit chi_square::chi_square(std::int32_t n)
```

Description

Constructor with parameters. n is the number of degrees of freedom.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $n < 1$

Characteristics

```
std::int32_t chi_square::n() const
```

Return Value

Returns the distribution parameter n - number of degrees of freedom.

Parent topic: *Distributions*

gaussian_mv

Class is used for generation of multivariate normally distributed real types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide n random numbers d-variate normally distributed, with mean a and variance-covariance matrix C, where $a \in R^d$; C is dxd symmetric positive matrix.

The probability density function is given by:

$$f_{a,C}(x) = \frac{1}{\sqrt{det(2\pi C)}} exp(-1/2(x-a)^T C^{-1}(x-a)).$$

class gaussian_mv

Syntax

```
namespace oneapi::mkl::rng {
template<typename RealType = std::int32_t, layout Layout = layout::packed, typename␣
↪Method = gaussian_mv_method::by_default>
class gaussian_mv {
public:
    using method_type = Method;
    using result_type = RealType;
    explicit gaussian_mv(std::uint32_t dimen, std::vector<RealType> mean, std::vector
↪<RealType> matrix);
    std::int32_t dimen() const;
    std::vector<RealType> mean() const;
    std::vector<RealType> matrix() const;
};
}
```

Template parameters

typename RealType

Type of the produced values. Supported types:

- `float`
- `double`

Template parameters

oneapi::mkl::rng::layout Layout

Matrix layout:

- `oneapi::mkl::rng::layout::full`
- `oneapi::mkl::rng::layout::packed`
- `oneapi::mkl::rng::layout::diagonal`

typename Method = oneapi::mkl::rng::gaussian_mv_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::gaussian_mv_method::by_default`
- `oneapi::mkl::rng::gaussian_mv_method::box_muller`
- `oneapi::mkl::rng::gaussian_mv_method::box_muller2`
- `oneapi::mkl::rng::gaussian_mv_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
explicit gaussian_mv(std::uint32_t dimen, std::vector<RealType> mean, std::vector<RealType> matrix)	Constructor with parameters
std::int32_t dimen() const	Method to obtain number of dimensions in output random vectors
std::vector<double> mean() const	Method to obtain mean vector *a* of dimension d.
std::vector<double> matrix() const	Method to obtain variance-covariance matrix *C*

Member types

```
gaussian_mv::method_type = Method
```

Description

The type which defines transformation method for generation.

```
gaussian_mv::result_type = RealType
```

Description

The type which defines type of generated random numbers.

Constructors

```
explicit gaussian_mv::gaussian_mv(std::uint32_t dimen, std::vector<RealType> mean,␣
↪std::vector<RealType> matrix)
```

Description

Constructor with parameters. *dimen* is the number of dimensions, *mean* is a mean vector, *matrix* is a variance-covariance matrix.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $mean.size() \leq 0$, or $matrix.size() \leq 0$

Characteristics

```
std::int32_t gaussian_mv::dimen() const
```

Return Value

Returns the distribution parameter *dimen*.

```
std::vector<double> gaussian_mv::mean() const
```

Return Value

Returns the mean vector.

```
std::vector<double> gaussian_mv::matrix() const
```

Return Value

Returns the variance-covariance matrix.

Parent topic: *Distributions*

uniform (discrete)

Class is used for generation of uniformly distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers uniformly distributed over the interval $[a, b)$, where a, b are the left and right bounds of the interval, respectively, and $a, b \in R; a < b$.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b-a}, k \in \{a, a+1, ..., b-1\}$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, x < a \\ \frac{\lfloor x-a+1 \rfloor}{b-a}, a \leq x < b, x \in R \\ 1, x \geq b \end{cases}$$

class uniform

Syntax

```
namespace oneapi::mkl::rng {
template<typename Method = uniform_method::by_default>
class uniform<std::int32_t, Method> {
public:
    using method_type = Method;
    using result_type = std::int32_t;
    uniform();
    explicit uniform(std::int32_t a, std::int32_t b);
    std::int32_t a() const;
    std::int32_t b() const;
};
}
```

Template parameters

typename Method = oneapi::mkl::rng::uniform_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::uniform_method::by_default`
- `oneapi::mkl::rng::uniform_method::standard`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
uniform()	Default constructor
explicit uniform(std::int32_t a, std::int32_t b)	Constructor with parameters
std::int32_t a() const	Method to obtain left bound *a*
std::int32_t b() const	Method to obtain right bound *b*

Member types

```
method_type = Method
```

Description

The type which defines transformation method for generation.

```
result_type = std::int32_t
```

Description

The type which defines type of generated random numbers.

Constructors

```
uniform()
```

Description

Default constructor for distribution, parameters set as a = 0, b = std::numeric_limits<std::int32_t>::max().

```
uniform(std::int32_t a, std::int32_t b)
```

Description

Constructor with parameters. a is a left bound, b is a right bound, assume $a < b$.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $a \geq b$

Characteristics

```
a() const
```

Return Value

Returns the distribution parameter a - left bound.

```
b() const
```

Return Value

Returns the distribution parameter b - right bound.

Parent topic: *Distributions*

uniform_bits

Class is used for generation of uniformly distributed bits in 32/64-bit chunks.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide uniformly distributed bits in 32/64-bit chunks. It is designed to ensure each bit in the 32/64-bit chunk is uniformly distributed. Can be not supported by the specific engine.

class uniform_bits

Syntax

```
namespace oneapi::mkl::rng {
template<typename UIntType = std::uint32_t>
class uniform_bits {
public:
    using result_type = UIntType;
};
}
```

Template parameters

typename UIntType

Type of the produced values. Supported types:

- `std::uint32_t`
- `std::uint64_t`

Member types

```
uniform_bits::result_type = UIntType
```

Description

The type which defines type of generated random numbers.

Parent topic: *Distributions*

bits

Class is used for generation of underlying engine integer recurrence.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide integer random numbers. Each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated.

class bits

Syntax

```
namespace oneapi::mkl::rng {
template<typename UIntType = std::uint32_t>
class bits {
public:
    using result_type = UIntType;
};
}
```

Template parameters

typename UIntType

Type of the produced values. Supported types:

- `std::uint32_t`

Member types

```
bits::result_type = UIntType
```

Description

The type which defines type of generated random numbers.

Parent topic: *Distributions*

bernoulli

Class is used for generation of Bernoulli distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Bernoulli distributed with probability p of a single trial success, where $p \in R; 0 \leq p; p \leq 1$.

The probability distribution is given by:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, x < 0 \\ 1 - p, 0 \leq x < 1, x \in R \\ 1, x \geq 1 \end{cases}$$

class bernoulli

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = bernoulli_method::by_
↪default>
class bernoulli {
public:
    using method_type = Method;
    using result_type = IntType;
    bernoulli();
    explicit bernoulli(float p);
    float p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::bernoulli_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::bernoulli_method::by_default`
- `oneapi::mkl::rng::bernoulli_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
bernoulli()	Default constructor
explicit bernoulli(float p)	Constructor with parameters
float p() const	Method to obtain probability *p*

Member types

```
bernoulli::method_type = Method
```

Description

The type which defines transformation method for generation.

```
bernoulli::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
bernoulli::bernoulli()
```

Description

Default constructor for distribution, parameters set as $p = 0.5$f.

```
explicit bernoulli::bernoulli(float p)
```

Description

Constructor with parameters. *p* is a probability.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $p > 1.0f$, or $p < 0.0f$

Characteristics

```
float p() const
```

Return Value

Returns the distribution parameter p - probability.

Parent topic: *Distributions*

geometric

Class is used for generation of geometrically distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers geometrically distributed with probability p of a single success trial, where $p \in R; 0 < p < 1$.

The probability distribution is given by:

$$P(X = k) = p * (1 - p)^k, k = \{0, 1, 2, ...\}.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, x < 0 \\ 1 - (1 - p)^{\lfloor x+1 \rfloor}, x \geq 0 \end{cases}$$

class geometric

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = geometric_method::by_
↪default>
class geometric {
public:
    using method_type = Method;
    using result_type = IntType;
    geometric();
    explicit geometric(float p);
    float p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::geometric_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::geometric_method::by_default`
- `oneapi::mkl::rng::geometric_method::icdf`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
geometric()	Default constructor
explicit geometric(float p)	Constructor with parameters
float p() const	Method to obtain probability value

Member types

```
geometric::method_type = Method
```

Description

The type which defines transformation method for generation.

```
geometric::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
geometric::geometric()
```

Description

Default constructor for distribution, parameters set as $p = 0.5$.

```
explicit geometric::geometric(float p)
```

Description

Constructor with parameters. *p* is a probability value.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $p \geq 1.0f$, or $p \leq 0.0f$

Characteristics

```
float geometric::p() const
```

Return Value

Returns the distribution parameter *p* - probability value.

Parent topic: *Distributions*

binomial

Class is used for generation of binomially distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers binomially distributed with a number of independent Bernoulli trials m, and with probability p of a single trial success, where $p \in R; 0 \leq p \leq 1, m \in N$.

A binomially distributed variate represents the number of successes in m independent Bernoulli trials with probability of a single trial success p.

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1-p)^{m-k}, k \in \{0, 1, ..., m\}$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1-p)^{m-k}, 0 \leq x < m, x \in R \\ 1, x \geq m \end{cases}$$

class binomial

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = binomial_method::by_
↪default>
class binomial {
public:
    using method_type = Method;
    using result_type = IntType;
    binomial();
    explicit binomial(std::int32_t ntrial, double p);
    std::int32_t ntrial() const;
    double p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`

typename Method = oneapi::mkl::rng::binomial_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::binomial_method::by_default`
- `oneapi::mkl::rng::binomial_method::btpe`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
binomial()	Default constructor
explicit binomial(std::int32_t ntrial, double p)	Constructor with parameters
std::int32_t ntrial() const	Method to obtain number of independent trials m
double p() const	Method to obtain success probability of a single trial p

Member types

```
binomial::method_type = Method
```

Description

The type which defines transformation method for generation.

```
binomial::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
binomial::binomial()
```

Description

Default constructor for distribution, parameters set as $m = 5$, $p = 0.5$.

```
explicit binomial::binomial(std::int32_t ntrial, double p)
```

Description

Constructor with parameters. *ntrial* is the number of independent trials, *p* is the success probability of a single trial.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $p > 1.0$, or $p < 0.0$, or $ntrial < 1$

Characteristics

```
std::int32_t binomial::ntrial() const
```

Return Value

Returns the distribution parameter *m* - number of independent trials.

```
double binomial::p() const
```

Return Value

Returns the distribution parameter *p* - success probability of a single trial.

Parent topic: *Distributions*

hypergeometric

Class is used for generation of hypergeometrically distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers hypergeometrically distributed with lot size l, size of sampling s, and number of marked elements in the lot m, where $l, m, s \in N \bigcup \{0\}; l \geq max(s, m)$.

Consider a lot of l elements comprising m marked and l - m unmarked elements. A trial sampling without replacement of exactly s elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of s elements contains exactly k marked elements.

The probability distribution is given by:

$$P_(X = k) = \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, k \in \{max(0, s + m - l), ..., min(s, m)\}.$$

The cumulative distribution function is as follows:

$$F_{l,s,m}(x) = \begin{cases} 0, x < max(0, s + m - l) \\ \sum_{k=max(0,s+m-l)}^{\lfloor x \rfloor} \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, max(0, s + m - l) \leq x \leq min(s, m) \\ 1, x > min(s, m) \end{cases}$$

class hypergeometric

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = hypergeometric_method::by_
↪default>
class hypergeometric {
public:
    using method_type = Method;
    using result_type = IntType;
    hypergeometric();
    explicit hypergeometric(std::int32_t l, std::int32_T s, std::int32_T m);
    std::int32_t s() const;
    std::int32_t m() const;
    std::int32_t l() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::hypergeometric_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::hypergeometric_method::by_default`
- `oneapi::mkl::rng::hypergeometric_method::h2pe`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
hypergeometric()	Default constructor
explicit hypergeometric(std::int32_t l, std::int32_T s, std::int32_T m)	Constructor with parameters
std::int32_t s() const	Method to obtain lot size
std::int32_t m() const	Method to obtain size of sampling without replacement
std::int32_t l() const	Method to obtain number of marked elements

Member types

```
hypergeometric::method_type = Method
```

Description

The type which defines transformation method for generation.

```
hypergeometric::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
hypergeometric::hypergeometric()
```

Description

Default constructor for distribution, parameters set as $l = 1$, $s = 1$, $m = 1$.

```
explicit hypergeometric::hypergeometric(std::int32_t l, std::int32_T s, std::int32_T␣
↪m)
```

Description

Constructor with parameters. l is a lot size, s is a size of sampling without replacement, m is a number of marked elements.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $s < 0$, or $m < 0$, or $l < (s > m ? s : m)$

Characteristics

```
std::int32_t hypergeometric::l() const
```

Return Value

Returns the distribution parameter l - lot size value.

```
std::int32_t hypergeometric::s() const
```

Return Value

Returns the distribution parameter s - size of sampling without replacement.

```
std::int32_t hypergeometric::m() const
```

Return Value

Returns the distribution parameter m - number of marked elements.

Parent topic: *Distributions*

poisson

Class is used for generation of Poisson distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers Poisson distributed with distribution parameter λ, where $\lambda \in R; \lambda > 0;$.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}.$$

The cumulative distribution function is as follows:

$$F_\lambda(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, x \geq 0 \\ 0, x < 0 \end{cases}$$

class poisson

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = poisson_method::by_
↪default>
class poisson {
public:
    using method_type = Method;
    using result_type = IntType;
    poisson();
    explicit poisson(double lambda);
    double lambda() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`

typename Method = oneapi::mkl::rng::poisson_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::poisson_method::by_default`
- `oneapi::mkl::rng::poisson_method::ptpe`
- `oneapi::mkl::rng::poisson_method::gaussian_icdf_based`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
poisson()	Default constructor
explicit poisson(double lambda)	Constructor with parameters
double lambda() const	Method to obtain distribution parameter

Member types

```
poisson::method_type = Method
```

Description

The type which defines transformation method for generation.

```
poisson::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
poisson::poisson()
```

Description

Default constructor for distribution, parameters set as *lambda* = 0.5.

```
explicit poisson::poisson(double lambda)
```

Description

Constructor with parameters. *lambda* is a distribution parameter.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $lambda \leq 0.0$

Characteristics

```
double poisson::lambda() const
```

Return Value

Returns the distribution parameter *lambda*.

Parent topic: *Distributions*

poisson_v

Class is used for generation of Poisson distributed integer types random numbers with varying mean.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide n random numbers Poisson distributed, with distribution parameter λ_i, where $\lambda_i \in R; \lambda_i > 0; i = 1, ..., n$.

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k e^{-\lambda_i}}{k!}, k \in \{0, 1, 2, ...\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, x \geq 0 \\ 0, x < 0 \end{cases}$$

class poisson_v

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = poisson_v_method::by_
↪default>
class poisson_v {
public:
    using method_type = Method;
    using result_type = IntType;
    explicit poisson_v(std::vector<double> lambda);
    std::vector<double> lambda() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`

typename Method = oneapi::mkl::rng::poisson_v_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::poisson_v_method::by_default`
- `oneapi::mkl::rng::poisson_v_method::gaussian_icdf_based`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
explicit poisson_v(std::vector<double> lambda)	Constructor with parameters
std::vector<double> lambda() const	Method to obtain distribution parameter

Member types

```
poisson_v::method_type = Method
```

Description

The type which defines transformation method for generation.

```
poisson_v::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
explicit poisson_v::poisson_v(std::vector<double> lambda)
```

Description

Constructor with parameters. *lambda* is a distribution parameter.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $lambda.size() \leq 1$

Characteristics

```
double poisson_v::lambda() const
```

Return Value

Returns the distribution parameter *lambda*.

Parent topic: *Distributions*

negative_binomial

Class is used for generation of negative binomially distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide random numbers negative binomially distributed with distribution parameters a and p, where $p, a \in R; 0 \leq p \leq 1, a > 0$.

The probability distribution is given by:

$$P(X = k) = C^k_{a+k-1} p^a (1-p)^k, k \in \{0, 1, 2, ...\}$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C^k_{a+k-1} p^a (1-p)^k, x \geq 0, x \in R \\ 0, x < 0 \end{cases}$$

class negative_binomial

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = negative_binomial_
↪method::by_default>
class negative_binomial {
public:
```

(continues on next page)

(continued from previous page)

```
    using method_type = Method;
    using result_type = IntType;
    negative_binomial();
    explicit negative_binomial(double a, double p);
    double a() const;
    double p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::negative_binomial_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::negative_binomial_method::by_default`
- `oneapi::mkl::rng::negative_binomial_method::nbar`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
negative_binomial()	Default constructor
explicit negative_binomial(double a, double p)	Constructor with parameters
double a() const	Method to obtain the first distribution parameter *a*
double p() const	Method to obtain the second distribution parameter *p*

Member types

```
negative_binomial::method_type = Method
```

Description

The type which defines transformation method for generation.

```
negative_binomial::result_type = IntType
```

Description

The type which defines type of generated random numbers.

Constructors

```
negative_binomial::negative_binomial()
```

Description

Default constructor for distribution, parameters set as $a = 0.1$, $p = 0.5$.

```
explicit negative_binomial::negative_binomial(double a, double p)
```

Description

Constructor with parameters. a is the first distribution parameter, p is the second distribution parameter.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $p \geq 1.0$, or $p \leq 0.0$, or $a \leq 0.0$

Characteristics

```
double negative_binomial::a() const
```

Return Value

Returns the distribution parameter a - the first distribution parameter.

```
double negative_binomial::p() const
```

Return Value

Returns the distribution parameter p - the second distribution parameter.

Parent topic: *Distributions*

multinomial

Class is used for generation of multinomially distributed integer types random numbers.

Description

The class object is used in the *oneapi::mkl::rng::generate()* function to provide *n* random numbers multinomially distributed, with independent trials $(ntrial, m)$ and possible mutually exclusive outcomes k, with corresponding probabilities p_i, where $p_i \in R; 0 \leq p_i \leq 1; m, k \in N$.

The probability distribution is given by:

$$P(X_1 = x_1, ..., X_k = x_k) = \frac{m!}{\Pi_{i=1}^{k} x_i!} \Pi_{i=1}^{k} p_i^{x_i}, 0 \leq x_i \leq m, \sum_{i=1}^{k} x_i = m$$

class multinomial

Syntax

```
namespace oneapi::mkl::rng {
template<typename IntType = std::int32_t, typename Method = multinomial_method::by_
↪default>
class multinomial {
public:
    using method_type = Method;
    using result_type = IntType;
    explicit multinomial(double ntrial, std::vector<double> p);
    std::int32_t ntrial() const;
    std::vector<double> p() const;
};
}
```

Template parameters

typename IntType

Type of the produced values. Supported types:

- `std::int32_t`
- `std::uint32_t`

typename Method = oneapi::mkl::rng::multinomial_method::by_default Transformation method, which will be used for generation. Supported types:

- `oneapi::mkl::rng::multinomial_method::by_default`
- `oneapi::mkl::rng::multinomial_method::poisson_icdf_based`

See description of the methods in *Distributions methods template parameter*.

Class Members

Routine	Description
explicit multinomial(double ntrial, std::vector<double> p)	Constructor with parameters
std::int32_t ntrial() const	Method to obtain number of independent trials
std::vector<double> p() const	Method to obtain probability vector of possible outcomes

Member types

```
multinomial::method_type = Method
```

Description

The type which defines the transformation method for generation.

```
multinomial::result_type = IntType
```

Description

The type which defines the type of generated random numbers.

Constructors

```
explicit multinomial::multinomial(double ntrial, std::vector<double> p)
```

Description

Constructor with parameters. *ntrial* is a number of independent trials, *p* is a probability vector.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $ntrial < 0$, or $p.size() < 1$

Characteristics

```
std::int32_t multinomial::ntrial() const
```

Return Value

Returns the distribution parameter *ntrial*.

```
std::vector<double> multinomial::p() const
```

Return Value

Returns the distribution parameter *p*.

Parent topic: *Distributions*

Bibliography

For more information about the VS RNG functionality, refer to the following publications:

- **VS RNG**

[Bratley88] Bratley P. and Fox B.L. *Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.

[Bratley92] Bratley P., Fox B.L., and Niederreiter H. *Implementation and Tests of Low-Discrepancy Sequences*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.

[Coddington94] Coddington, P. D. *Analysis of Random Number Generators Using Monte Carlo Simulation*. Int. J. Mod. Phys. C-5, 547, 1994.

[L'Ecuyer99] L'Ecuyer, Pierre. *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. Mathematics of Computation, 68, 225, 249-260, 1999.

[L'Ecuyer99a] L'Ecuyer, Pierre. *Good Parameter Sets for Combined Multiple Recursive Random Number Generators*. Operations Research, 47, 1, 159-164, 1999.

[Kirkpatrick81] Kirkpatrick, S., and Stoll, E. *A Very Fast Shift-Register Sequence Random Number Generator*. Journal of Computational Physics, V. 40. 517-526, 1981.

[Matsumoto98] Matsumoto, M., and Nishimura, T. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.

[Matsumoto00] Matsumoto, M., and Nishimura, T. *Dynamic Creation of Pseudorandom Number Generators*, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html.

[NAG] NAG Numerical Libraries. http://www.nag.co.uk/numeric/numerical_libraries.asp

[Saito08] Saito, M., and Matsumoto, M. *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*. Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, Pages 607 – 622, 2008.

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html

[Salmon11] Salmon, John K., Morales, Mark A., Dror, Ron O., and Shaw, David E., *Parallel Random Numbers: As Easy as 1, 2, 3*. SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.

[Sobol76] Sobol, I.M., and Levitan, Yu.L. *The production of points uniformly distributed in a multidimensional cube*. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).

[MT2203] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html.

[FIPS-197] Federal Information Processing Standards Publication 197, ADVANCED ENCRYPTION STANDARD (AES)

12.2.5 Summary Statistics

The oneMKL provides a set of *Summary Statistics routines* that compute basic statistical estimates for single and double precision multi-dimensional datasets.

Summary Statistics Overview

Definitions

The oneMKL Summary Statistics domains consists of:

- Dataset structure. The structure consolidates the information of a multi-dimensional dataset (see detailed description in *dataset*).
- Computation routines. The routines are represented as free functions (see detailed description for each routine in *Summary Statistics Routines*):
  - Raw and central sums / moments up to the fourth order
  - Variation coefficient
  - Skewness and excess kurtosis (further referred as kurtosis)
  - Minimum and maximum

Refer to *oneMKL Summary Statistics Usage Model*.

oneMKL Summary Statistics Usage Model

Description

A typical algorithm for summary statistics is as follows:

1. Create and initialize an object for dataset.
2. Call the summary statistics routine to calculate the appropriate estimate.

The following example demonstrates how to calculate mean values for a 3-dimensional dataset filled with random numbers. For dataset creation, the *make_dataset* helper function is used.

Buffer-based example

```
#include <iostream>
#include <vector>

#include "CL/sycl.hpp"
#include "mkl_stats_sycl.hpp"

int main() {
    sycl::queue queue;

    const size_t n_observations = 1000;
```

(continues on next page)

(continued from previous page)

```
    const size_t n_dims = 3;
    std::vector<float> x(n_observations * n_dims);
    // fill x storage with random numbers
    for(int i = 0; i < n_dims, i++) {
      for(int j = 0; j < n_observations; j++) {
            x[j + i * n_observations] = float(std::rand()) / float(RAND_MAX)
        }
    }
    //create buffer for dataset
    sycl::buffer<float, 1> x_buf(x.data(), x.size());
    // create buffer for mean values
    sycl::buffer<float, 1> mean_buf(n_dims);
    // create oneapi::mkl::stats::dataset
    auto dataset = oneapi::mkl::stats::make_dataset<oneapi::mkl::stats::layout::row_
↪major>(n_dims, n_observations, x_buf);


    oneapi::mkl::stats::mean(queue, dataset, mean_buf);


    // create host accessor for mean_buf to print results
    auto acc = mean_buf.template get_access<sycl::access::mode::read>();


    for(int i = 0; i < n_dims; i++) {
      std::cout << "Mean value for dimension " << i << ": " << acc[i] << std::endl;
    }
    return 0;
}
```

USM-based example

```
#include <iostream>
#include <vector>

#include "CL/sycl.hpp"
#include "mkl_stats_sycl.hpp"

int main() {
    sycl::queue queue;

    const size_t n_observations = 1000;
    const size_t n_dims = 3;

    sycl::usm_allocator<float, sycl::usm::alloc::shared> allocator(queue);

    std::vector<float, decltype(allocator)> x(n_observations * n_dims, allocator);
    // fill x storage with random numbers
    for(int i = 0; i < n_dims, i++) {
      for(int j = 0; j < n_observations; j++) {
            x[j + i * n_observations] = float(std::rand()) / float(RAND_MAX)
        }
    }
    std::vector<float, decltype(allocator)> mean_buf(n_dims, allocator);
```

(continues on next page)

(continued from previous page)

```
    // create oneapi::mkl::stats::dataset
    auto dataset = oneapi::mkl::stats::make_dataset<oneapi::mkl::stats::layout::row_
↪major>(n_dims,  n_observations, x);

   sycl::event event = oneapi::mkl::stats::mean(queue, dataset, mean);
   event.wait();
   for(int i = 0; i < n_dims; i++) {
     std::cout << "Mean value for dimension " << i << ": " << mean[i] << std::endl;
   }
   return 0;
}
```

USM usage

You can also use USM with raw pointers by using the sycl::malloc_shared/malloc_device functions.

Parent topic: *Summary Statistics*

dataset

The structure consolidates the information of a multi-dimensional dataset.

Description

The `dataset` struct object is used in *Summary Statistics Routines* as a multi-dimensional data storage. `dataset` struct contains information about observations matrix and its size (dimensions x observations), observations weights and indices for dimensions (defines dimensions to be processed).

structure dataset (Buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<layout ObservationsLayout, typename Type>
    struct dataset<ObservationsLayout, sycl::buffer<Type, 1>> {

    explicit dataset(std::int64_t n_dims_, std::int64_t n_observations_,
                sycl::buffer<Type, 1> observations_, sycl::buffer<Type, 1> weights_ =
↪{0},
                sycl::buffer<std::int64_t, 1> indices_ = {0}) :
                n_dims(n_dims_), n_observations(n_observations_),
                observations(observations_),
                weights(weights_), indices(indices_) {};

    std::int64_t n_dims;
    std::int64_t n_observations;
    sycl::buffer<Type, 1> observations;
    sycl::buffer<Type, 1> weights = {0};
    sycl::buffer<std::int64_t, 1> indices = {0};
    static constexpr layout layout = ObservationsLayout;
    };
}
```

Template parameters

typename Type Type of the multi-dimensional data. Supported types:

- `float`
- `double`

oneapi::mkl::stats::layout ObservationsLayout Type of the multi-dimensional data layout. Supported types:

- `oneapi::mkl::stats::layout::row_major`
- `oneapi::mkl::stats::layout::col_major`

Struct Members

Routine	Description
explicit dataset(std::int64_t n_dims_, std::int64_t n_observations_, sycl::buffer<Type, 1> observations_, sycl::buffer<Type, 1> weights_ = {0}, sycl::buffer<std::int64_t, 1> indices_ = {0})	Constructor

Constructors

```
explicit dataset::dataset(std::int64_t n_dims_, std::int64_t n_observations_,
    sycl::buffer<Type, 1> observations_,
    sycl::buffer<Type, 1> weights_ = {0},
    sycl::buffer<std::int64_t, 1> indices_ = {0})
```

Description

Constructor with parameters.

- *n_dims_* is the number of dimensions
- *n_observations_* is the number of observations
- *observations_* is the matrix of observations
- *weights_* is an optional parameter, represents array of weights for observations (of size *n_observations*). If the parameter is not specified, each observation is assigned a weight equal 1.
- *indices_* is an optional parameter, represents array of dimensions that are processed (of size *n_dims*). If the parameter is not specified, all dimensions are processed.

Throws

oneapi::mkl::invalid_argument Exception is thrown when *n_dims_* $\leq$ 0, or *n_observations_* $\leq$ 0, or *observations_.get_count() == 0*

structure dataset (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<layout ObservationsLayout, typename Type>
    struct dataset<Type*, ObservationsLayout> {
    explicit dataset(std::int64_t n_dims_, std::int64_t n_observations_, Type*␣
↪observations_,
                Type* weights_ = nullptr, std::int64_t* indices_ = nullptr) :
                n_dims(n_dims_), n_observations(n_observations_),
                observations(observations_),
                weights(weights_), indices(indices_) {};

    std::int64_t n_dims;
    std::int64_t n_observations;
    Type* observations;
    Type* weights = nullptr;
    std::int64_t* indices = nullptr;
    static constexpr layout layout = ObservationsLayout;
    };
}
```

Template parameters

typename Type Type of the multi-dimensional data. Supported types:

- `float`
- `double`

oneapi::mkl::stats::layout ObservationsLayout Type of the multi-dimensional data layout. Supported types:

- `oneapi::mkl::stats::layout::row_major`
- `oneapi::mkl::stats::layout::col_major`

Struct Members

Routine	Description
explicit dataset(std::int64_t n_dims_, std::int64_t n_observations_, Type observations_, Type* weights_ = nullptr, std::int64_t* indices_ = nullptr)*	Constructor

Constructors

```
explicit dataset::dataset(std::int64_t n_dims_, std::int64_t n_observations_,
    Type* observations_,
    Type* weights_ = nullptr,
    std::int64_t* indices_ = nullptr)
```

Description

Constructor with parameters.

- *n_dims_* is the number of dimensions
- *n_observations_* is the number of observations
- *observations_* is the matrix of observations
- *weights_* is an optional parameter, represents array of weights for observations (of size *n_observations*). If the parameter is not specified, each observation is assigned a weight equal 1.
- *indices_* is an optional parameter, represents array of dimensions that are processed (of size *n_dims*). If the parameter is not specified, all dimensions are processed.

Throws

oneapi::mkl::invalid_argument Exception is thrown when *n_dims_* ≤ 0, or *n_observations_* ≤ 0, or *observations_* == *nullptr*

Parent topic: *Summary Statistics*

Summary Statistics Routines

The oneMKL Summary Statistics routines calculate next estimates:

Routine	Description
raw_sum	Raw sums up to the fourth order
central_sum	Central sums up to the fourth order
central_sum with provided mean	Central sums up to the fourth order with provided mean
mean	Mean value
raw_moment	Raw moments up to the fourth order
central_moment	Central moments up to the fourth order
central_moment with provided mean	Central moments up to the fourth order with provided mean
variation	Variation coefficient
variation with provided mean	Variation coefficient with provided mean
skewness	Skewness value
skewness with provided mean	Skewness value with provided mean
kurtosis	Kurtosis value
kurtosis with provided mean	Kurtosis value with provided mean
min	Min value
max	Max value
min_max	Min and max values

Parent topic: *Summary Statistics*

raw_sum

Entry point to compute raw sums up to the 4th order.

Description and Assumptions

The oneapi::mkl::stats::raw_sum function is used to compute an array of raw sums up to the 4th order (raw sums for each dataset's dimension).

raw_sum supports the following precisions for data:

T
float
double

raw_sum (Buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void raw_sum(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> sum,
    sycl::buffer<Type, 1> raw_sum_2 = {0},
    sycl::buffer<Type, 1> raw_sum_3 = {0},
    sycl::buffer<Type, 1> raw_sum_4 = {0});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

sum sycl::buffer array of sum values.

raw_sum_2 Optional parameter. sycl::buffer array of 2nd order raw sum values.

raw_sum_3 Optional parameter. sycl::buffer array of 3rd order raw sum values.

raw_sum_4 Optional parameter. sycl::buffer array of 4th order raw sum values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when sum.get_count() == 0 & raw_sum_2.get_count() == 0 & raw_sum_3.get_count() == 0 & raw_sum_4.get_count() == 0, or dataset object is invalid

raw_sum (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event raw_sum(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* sum,
    Type* raw_sum_2 = nullptr,
    Type* raw_sum_3 = nullptr,
    Type* raw_sum_4 = nullptr,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

sum Pointer to the array of sum values.

raw_sum_2 Optional parameter. Pointer to the array of the 2nd order raw sum values.

raw_sum_3 Optional parameter. Pointer to the array of the 3rd order raw sum values.

raw_sum_4 Optional parameter. Pointer to the array of the 2nd order raw sum values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when sum == nullptr & raw_sum_2 == nullptr & raw_sum_3 == nullptr & raw_sum_4 == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

central_sum

Entry point to compute central sums up to the 4th order.

Description and Assumptions

The oneapi::mkl::stats::central_sum function is used to compute an array of central sums up to the 4th order (central sums for each dataset's dimension).

central_sum supports the following precisions for data:

T
float
double

central_sum (Buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void central_sum(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> central_sum_2,
    sycl::buffer<Type, 1> central_sum_3 = {0},
    sycl::buffer<Type, 1> central_sum_4 = {0});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

central_sum_2 sycl::buffer array of 2nd order central sum values.

central_sum_3 Optional parameter. sycl::buffer array of 3rd order central sum values.

central_sum_4 Optional parameter. sycl::buffer array of 4th order central sum values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when central_sum_2.get_count() == 0 & central_sum_3.get_count() == 0 & central_sum_4.get_count() == 0, or dataset object is invalid

central_sum (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event central_sum(
    sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* central_sum_2,
    Type* central_sum_3 = nullptr,
    Type* central_sum_4 = nullptr,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

central_sum_2 Pointer to the array of the 2nd order central sum values.

central_sum_3 Optional parameter. Pointer to the array of the 3rd order central sum values.

central_sum_4 Optional parameter. Pointer to the array of the 2nd order central sum values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when central_sum_2 == nullptr & central_sum_3 == nullptr & central_sum_4 == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

central_sum with provided mean

Entry point to compute central sums up to the 4th order with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::central_sum function is used to compute an array of central sums up to the 4th order (central sums for each dataset's dimension) with the provided mean values.

central_sum with provided mean supports the following precisions for data:

T
`float`
`double`

central_sum with provided mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void central_sum(sycl::queue& queue,
    sycl::buffer<Type, 1> mean,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> central_sum_2,
    sycl::buffer<Type, 1> central_sum_3 = {0},
    sycl::buffer<Type, 1> central_sum_4 = {0});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean sycl::buffer to the array of provided mean values.

data Dataset which is used for computation.

Output Parameters

central_sum_2 sycl::buffer array of 2nd order central sum values.

central_sum_3 Optional parameter. sycl::buffer array of 3rd order central sum values.

central_sum_4 Optional parameter. sycl::buffer array of 4th order central sum values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when central_sum_2.get_count() == 0 & central_sum_3.get_count() == 0 & central_sum_4.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

central_sum with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event central_sum(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* central_sum_2,
    Type* central_sum_3 = nullptr,
    Type* central_sum_4 = nullptr,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean Pointer to the array of provided mean values.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

central_sum_2 Pointer to the array of the 2nd order central sum values.

central_sum_3 Optional parameter. Pointer to the array of the 3rd order central sum values.

central_sum_4 Optional parameter. Pointer to the array of the 2nd order central sum values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when central_sum_2 == nullptr & central_sum_3 == nullptr & central_sum_4 == nullptr, or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

mean

Entry point to compute mean values.

Description and Assumptions

The oneapi::mkl::stats::mean function is used to compute a mean array (mean value for each dataset's dimension).

mean supports the following precisions for data:

T
float
double

mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void mean(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> mean);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

mean sycl::buffer array of mean values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when mean.get_count() == 0, or dataset object is invalid

mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event mean(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* mean,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

mean Pointer to the array of mean values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

raw_moment

Entry point to compute raw moments up to the 4th order.

Description and Assumptions

The oneapi::mkl::stats::raw_moment function is used to compute array of raw moments up to the 4th order (raw moments for each dataset's dimension).

raw_moment supports the following precisions for data:

T
float
double

oneapi::mkl::stats::raw_moment (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = fast, typename Type, layout ObservationsLayout>
    void raw_moment(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> mean,
    sycl::buffer<Type, 1> raw_moment_2 = {0},
    sycl::buffer<Type, 1> raw_moment_3 = {0},
    sycl::buffer<Type, 1> raw_moment_4 = {0});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

mean sycl::buffer array of mean values.

raw_moment_2 Optional parameter. sycl::buffer array of 2nd order raw moment values.

raw_moment_3 Optional parameter. sycl::buffer array of 3rd order raw moment values.

raw_moment_4 Optional parameter. sycl::buffer array of 4th order raw moment values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when mean.get_count() == 0 & raw_moment_2.get_count() == 0 & raw_moment_3.get_count() == 0 & raw_moment_4.get_count() == 0, or dataset object is invalid

raw_moment (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event raw_moment(
    sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* mean,
    Type* raw_moment_2 = nullptr,
    Type* raw_moment_3 = nullptr,
    Type* raw_moment_4 = nullptr,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

mean Pointer to the array of mean values.

raw_moment_2 Optional parameter. Pointer to the array of the 2nd order raw moment values.

raw_moment_3 Optional parameter. Pointer to the array of the 3rd order raw moment values.

raw_moment_4 Optional parameter. Pointer to the array of the 2nd order raw moment values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when mean == nullptr & raw_moment_2 == nullptr & raw_moment_3 == nullptr & raw_moment_4 == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

central_moment

Entry point to compute central moments up to the 4th order.

Description and Assumptions

The oneapi::mkl::stats::central_moment function is used to compute an array of central moments up to the 4th order (central moments for each dataset's dimension).

central_moment supports the following precisions for data:

T
float
double

central_moment (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = oneapi::mkl::stats::method::fast, typename Type,
    layout ObservationsLayout>
    void central_moment(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> central_moment_2,
    sycl::buffer<Type, 1> central_moment_3 = {0},
    sycl::buffer<Type, 1> central_moment_4 = {0});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

central_moment_2 sycl::buffer array of 2nd order central moment values.

central_moment_3 Optional parameter. sycl::buffer array of 3rd order central moment values.

central_moment_4 Optional parameter. sycl::buffer array of 4th order central moment values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when central_moment_2.get_count() == 0 & central_moment_3.get_count() == 0 & central_moment_4.get_count() == 0, or dataset object is invalid

central_moment (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event central_moment(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data, Type* central_moment_2,
    Type* central_moment_3 = nullptr, Type* central_moment_4 = nullptr,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

central_moment_2 Pointer to the array of the 2nd order central moment values.

central_moment_3 Optional parameter. Pointer to the array of the 3rd order central moment values.

central_moment_4 Optional parameter. Pointer to the array of the 2nd order central moment values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when central_moment_2 == nullptr & central_moment_3 == nullptr & central_moment_4 == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

central_moment with provided mean

Entry point to compute central moments up to the 4th order with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::central_moment function is used to compute an array of central moments up to the 4th order (central moments for each dataset's dimension) with the provided mean values.

central_moment with provided mean supports the following precisions for data:

T
float
double

central_moment with provided mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void central_moment(sycl::queue& queue,
    sycl::buffer<Type, 1> mean,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> central_moment_2,
    sycl::buffer<Type, 1> central_moment_3 = {0},
    sycl::buffer<Type, 1> central_moment_4 = {0});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean sycl::buffer to the array of provided mean values.

data Dataset which is used for computation.

Output Parameters

central_moment_2 sycl::buffer array of 2nd order central moment values.

central_moment_3 Optional parameter. sycl::buffer array of 3rd order central moment values.

central_moment_4 Optional parameter. sycl::buffer array of 4th order central moment values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when central_moment_2.get_count() == 0 & central_moment_3.get_count() == 0 & central_moment_4.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

central_moment with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event central_moment(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* central_moment_2,
    Type* central_moment_3 = nullptr,
    Type* central_moment_4 = nullptr,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean Pointer to the array of provided mean values.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

central_moment_2 Pointer to the array of the 2nd order central moment values.

central_moment_3 Optional parameter. Pointer to the array of the 3rd order central moment values.

central_moment_4 Optional parameter. Pointer to the array of the 2nd order central moment values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when central_moment_2 == nullptr & central_moment_3 == nullptr & central_moment_4 == nullptr or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

variation

Entry point to compute variation.

Description and Assumptions

The oneapi::mkl::stats::variation function is used to compute a variation array (variation for each dataset's dimension).

variation supports the following precisions for data:

T
float
double

variation (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void variation(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> variation);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

variation sycl::buffer array of variation values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when variation.get_count() == 0, or dataset object is invalid

variation (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event variation(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* variation,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

variation Pointer to the array of variation values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when variation == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

variation with provided mean

Entry point to compute variation with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::variation function is used to compute an array of variation (variation for each dataset's dimension) with the provided mean values.

variation with provided mean supports the following precisions for data:

T
`float`
`double`

oneapi::mkl::stats::variation (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void variation(sycl::queue& queue, sycl::buffer<Type, 1> mean,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> variation);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean sycl::buffer to the array of provided mean values.

data Dataset which is used for computation.

Output Parameters

variation sycl::buffer array of variation values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when variation.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

variation with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event variation(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* variation,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean Pointer to the array of provided mean values.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

variation Pointer to the array of the variation values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when variation == nullptr, or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

skewness

Entry point to compute skewness.

Description and Assumptions

The oneapi::mkl::stats::skewness function is used to compute a skewness array (skewness for each dataset's dimension).

skewness supports the following precisions for data:

T
`float`
`double`

skewness (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void skewness(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> skewness);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

skewness sycl::buffer array of skewness values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when skewness.get_count() == 0, or dataset object is invalid

skewness (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event skewness(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* skewness,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

skewness Pointer to the array of skewness values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when skewness == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

skewness with provided mean

Entry point to compute skewness with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::skewness function is used to compute an array of skewness (skewness for each dataset's dimension) with the provided mean values.

skewness with provided mean supports the following precisions for data:

T
`float`
`double`

skewness with provided mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void skewness(sycl::queue& queue,
    sycl::buffer<Type, 1> mean,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> skewness);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean sycl::buffer to the array of provided mean values.

data Dataset which is used for computation.

Output Parameters

skewness sycl::buffer array of skewness values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when skewness.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

skewness with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event skewness(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* skewness,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean Pointer to the array of provided mean values.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

skewness Pointer to the array of the skewness values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when skewness == nullptr, or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

kurtosis

Entry point to compute kurtosis.

Description and Assumptions

The oneapi::mkl::stats::kurtosis function is used to compute a kurtosis array (kurtosis for each dataset's dimension).

kurtosis supports the following precisions for data:

T
`float`
`double`

kurtosis (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
    template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void kurtosis(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> kurtosis);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

kurtosis sycl::buffer array of kurtosis values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when kurtosis.get_count() == 0, or dataset object is invalid

kurtosis (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event kurtosis(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* kurtosis,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`
- `oneapi::mkl::stats::method::one_pass`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

kurtosis Pointer to the array of kurtosis values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when kurtosis == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

kurtosis with provided mean

Entry point to compute kurtosis with the provided mean values.

Description and Assumptions

The oneapi::mkl::stats::kurtosis function is used to compute an array of kurtosis (kurtosis for each dataset's dimension) with the provided mean values.

kurtosis with provided mean supports the following precisions for data:

T
`float`
`double`

kurtosis with provided mean (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type,
    layout ObservationsLayout>
    void oneapi::mkl::stats::kurtosis(sycl::queue& queue,
    sycl::buffer<Type, 1> mean,
    const oneapi::mkl::stats::dataset<sycl::buffer<Type, 1>, ObservationsLayout>&␣
↪data,
    sycl::buffer<Type, 1> kurtosis);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean sycl::buffer to the array of provided mean values.

data Dataset which is used for computation.

Output Parameters

kurtosis sycl::buffer array of kurtosis values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when kurtosis.get_count() == 0, or mean.get_count() == 0, or dataset object is invalid

kurtosis with provided mean (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = fast, typename Type, layout ObservationsLayout>
    sycl::event kurtosis(sycl::queue& queue,
    Type* mean,
    const dataset<ObservationsLayout, Type*>& data,
    Type* kurtosis,
```

(continues on next page)

(continued from previous page)

```
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

mean Pointer to the array of provided mean values.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

kurtosis Pointer to the array of the kurtosis values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when kurtosis == nullptr, or mean == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

min

Entry point to compute min values.

Description and Assumptions

The oneapi::mkl::stats::min function is used to compute min arrays (min value for each dataset's dimension).

min supports the following precisions for data:

T
float
double

min (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = fast, typename Type, layout ObservationsLayout>
    void min(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> min);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

min sycl::buffer array of min values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when min.get_count() == 0, or dataset object is invalid

min (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = fast, typename Type, layout ObservationsLayout>
    sycl::event min(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* min,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

min Pointer to the array of min values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when min == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

max

Entry point to compute max values.

Description and Assumptions

The oneapi::mkl::stats::max function is used to compute a max values arrays (max value for each dataset's dimension).

max supports the following precisions for data:

T
float
double

max (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void max(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> max);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

max sycl::buffer array of max values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when max.get_count() == 0, or dataset object is invalid

max (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event max(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* max,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

max Pointer to the array of max values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when max == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

min_max

Entry point to compute min and max values.

Description and Assumptions

The oneapi::mkl::stats::min_max function is used to compute min and max arrays (min and max values for each dataset's dimension).

min_max supports the following precisions for data:

T
float
double

min_max (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    void min_max(sycl::queue& queue,
    const dataset<ObservationsLayout, sycl::buffer<Type, 1>>& data,
    sycl::buffer<Type, 1> min,
    sycl::buffer<Type, 1> max);
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

Output Parameters

min sycl::buffer array of min values.

max sycl::buffer array of max values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when min.get_count() == 0, or max.get_count() == 0, or dataset object is invalid

min_max (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<method Method = method::fast, typename Type, layout ObservationsLayout>
    sycl::event min_max(sycl::queue& queue,
    const dataset<ObservationsLayout, Type*>& data,
    Type* min,
    Type* max,
    const sycl::vector_class<sycl::event> &dependencies = {});
}
```

Template Parameters

Method Method which is used for estimate computation. The specific values are as follows:

- `oneapi::mkl::stats::method::fast`

Type Data precision.

ObservationsLayout Data layout. The specific values are described in *dataset*.

Input Parameters

queue The queue where the routine should be executed.

data Dataset which is used for computation.

dependencies Optional parameter. List of events to wait for before starting computation, if any.

Output Parameters

min Pointer to the array of min values.

max Pointer to the array of max values.

Throws

oneapi::mkl::invalid_argument Exception is thrown when min == nullptr, or max == nullptr, or dataset object is invalid

Return Value

Output event to wait on to ensure computation is complete.

Parent topic: *Summary Statistics Routines*

Service Routines

Routine	Description
make_dataset	Creates a dataset from the provided parameters

Parent topic: *Summary Statistics*

make_dataset

Entry point to create a dataset from the provided parameters.

Description and Assumptions

The oneapi::mkl::stats::make_dataset function is used to create a dataset from the provided storage of the observations matrix, the number of dimensions and observations, and other parameters.

make_dataset supports the following precisions for data:

T
float
double

make_dataset (buffer version)

Syntax

```
namespace oneapi::mkl::stats {
template<layout ObservationsLayout = layout::row_major, typename Type>
    dataset<sycl::buffer<Type, 1>, ObservationsLayout> make_dataset(
    std::int64_t n_dims,
    std::int64_t n_observations,
    sycl::buffer<Type, 1> observations,
    sycl::buffer<Type, 1> weights = {0},
    sycl::buffer<std::int64_t, 1> indices = {0});
}
```

Template Parameters

ObservationsLayout Data layout. The specific values are described in *dataset*.

Type Data precision.

Input Parameters

n_dims The number of dimensions.

n_observations The number of observations.

observations Matrix of observations.

weights Optional parameter. Array of weights of size n_observations. Elements of the array are non-negative members. If the parameter is not specified, each observation has weight equal to 1.

indices Optional parameter. Array of vector components that are processed. The size of the array is n_dims. If the parameter is not specified, all components are processed.

Throws

oneapi::mkl::invalid_argument Exception is thrown when :math: *n_dims leq 0*, or :math: *n_observations leq 0*, or observations.get_count() == 0

Return Value

Dataset holding specified parameters.

make_dataset (USM version)

Syntax

```
namespace oneapi::mkl::stats {
template<layout ObservationsLayout = layout::row_major, typename Type>
    dataset<Type*, ObservationsLayout> make_dataset(std::nt64_t
    n_dims, std::int64_t n_observations,
    Type* observations, Type* weights = nullptr, std::int64_t* indices = nullptr);
}
```

Template Parameters

ObservationsLayout Data layout. The specific values are described in *dataset*.

Type Data precision.

Input Parameters

n_dims The number of dimensions.

n_observations The number of observations.

observations Matrix of observations.

weights Optional parameter. Array of weights of size n_observations. Elements of the array are non-negative members. If the parameter is not specified, each observation has weight equal to 1.

indices Optional parameter. Array of vector components that are processed. Size of array is n_dims. If the parameter is not specified, all components are processed.

Throws

oneapi::mkl::invalid_argument Exception is thrown when $n_dims \leq 0$, or $n_observations \leq 0$, or observations == nullptr

Return Value

Dataset holding specified parameters.

Parent topic: *Service Routines*

12.2.6 Vector Math

oneMKL Vector Mathematics functions (VM) compute a mathematical function of each of the vector elements. VM includes a set of functions (arithmetic, power, trigonometric, exponential, hyperbolic, special, and rounding) that operate on vectors of real and complex numbers.

Application programs that improve performance with VM include nonlinear programming software, computation of integrals, financial calculations, computer graphics, and many others.

VM functions fall into the following groups according to the operations they perform:

- *VM Mathematical Functions* compute values of mathematical functions, such as sine, cosine, exponential, or logarithm, on vectors stored contiguously in memory.
- *VM Service Functions* set/get the accuracy modes and the error codes, and create error handlers for mathematical functions.

The VM mathematical functions take an input vector as an argument, compute values of the respective function element-wise, and return the results in an output vector. All the VM mathematical functions can perform in-place operations, where the input and output arrays are at the same memory locations.

- *Special Value Notations*

Special Value Notations

This defines notations of special values for complex functions. The definitions are provided in text, tables, or formulas.

- z, z1, z2, etc. denote complex numbers.
- i, `i2=-1` is the imaginary unit.
- x, X, x1, x2, etc. denote real imaginary parts.
- y, Y, y1, y2, etc. denote imaginary parts.
- **X and Y represent any finite positive IEEE-754 floating point** values, if not stated otherwise.
- **Quiet NaN and signaling NaN are denoted with QNAN and SNAN,** respectively.
- **The IEEE-754 positive infinities or floating-point numbers are** denoted with a + sign before X, Y, etc.
- **The IEEE-754 negative infinities or floating-point numbers are** denoted with a − sign before X, Y, etc.

`CONJ(z)` and `CIS(z)` are defined as follows:

`CONJ(x+i·y)=x-i·y`

`CIS(y)=cos(y)+i·sin(y).`

The special value tables show the result of the function for the z argument at the intersection of the RE(z) column and the i*IM(z) row. If the function encounters a special point for the argument z, the lower part of this cell shows the special point and the VM status value for this element. An empty cell indicates that this argument is normal and the result is well-defined computationally.

Parent topic: *Vector Math*

VM Mathematical Functions

This section describes VM functions that compute values of mathematical functions on real and complex vector arguments with unit increment.

Each function is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data, as well as a description of the input/output arguments.

The input range of parameters is equal to the mathematical range of the input data type, unless the function description specifies input threshold values, which mark off the precision overflow, as follows:

- `FLT_MAX` denotes the maximum number representable in single precision real data type
- `DBL_MAX` denotes the maximum number representable in double precision real data type

The following tables list the available mathematical functions grouped by category.

Arithmetic Routines	Description
add	Adds vector elements
sub	Subtracts vector elements
sqr	Squares vector elements
mul	Multiplies vector elements
mulbyconj	Multiplies elements of one vector by conjugated elements of the second vector
conj	Conjugates vector elements
abs	Computes the absolute value of vector elements
arg	Computes the argument of vector elements
linearfrac	Performs linear fraction transformation of vectors

continues on next page

Table 7 – continued from previous page

Arithmetic Routines	Description
fmod	Performs element by element computation of the modulus function of vector `a` with respect to vector `b`
remainder	Performs element by element computation of the remainder function on the elements of vector `a` and the corresponding elements of vector `b`

Power and Root Routines	Description
inv	Inverts vector elements
div	Divides elements of one vector by elements of the second vector
sqrt	Computes the square root of vector elements
invsqrt	Computes the inverse square root of vector elements
cbrt	Computes the cube root of vector elements
invcbrt	Computes the inverse cube root of vector elements
pow2o3	Computes the cube root of the square of each vector element
pow3o2	Computes the square root of the cube of each vector element
pow	Raises each vector element to the specified power
powx	Raises each vector element to the constant power
powr	Computes `a` to the power `b` for elements of two vectors, where the elements of vector argument `a` are all non-negative
hypot	Computes the square root of sum of squares

Exponential and Logarithmic Routines	Description
exp	Computes the base `e` exponential of vector elements
exp2	Computes the base 2 exponential of vector elements
exp10	Computes the base 10 exponential of vector elements
expm1	Computes the base `e` exponential of vector elements decreased by 1
ln	Computes the natural logarithm of vector elements
log2	Computes the base 2 logarithm of vector elements
log10	Computes the base 10 logarithm of vector elements
log1p	Computes the natural logarithm of vector elements that are increased by 1
logb	Computes the exponents of the elements of input vector `a`

Trigonometric Routines	Description
cos	Computes the cosine of vector elements
sin	Computes the sine of vector elements
sincos	Computes the sine and cosine of vector elements
cis	Computes the complex exponent of vector elements (cosine and sine combined to complex value)
tan	Computes the tangent of vector elements
acos	Computes the inverse cosine of vector elements
asin	Computes the inverse sine of vector elements
atan	Computes the inverse tangent of vector elements
atan2	Computes the four-quadrant inverse tangent of ratios of the elements of two vectors
cospi	Computes the cosine of vector elements multiplied by π
sinpi	Computes the sine of vector elements multiplied by π

continues on next page

Table 10 – continued from previous page

Trigonometric Routines	Description
tanpi	Computes the tangent of vector elements multiplied by π
acospi	Computes the inverse cosine of vector elements divided by π
asinpi	Computes the inverse sine of vector elements divided by π
atanpi	Computes the inverse tangent of vector elements divided by π
atan2pi	Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by π
cosd	Computes the cosine of vector elements multiplied by $\pi/180$
sind	Computes the sine of vector elements multiplied by $\pi/180$
tand	Computes the tangent of vector elements multiplied by $\pi/180$

Hyperbolic Routines	Description
cosh	Computes the hyperbolic cosine of vector elements
sinh	Computes the hyperbolic sine of vector elements
tanh	Computes the hyperbolic tangent of vector elements
acosh	Computes the inverse hyperbolic cosine of vector elements
asinh	Computes the inverse hyperbolic sine of vector elements
atanh	Computes the inverse hyperbolic tangent of vector elements.

Special Routines	Description
erf	Computes the error function value of vector elements
erfc	Computes the complementary error function value of vector elements
cdfnorm	Computes the cumulative normal distribution function value of vector elements
erfinv	Computes the inverse error function value of vector elements
erfcinv	Computes the inverse complementary error function value of vector elements
cdfnorminv	Computes the inverse cumulative normal distribution function value of vector elements
lgamma	Computes the natural logarithm for the absolute value of the gamma function of vector elements
tgamma	Computes the gamma function of vector elements
expint1	Computes the exponential integral of vector elements

Rounding Routines	Description
floor	Rounds towards minus infinity
ceil	Rounds towards plus infinity
trunc	Rounds towards zero infinity
round	Rounds to nearest integer
nearbyint	Rounds according to current mode
rint	Rounds according to current mode and reports inexact result status
modf	Computes the integer and fractional parts
frac	Computes the fractional part

Miscellaneous Routines	Description
copysign	Returns vector of elements of one argument with signs changed to match other argument elements
nextafter	Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector
fdim	Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise
fmax	Returns the larger of each pair of elements of the two vector arguments
fmin	Returns the smaller of each pair of elements of the two vector arguments
maxmag	Returns the element with the larger magnitude between each pair of elements of the two vector arguments
minmag	Returns the element with the smaller magnitude between each pair of elements of the two vector arguments

Parent topic: *Vector Math*

abs

Computes absolute value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event abs(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<R,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event abs(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        R* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`abs` supports the following precisions.

T	R
`float`	`float`
`double`	`double`
`std::complex<float>`	`float`
`std::complex<double>`	`double`

Description

The abs(a) function computes an absolute value of vector elements.

Argument	Result	Status code
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`abs(a) = hypot(RE(a), IM(a))`.

The abs function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

acos

Computes inverse cosine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event acos(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event acos(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`acos` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The acos(a) function computes inverse cosine of vector elements.

Argument	Result	Status code
+0	+π/2	
-0	+π/2	
+1	+0	
-1	+π	
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+3·π/4-i·∞	+π/2-i·∞	+π/2-i·∞	+π/2-i·∞	+π/2-i·∞	+π/4-i·∞	QNAN-i·∞
+i·Y	+π-i·∞					+0-i·∞	QNAN+i·QNAN
+i·0	+π-i·∞		+π/2-i·0	+π/2-i·0		+0-i·∞	QNAN+i·QNAN
-i·0	+π+i·∞		+π/2+i·0	+π/2+i·0		+0+i·∞	QNAN+i·QNAN
-i·Y	+π+i·∞					+0+i·∞	QNAN+i·QNAN
-i·∞	+3π/4+i·∞	+π/2+i·∞	+π/2+i·∞	+π/2+i·∞	+π/2+i·∞	+π/4+i·∞	QNAN+i·∞
+i·NAN	QNAN+i·∞	QNAN+i·QNAN	+π/2+i·QNAN	+π/2+i·QNAN	QNAN+i·QNAN	QNAN+i·∞	QNAN+i·QNAN

Notes:

- `acos(CONJ(a))=CONJ(acos(a))`.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event acosh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event acosh(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`acosh` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The acosh(a) function computes inverse hyperbolic cosine (nonnegative) of vector elements.

Argument	Result	Status code
+1	+0	
a < +1	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i \cdot \frac{3\pi}{4}$	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	+∞+i·π					+∞+i·0	QNAN+i·QNAN
+i·0	+∞+i·π		+0+i·π/2	+0+i·π/2		+∞+i·0	QNAN+i·QNAN
-i·0	+∞+i·π		+0+i·π/2	+0+i·π/2		+∞+i·0	QNAN+i·QNAN
-i·Y	+∞+i·π					+∞+i·0	QNAN+i·QNAN
-i·∞	$+\infty - i \cdot \frac{3\pi}{4}$	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	+∞+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- `acosh(CONJ(a))=CONJ(acosh(a))`.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

acospi

Computes the inverse cosine of vector elements divided by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event acospi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event acospi(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`acospi` supports the following precisions.

T
float
double

Description

The acospi(a) function computes the inverse cosine of vector elements divided by π. For an argument `a`, the function computes acos(`a`)/π.

Argument	Result	Status code
+0	+1/2	
-0	+1/2	
+1	+0	
-1	+1	
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
$+\infty$	QNAN	`oneapi::mkl::vm::status::errdom`
• ∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

add

Performs element by element addition of vector `a` and vector `b`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event add(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event add(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`add` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The add(a, b) function performs element by element addition of vector `a` and vector `b`.

Argument 1	Argument 2	Result	Status code
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
+∞	+∞	+∞	
+∞	-∞	QNAN	
-∞	+∞	QNAN	
-∞	-∞	-∞	
SNAN	any value	QNAN	
any value	SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

```
add(x1+i*y1, x2+i*y2) = (x1+x2) + i*(y1+y2)
```

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow` (overriding any possible `oneapi::mkl::vm::status::accuracy_warning` status).

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

arg

Computes argument of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event arg(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<R,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event arg(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        R* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`arg` supports the following precisions.

T	R
`std::complex<float>`	`float`
`std::complex<double>`	`double`

Description

The arg(a) function computes argument of vector elements.

See *Special Value Notations* for the conventions used in the table below.

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+3·π/4	+π/2	+π/2	+π/2	+π/2	+π/4	NAN
+i·Y	+π		+π/2	+π/2		+0	NAN
+i·0	+π	+π	+π	+0	+0	+0	NAN
-i·0	-π	-π	-π	-0	-0	-0	NAN
-i·Y	-π		-π/2	-π/2		-0	NAN
-i·∞	-3·π/4	-π/2	-π/2	-π/2	-π/2	-π/4	NAN
+i·NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

Note

`arg(a)=atan2(IM(a), RE(a))`

The arg function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

asin

Computes inverse sine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event asin(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event asin(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`asin` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The asin(a) function computes inverse sine of vector elements.

Argument	Result	Status code
+0	+0	
-0	-0	
+1	+π/2	
-1	-π/2	
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`asin(a) = -i*asinh(i*z).`

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

asinh

Computes inverse hyperbolic sine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event asinh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event asinh(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`asinh` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The asinh(a) function computes inverse hyperbolic sine of vector elements.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-∞+i·π/4	-∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	-∞+i·0					+∞+i·0	QNAN+i·QNAN
+i·0	+∞+i·0		+0+i·0	+0+i·0		+∞+i·0	QNAN+i·QNAN
-i·0	-∞-i·0		-0-i·0	+0-i·0		+∞-i·0	QNAN- i·QNAN
-i·Y	-∞-i·0					+∞-i·0	QNAN+i·QNAN
-i·∞	-∞-i·π/4	-∞-i·π/2	-∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	- ∞+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

The asinh(a) function does not generate any errors.

Notes:

- `asinh(CONJ(a))=CONJ(asinh(a))`
- `asinh(-a)=-asinh(a)`.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

asinpi

Computes the inverse sine of vector elements divided by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event asinpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event asinpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`asinpi` supports the following precisions.

T
`float`
`double`

Description

The asinpi(a) function computes the inverse sine of vector elements divided by π. For an argument `a`, the function computes asinpi(`a`)/π.

Argument	Result	Status code
+0	+0	
-0	-0	
+1	+1/2	
-1	-1/2	
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
$+\infty$	QNAN	`oneapi::mkl::vm::status::errdom`
$-\infty$	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atan

Computes inverse tangent of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atan(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atan(
        sycl::queue& exec_queue,
```

(continues on next page)

(continued from previous page)

```
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`atan` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The atan(a) function computes inverse tangent of vector elements.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+π/2	
-∞	-π/2	
QNAN	QNAN	
SNAN	QNAN	

The atan function does not generate any errors.

Specifications for special values of the complex functions are defined according to the following formula

`atan(a) = -i*atanh(i*a).`

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atan2(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atan2(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
```

(continues on next page)

(continued from previous page)

```
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`ad2d` supports the following precisions.

T
`float`
`double`

Description

The atan2(a, b) function computes four-quadrant inverse tangent of elements of two vectors.

The elements of the output vector are computed as the four-quadrant arctangent of `a[i] / b[i]`.

Argument 1	Argument 2	Result	Status code
-∞	-∞	-3*π/4	
-∞	b < +0	-π/2	
-∞	-0	-π/2	
-∞	+0	-π/2	
-∞	b > +0	-π/2	
-∞	+∞	-π/4	
a < +0	-∞	-π	
a < +0	-0	-π/2	
a < +0	+0	-π/2	
a < +0	+∞	-0	
-0	-∞	-π	
-0	b < +0	-π	
-0	-0	-π	
-0	+0	-0	
-0	b > +0	-0	
-0	+∞	-0	
+0	-∞	+π	
+0	b < +0	+π	
+0	-0	+π	
+0	+0	+0	
+0	b > +0	+0	
+0	+∞	+0	
a > +0	-∞	+π	
a > +0	-0	+π/2	
a > +0	+0	+π/2	
a > +0	+∞	+0	
+∞	-∞	+3*π/4	
+∞	b < +0	+π/2	
+∞	-0	+π/2	
+∞	+0	+π/2	
+∞	b > +0	+π/2	
+∞	+∞	+π/4	
a > +0	QNAN	QNAN	

continues on next page

Table 15 – continued from previous page

Argument 1	Argument 2	Result	Status code
a > +0	SNAN	QNAN	
QNAN	b > +0	QNAN	
SNAN	b > +0	QNAN	
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	
SNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

The atan2(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atan2pi

Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atan2pi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atan2pi(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`atan2pi` supports the following precisions.

T
float
double

Description

The atan2pi(a, b) function computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by π.

For the elements of the output vector y, the function computers the four-quadrant arctangent of a_i/b_i, with the result divided by π.

Argument 1	Argument 2	Result	Status code
-∞	-∞	-3/4	
-∞	b < +0	-1/2	
-∞	-0	+1/2	
-∞	+0	-1/2	
-∞	x > +0	-1/2	
-∞	+∞	-1/4	
a < +0	-∞	-1	
a < +0	-0	-1/2	
a < +0	+0	-1/2	
a < +0	+∞	-0	
-0	-∞	-1	
-0	b < +0	-1	
-0	-0	-1	
-0	+0	-0	
-0	b > +0	-0	
-0	+∞	-0	
+0	-∞	+1	
+0	b < +0	+1	
+0	-0	+1	
+0	+0	+0	
+0	b > +0	+0	
+0	+∞	+0	
a > +0	-∞	+1	
a > +0	-0	+1/2	
x > +0	+0	+1/2	
a > +0	+∞	+1/4	
+∞	-∞	+3/4	
+∞	b < +0	+1/2	
+∞	-0	+1/2	
+∞	+0	+1/2	
+∞	b > +0	+1/2	
+∞	+∞	+1/4	
a > +0	QNAN	QNAN	
a > +0	SNAN	QNAN	
QNAN	b > +0	QNAN	
SNAN	x > +0	QNAN	
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	
SNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

The atan2pi(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atanh

Computes inverse hyperbolic tangent of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atanh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atanh(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`atanh` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The atanh(a) function computes inverse hyperbolic tangent of vector elements.

Argument	Result	Status code
+1	+∞	`oneapi::mkl::vm::status::sing`
-1	-∞	`oneapi::mkl::vm::status::sing`
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-0+i·π/2	-0+i·π/2	-0+i·π/2	+0+i·π/2	+0+i·π/2	+0+i·π/2	+0+i·π/2
+i·Y	-0+i·π/2					+0+i·π/2	QNAN+i·QNAN
+i·0	-0+i·π/2		-0+i·0	+0+i·0		+0+i·π/2	QNAN+i·QNAN
-i·0	-0-i·π/2		-0-i·0	+0-i·0		+0-i·π/2	QNAN- i·QNAN
-i·Y	-0-i·π/2					+0-i·π/2	QNAN+i·QNAN
-i·∞	-0-i·π/2	-0-i·π/2	-0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2
+i·NAN	- 0+i·QNAN	QNAN+i·QNAN	- 0+i·QNAN	+0+i·QNAN	QNAN+i·QNAN	+0+i·QNAN	QNAN+i·QNAN

Notes:

- `atanh(±1±i*0)=±∞±i*0`, and oneapi::mkl::vm::status::sing error is generated
- `atanh(CONJ(a))=CONJ(atanh(a))`
- `atanh(-a)=-atanh(a)`.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

atanpi

Computes the inverse tangent of vector elements divided by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event atanpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event atanpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`atanpi` supports the following precisions.

T
float
double

Description

The atanpi(a) function computes the inverse tangent of vector elements divided by π. For an argument `a`, the function computes atan(`a`)/π.

Argument	Result	Status code
+0	+0	
-0	-0	
$+\infty$	+1/2	
$-\infty$	-1/2	
QNAN	QNAN	
SNAN	QNAN	

The atanpi function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cbrt

Computes a cube root of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cbrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cbrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`cbrt` supports the following precisions.

T
float
double

Description

The cbrt(a)function computes a cube root of vector elements.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	
+0	+0	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cdfnorm

Computes the cumulative normal distribution function values of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cdfnorm(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cdfnorm(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cdfnorm` supports the following precisions.

T
float
double

Description

The cdfnorm function computes the cumulative normal distribution function values for elements of the input vector `a` and writes them to the output vector `y`.

The cumulative normal distribution function is defined as given by:

$$\mathrm{cdfnorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} \, \mathrm{d}x$$

Useful relations for these functions:

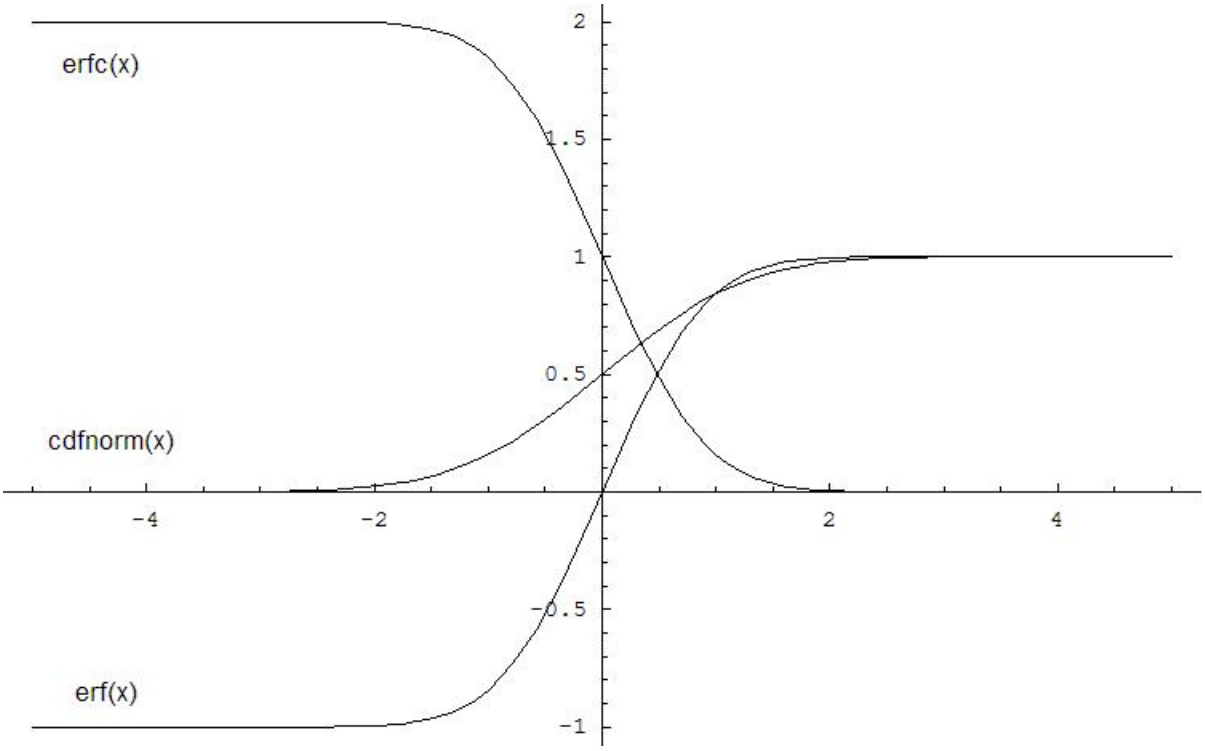
$$\mathrm{erf}(x) + \mathrm{erfc}(x) = 1$$

$$\begin{aligned} \mathrm{cdfnorm}(x) &= \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \\ &= 1 - \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right) \end{aligned}$$

where erf and erfc are the error and complementary error functions, respectively.

The following figure illustrates the relationships among the family of error functions (erf, erfc, cdfnorm).

cdfnorm Family Functions Relationship |



Argument	Result	Status code
a < underflow	+0	`oneapi::mkl::vm::status::underflow`
+∞	+1	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cdfnorminv

Computes the inverse cumulative normal distribution function values of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cdfnorminv(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cdfnorminv(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cdfnorminv` supports the following precisions.

T
float
double

Description

The cdfnorminv(a) function computes the inverse cumulative normal distribution function values for elements of the input vector `a` and writes them to the output vector `y`.

The inverse cumulative normal distribution function is defined as given by:

$$\mathrm{cdfnorminv}(x) = \mathrm{cdfnorm}^{-1}(x)$$

where $\mathrm{cdfnorm}(x)$ denotes the cumulative normal distribution function.

Useful relations:

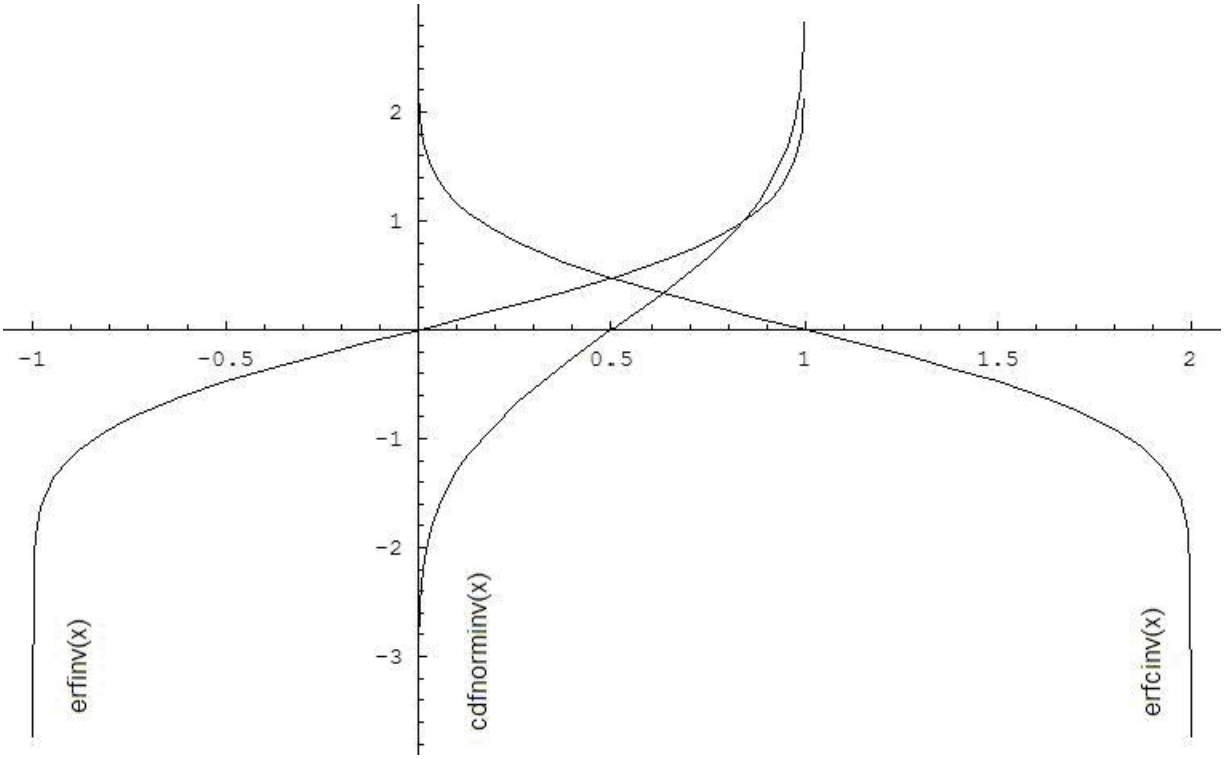
$$\mathrm{erfcinv}(x) = \mathrm{erfinv}(1 - x)$$

$$\begin{aligned}\mathrm{cdfnorminv}(x) &= \sqrt{2}\,\mathrm{erfinv}(2x-1)\\ &= \sqrt{2}\,\mathrm{erfcinv}(2-2x)\end{aligned}$$

where $\mathrm{erfinv}(x)$ denotes the inverse error function and $\mathrm{erfcinv}(x)$ denotes the inverse complementary error function.

The following figure illustrates the relationships among erfinv family functions (erfinv, erfcinv, cdfnorminv).

cdfnorminv Family Functions Relationship |



Argument	Result	Status code
+0.5	+0	
+1	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
+0	-∞	`oneapi::mkl::vm::status::sing`
a < -0	QNAN	`oneapi::mkl::vm::status::errdom`
a > +1	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

ceil

Computes an integer value rounded towards plus infinity for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event ceil(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event ceil(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`ceil` supports the following precisions.

T
`float`
`double`

Description

The ceil(a) function computes an integer value rounded towards plus infinity for each vector element.

$$y_i = \lceil a_i \rceil$$

Argument	Result	Status code
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
QNAN	QNAN	
SNAN	QNAN	

The ceil function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cis

Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cis(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<R,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cis(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        R* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cis` supports the following precisions.

T	R
`float`	`std::complex<float>`
`double`	`std::complex<double>`

Description

The cis(a) function computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Argument	Result	Status code
• 0	+1+i·0	
• 0	+1-i·0	
• ∞	QNAN+i·QNAN	`oneapi::mkl::vm::status::errdom`
• ∞	QNAN+i·QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN+i·QNAN	
SNAN	QNAN+i·QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

conj

Performs element by element conjugation of the vector.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event conj(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event conj(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`conj` supports the following precisions.

T
std::complex<float>
std::complex<double>

Description

The conj function performs element by element conjugation of the vector.

No special values are specified. The conj function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

copysign

Returns vector of elements of one argument with signs changed to match other argument elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event copysign(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event copysign(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`copysign` supports the following precisions.

T
float
double

Description

The copysign(a, b) function returns the first vector argument elements with the sign changed to match the sign of the second vector argument's corresponding elements.

Argument 1	Argument 2	Result	Status code
any value	positive value	+any value	
any value	negative value	-any value	

The copysign(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cos

Computes cosine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cos(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cos(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cos` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The cos(a) function computes cosine of vector elements.

Note that arguments `abs(a[i]) ≤ 213` and `abs(a[i]) ≤ 216` for single and double precisions, respectively, are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result	VM status code
+0	+1	
-0	+1	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`Cos(z) = Cosh(i*z).`

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cosd

Computes the cosine of vector elements multiplied by $\pi/180$.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cosd(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cosd(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cosd` supports the following precisions.

T
float
double

Description

The cosd(a) function is a degree argument trigonometric function. It computes the cosine of vector elements multiplied by $\pi/180$. For an argument `a`, the function computes cos(π*a/180).

Note that arguments abs(a_i) $\leq 2^{24}$ for single precision or abs(a_i) $\leq 2^{52}$ for double precision, they belong to the *fast computational path*: trigonometric function arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Argument	Result	Status code
+0	+1	
-0	+1	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cosh

Computes hyperbolic cosine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cosh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cosh(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cosh` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The cosh(a) function computes hyperbolic cosine of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	`-Log(FLT_MAX)-Log2 <a[i] < Log(FLT_MAX)+Log2`
double precision	`-Log(DBL_MAX)-Log2 <a[i] < Log(DBL_MAX)+Log2`

Argument	Result	Status code
+0	+1	
-0	+1	
X > overflow	+∞	`oneapi::mkl::vm::status::overflow`
X < -overflow	+∞	`oneapi::mkl::vm::status::overflow`
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	+∞+i·QNAN	QNAN+i·QNAN	QNAN-i·0	QNAN+i·0	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN
+i·Y	+∞·Cos(Y)-i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
+i·0	+∞-i·0		+1-i·0	+1+i·0		+∞+i·0	QNAN+i·0
-i·0	+∞+i·0		+1+i·0	+1-i·0		+∞-i·0	QNAN-i·0
-i·Y	+∞·Cos(Y)-i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
-i·∞	+∞+i·QNAN	QNAN+i·QNAN	QNAN+i·0	QNAN-i·0	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN
+i·NAN	+∞+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN-i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- **The complex cosh(a) function sets the VM status code to** `oneapi::mkl::vm::status::overflow` in the case of overflow, that is, when RE(a), IM(a) are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- `cosh(CONJ(a))=CONJ(cosh(a))`
- `cosh(-a)=cosh(a).`

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

cospi

Computes the cosine of vector elements multiplied by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event cospi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event cospi(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`cospi` supports the following precisions.:

T
float
double

Description

The cospi(a) function computes the cosine of vector elements multiplied by π. For an argument `a`, the function computes cos(π*a).

Argument	Result	Status code
+0	+1	
-0	+1	
n + 0.5, for any integer n where n + 0.5 is representable	+0	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

If arguments abs(a_i) $\leq 2^{22}$ for single precision or abs(a_i) $\leq 2^{51}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

div

Performs element by element division of vector `a` by vector `b`

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event div(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event div(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`div` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The div(a, b) function performs element by element division of vector `a` by vector `b`.

Argument 1	Argument 2	Result	VM status code
X > +0	+0	+∞	`oneapi::mkl::vm::status::sing`
X > +0	-0	-∞	`oneapi::mkl::vm::status::sing`
X < +0	+0	-∞	`oneapi::mkl::vm::status::sing`
X < +0	-0	+∞	`oneapi::mkl::vm::status::sing`
+0	+0	QNAN	`oneapi::mkl::vm::status::sing`
-0	-0	QNAN	`oneapi::mkl::vm::status::sing`
X > +0	+∞	+0	
X > +0	-∞	-0	
+∞	+∞	QNAN	
-∞	-∞	QNAN	
QNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`Div(x1+i*y1, x2+i*y2) = (x1+i*y1)*(x2-i*y2)/(x2*x2+y2*y2).`

Overflow in a complex function occurs when `x2+i*y2` is not zero, x1, x2, y1, y2 are finite numbers, but the real or imaginary part of the exact result is so large that it does not fit the target precision. In that case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow`.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

erf

Computes the error function value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event erf(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event erf(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`erf` supports the following precisions.

T
float
double

Description

The erf function computes the error function values for elements of the input vector `a` and writes them to the output vector `y`.

The error function is defined as given by:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, \mathrm{d}t$$

Useful relations:

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$$

where erfc is the complementary error function.

$$\Phi(x) = \frac{1}{2} \left(1 + \operatorname{erf} \left(x/\sqrt{2} \right) \right)$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) \, \mathrm{d}t$$

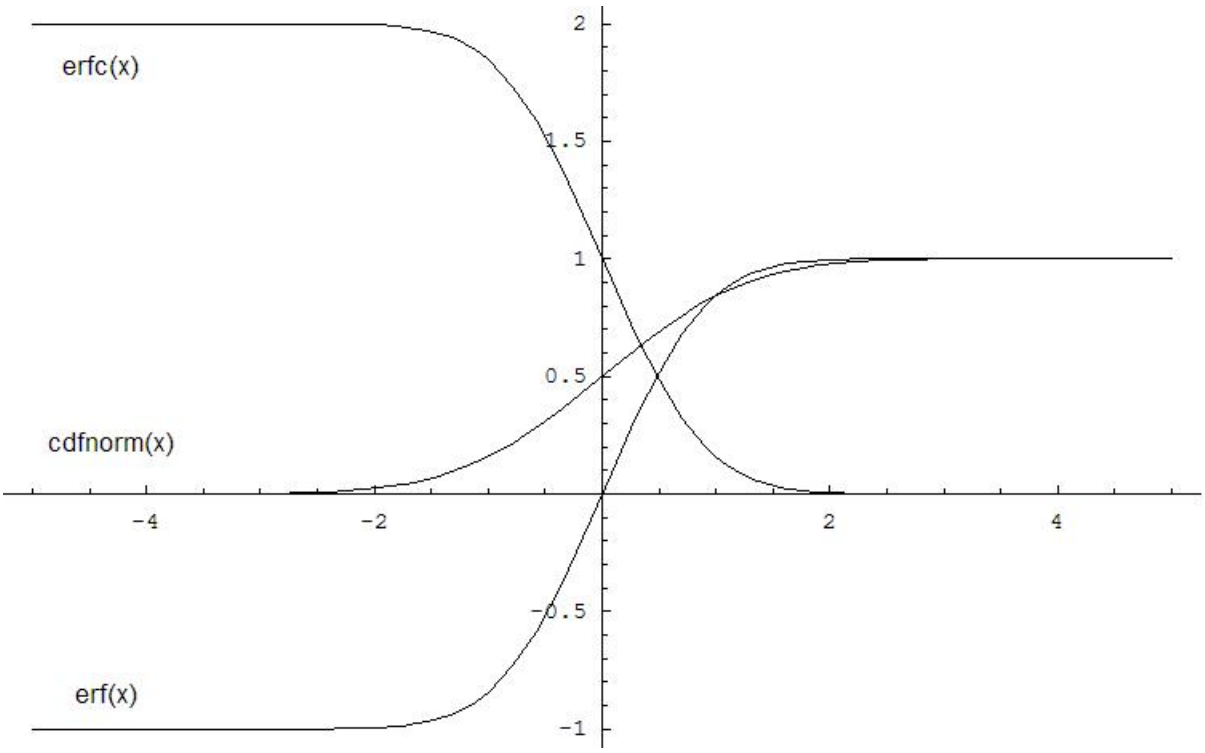
is the cumulative normal distribution function.

$$\Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1)$$

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$, respectively.

The following figure illustrates the relationships among erf family functions (erf, erfc, cdfnorm).

erf Family Functions Relationship |



Useful relations for these functions:

$$\mathrm{erf}(x) + \mathrm{erfc}(x) = 1$$

$$\begin{aligned}\mathrm{cdfnorm}(x) &= \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \\ &= 1 - \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right)\end{aligned}$$

Argument	Result	Status code
+∞	+1	
-∞	-1	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

erfc

Computes the complementary error function value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event erfc(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event erfc(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`erfc` supports the following precisions.

T
float
double

Description

The erfc function computes the complementary error function values for elements of the input vector `a` and writes them to the output vector `y`.

The complementary error function is defined as follows:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\,\mathrm{d}t$$

Useful relations:

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$$

$$\Phi(x) = \frac{1}{2}\left(1 + \operatorname{erf}\left(x/\sqrt{2}\right)\right)$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2)\,\mathrm{d}t$$

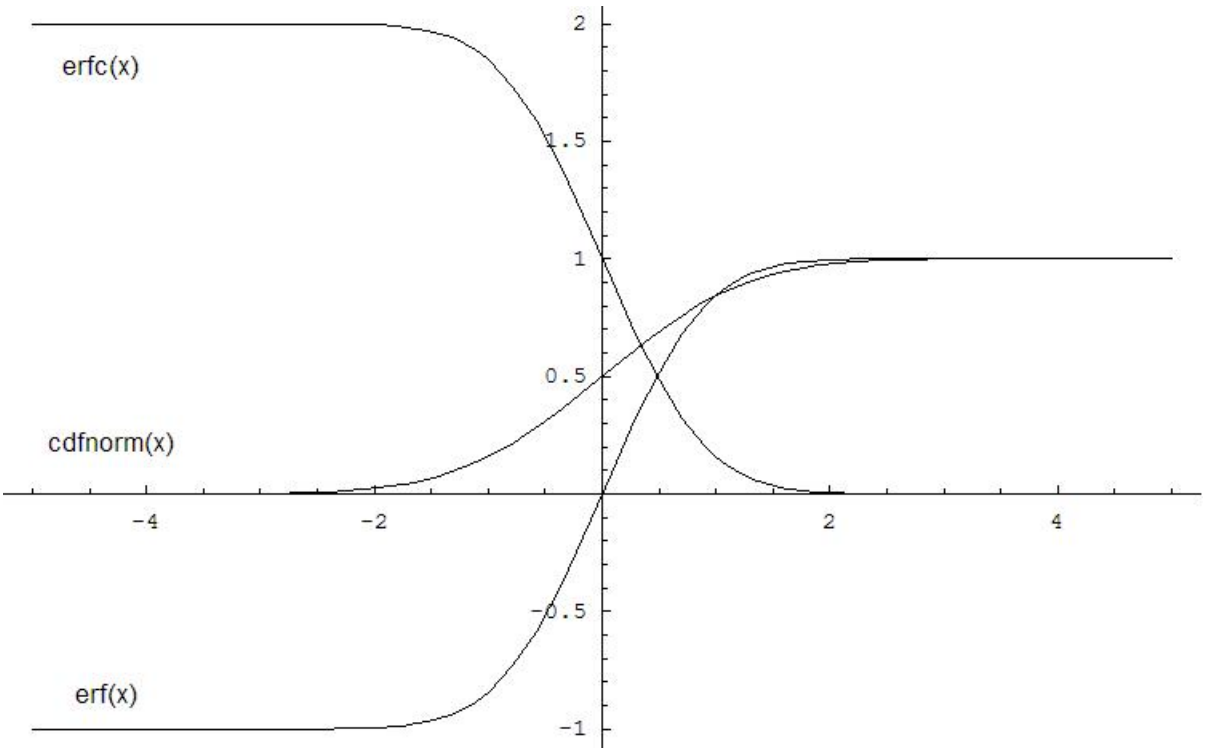
is the cumulative normal distribution function.

$$\Phi^{-1}(x) = \sqrt{2}\,\operatorname{erf}^{-1}(2x - 1)$$

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$, respectively.

The following figure illustrates the relationships among erf family functions (erf, erfc, cdfnorm).

erfc Family Functions Relationship |



Useful relations for these functions:

$$\mathrm{erf}(x) + \mathrm{erfc}(x) = 1$$

$$\begin{aligned}\mathrm{cdfnorm}(x) &= \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \\ &= 1 - \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right)\end{aligned}$$

Argument	Result	Status code
a > underflow	+0	`oneapi::mkl::vm::status::underflow`
+∞	+0	
-∞	+2	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

erfcinv

Computes the inverse complementary error function value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event erfcinv(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event erfcinv(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`erfcinv` supports the following precisions.

T
float
double

Description

The erfcinv(a) function computes the inverse complimentary error function values for elements of the input vector `a` and writes them to the output vector `y`.

The inverse complementary error function is defined as given by:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

Useful relations for these functions:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

$$\begin{aligned}\text{cdfnorminv}(x) &= \sqrt{2}\,\text{erfinv}(2x - 1) \\ &= \sqrt{2}\,\text{erfcinv}(2 - 2x)\end{aligned}$$

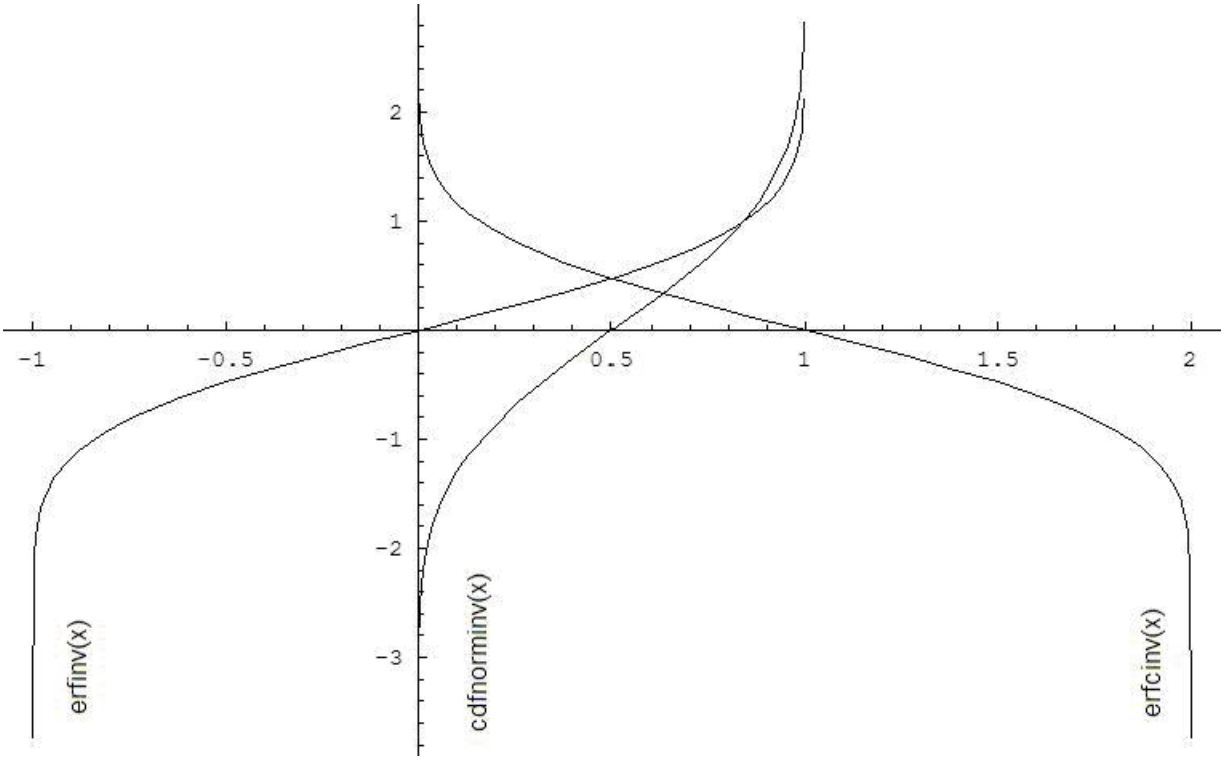
$$\text{erfinv}(x) = \text{erf}^{-1}(x)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,\mathrm{d}t$$

where $\text{erf}(x)$ denotes the error function and $\text{erfinv}(x)$ denotes the inverse error function.

The following figure illustrates the relationships among erfinv family functions (erfinv, erfcinv, cdfnorminv).

erfcinv Family Functions Relationship |



Argument	Result	Status code
+1	+0	
+2	-∞	`oneapi::mkl::vm::status::sing`
-0	+∞	`oneapi::mkl::vm::status::sing`
+0	+∞	`oneapi::mkl::vm::status::sing`
a < -0	QNAN	`oneapi::mkl::vm::status::errdom`
a > +2	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

erfinv

Computes inverse error function value of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event erfinv(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event erfinv(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`erfinv` supports the following precisions.

T
`float`
`double`

Description

The erfinv(a) function computes the inverse error function values for elements of the input vector `a` and writes them to the output vector `y`.

$$y_i = \mathrm{erf}^{-1}(a)$$

where $\mathrm{erf}(x)$ is the error function defined as given by:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,\mathrm{d}t$$

Useful relations for these functions:

$$\mathrm{erfcinv}(x) = \mathrm{erfinv}(1-x)$$

$$\begin{aligned}\mathrm{cdfnorminv}(x) &= \sqrt{2}\,\mathrm{erfinv}(2x-1)\\ &= \sqrt{2}\,\mathrm{erfcinv}(2-2x)\end{aligned}$$

$$\mathrm{erf}^{-1}(x) = \mathrm{erfc}^{-1}(1-x)$$

where erfc is the complementary error function.

$$\Phi(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(x/\sqrt{2}\right)\right)$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2)\,\mathrm{d}t$$

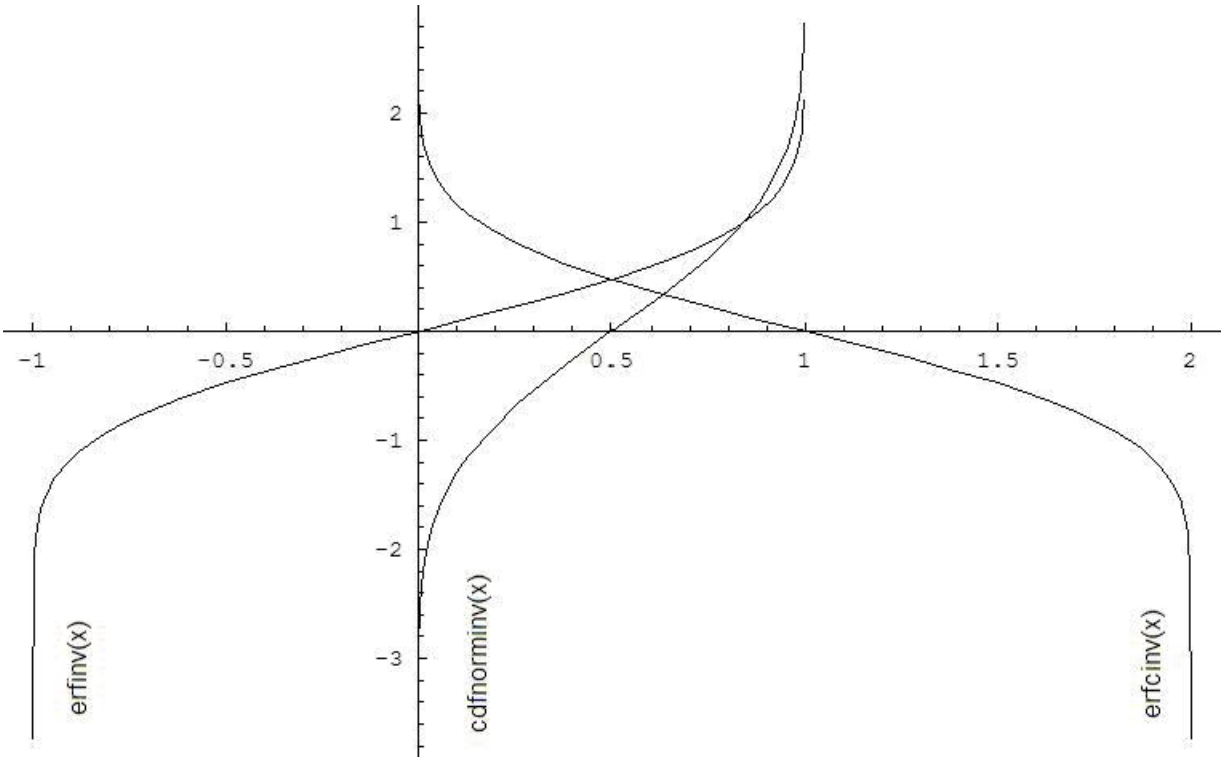
is the cumulative normal distribution function.

$$\Phi^{-1}(x) = \sqrt{2}\,\mathrm{erf}^{-1}(2x-1)$$

where $\Phi^{-1}(x)$ and $\mathrm{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\mathrm{erf}(x)$, respectively.

The following figure illustrates the relationships among erfinv family functions (erfinv, erfcinv, cdfnorminv).

erfinv Family Functions Relationship |



Argument	Result	Status code
+0	+0	
-0	-0	
+1	+∞	`oneapi::mkl::vm::status::sing`
-1	-∞	`oneapi::mkl::vm::status::sing`
|a| > 1	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

exp

Computes an exponential of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event exp(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event exp(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`exp` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The exp(a) function computes an exponential of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	`a[i] < Log( FLT_MAX )`
double precision	`a[i] < Log( DBL_MAX )`

Argument	Result	Status code
+0	+1	
-0	+1	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
a < underflow	+0	`oneapi::mkl::vm::status::overflow`
+∞	+∞	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

+i·∞							
+i·Y							
+i·0							
-i·0							
-i·Y							
-i·∞							
+i·NAN							

Notes:

- **The complex exp(z) function sets the VM status code to** `oneapi::mkl::vm::status::overflow` in the case of overflow, that is, when both `RE(z)` and `IM(z)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

exp10

Computes the base 10 exponential of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event exp10(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event exp10(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`exp10` supports the following precisions.

T
float
double

Description

The exp10(a) function computes the base 10 exponential of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_{10}(\text{FLT\_MAX})$
double precision	$a_i < \log_{10}(\text{DBL\_MAX})$

Argument	Result	VM status code
+0	+1	
-0	+1	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
a < underflow	+0	`oneapi::mkl::vm::status::underflow`
+∞	+∞	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

exp2

Computes the base 2 exponential of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event exp2(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event exp2(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`exp2` supports the following precisions.

T
`float`
`double`

Description

The exp2 function computes the base 2 exponential of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_2(\text{FLT\_MAX})$
double precision	$a_i < \log_2(\text{DBL\_MAX})$

Argument	Result	Status code
+0	+1	
-0	+1	
`a` > overflow	+∞	`oneapi::mkl::vm::status::overflow`
`a` < underflow	+0	`oneapi::mkl::vm::status::underflow`
+∞	+∞	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

expint1

Computes the exponential integral of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event expint1(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event expint1(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`expint1` supports the following precisions.

T
`float`
`double`

Description

The expint1(a) function computes the exponential integral of vector elements of the input vector `a` and writes them to the output vector `y`.

For positive real values `x`, this can be written as:

$$E_1(x) = \int_x^\infty \frac{e^{-t}}{t}\,\mathrm{d}t = \int_1^\infty \frac{e^{-xt}}{t}\,\mathrm{d}t$$

For negative real values `x`, the result is defined as NAN.

Argument	Result	Status code
x < +0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	+∞	`oneapi::mkl::vm::status::sing`
+∞	+0	
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

expm1

Computes an exponential of vector elements decreased by 1.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event expm1(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event expm1(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`expm1` supports the following precisions.

T
`float`
`double`

Description

The expm1(a) function computes an exponential of vector elements decreased by 1.

Argument	Result	Status code
+0	+1	
-0	+1	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
+∞	+∞	
-∞	-0	
QNAN	QNAN	
SNAN	QNAN	

Data Type	Threshold Limitations on Input Parameters
single precision	`a[i] < Log( FLT_MAX )`
double precision	`a[i] < Log( DBL_MAX )`

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

fdim

Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event fdim(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event fdim(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`fdim` supports the following precisions.

T
`float`
`double`

Description

The fdim(a, b) function returns a vector containing the differences of the corresponding elements of the first and second vector arguments if the first element is larger, and +0 otherwise.

Argument 1	Argument 2	Result	Status code
any	QNAN	QNAN	
any	SNAN	QNAN	
QNAN	any	QNAN	
SNAN	any	QNAN	

The fdim(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

floor

Computes an integer value rounded towards minus infinity for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event floor(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event floor(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`floor` supports the following precisions.

T
float
double

Description

The floor(a)function computes an integer value rounded towards minus infinity for each vector element.

$$y_i = \lfloor a_i \rfloor$$

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The floor function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

fmax

Returns the larger of each pair of elements of the two vector arguments.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event fmax(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event fmax(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`fmax` supports the following precisions.

T
float
double

Description

The fmax(a, b) function returns a vector with element values equal to the larger value from each pair of corresponding elements of the two vectors `a` and `b`: if `a` < `b`fmax(a, b) returns `b`, otherwise fmax(a, b) returns `a`.

Argument 1	Argument 2	Result	Status code
`a` not NAN	NAN	`a`	
NAN	`b` not NAN	`b`	
NAN	NAN	NAN	

The fmax(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

fmin

Returns the smaller of each pair of elements of the two vector arguments.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event fmin(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event fmin(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`fmin` supports the following precisions.

T
float
double

Description

The fmin(a, b) function returns a vector with element values equal to the smaller value from each pair of corresponding elements of the two vectors `a` and `b`: if `a` > `b`fmin(a, b) returns `b`, otherwise fmin(a, b) returns `a`.

Argument 1	Argument 2	Result	Status code
`a` not NAN	NAN	`a`	
NAN	`b` not NAN	`b`	
NAN	NAN	NAN	

The fmin(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

fmod

The fmod function performs element by element computation of the modulus function of vector `a` with respect to vector `b`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event fmod(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event fmod(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`fmod` supports the following precisions.

T
float
double

Description

The fmod (a, b) function computes the modulus function of each element of vector `a`, with respect to the corresponding elements of vector `b`:

```
ai - bi*trunc(ai/bi)
```

In general, the modulus function `fmod (ai, bi)` returns the value `ai - n*bi` for some integer `n` such that if b_i is nonzero, the result has the same sign as a_i and a magnitude less than the magnitude of b_i.

Argument 1	Argument 2	Result	Status code
`a` not NAN	± 0	NAN	`oneapi::mkl::vm::status::sing`
$\pm\infty$	`b` not NAN	NAN	`oneapi::mkl::vm::status::sing`
± 0	`b` $\neq 0$, not NAN	± 0	
`a` finite	$\pm\infty$	`a`	
NAN	`b`		
`a`	NAN	NAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

frac

Computes a signed fractional part for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event frac(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event frac(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`frac` supports the following precisions.

T
float
double

Description

The frac(a) function computes a signed fractional part for each vector element.

$$y_i = \begin{cases} a_i - \lfloor a_i \rfloor, a_i \geq 0 \\ a_i - \lceil a_i \rceil, a_i < 0 \end{cases}$$

Argument	Result	Status code
+0	+0	
-0	-0	
$+\infty$	+0	
$-\infty$	-0	
QNAN	QNAN	
SNAN	QNAN	

The frac function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

hypot

Computes a square root of sum of two squared elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event hypot(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event hypot(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`hypot` supports the following precisions.

T
float
double

Description

The function hypot(a, b) computes a square root of sum of two squared elements.

Argument 1	Argument 2	Result	Status code
+0	+0	+0	
-0	-0	+0	
+∞	any value	+∞	
any value	+∞	+∞	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	any value	QNAN	
any value	QNAN	QNAN	

Data Type	Threshold Limitations on Input Parameters
single precision	`abs(a[i]) < sqrt(FLT_MAX)abs(b[i]) < sqrt(FLT_MAX)`
double precision	`abs(a[i]) < sqrt(DBL_MAX)abs(b[i]) < sqrt(DBL_MAX)`

The hypot(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

inv

Performs element by element inversion of the vector.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event inv(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event inv(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`inv` supports the following precisions.

T
`float`
`double`

Description

The inv(a) function performs element by element inversion of the vector.

Argument	Result	VM status code
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
+∞	+0	
-∞	-0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

invcbrt

Computes an inverse cube root of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event invcbrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event invcbrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`invcbrt` supports the following precisions.

T
`float`
`double`

Description

The invcbrt(a)function computes an inverse cube root of vector elements.

Argument	Result	Status code
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
+∞	+0	
-∞	-0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

invsqrt

Computes an inverse square root of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event invsqrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event invsqrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`invsqrt` supports the following precisions.

T
`float`
`double`

Description

The invsqrt(a) function computes an inverse square root of vector elements.

Argument	Result	VM status code
a < +0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+0	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

lgamma

Computes the natural logarithm of the absolute value of gamma function for vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event lgamma(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event lgamma(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`lgamma` supports the following precisions.

T
`float`
`double`

Description

The lgamma(a) function computes the natural logarithm of the absolute value of gamma function for elements of the input vector `a` and writes them to the output vector `y`. Precision overflow thresholds for the lgamma function are beyond the scope of this document. If the result does not meet the target precision, the function sets the VM status code to `oneapi::mkl::vm::status::overflow`.

Argument	Result	VM status code
+1	+0	
+2	+0	
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	+∞	`oneapi::mkl::vm::status::sing`
negative integer	+∞	`oneapi::mkl::vm::status::sing`
-∞	+∞	
+∞	+∞	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

linearfrac

Performs linear fraction transformation of vectors `a` and `b` with scalar parameters.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event linearfrac(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        T scalea,
        T shifta,
        T scaleb,
        T shiftb,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event linearfrac(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T scalea,
        T shifta,
        T scaleb,
        T shiftb,
```

(continues on next page)

(continued from previous page)

```
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`linearfrac` supports the following precisions.

T
`float`
`double`

Description

The linearfrac(a, b, scalea, shifta, scaleb, shiftb) function performs a linear fraction transformation of vector `a` by vector `b` with scalar parameters: scaling multipliers scalea, scaleb and shifting addends shifta, shiftb:

`y[i]=(scalea·a[i]+shifta)/(scaleb·b[i]+shiftb), i=1,2 ... n`

The linearfrac function is implemented in the EP accuracy mode only, therefore no special values are defined for this function. If used in HA or LA mode, linearfrac sets the VM status code to `oneapi::mkl::vm::status::accuracy_warning`. Correctness is guaranteed within the threshold limitations defined for each input parameter (see the table below); otherwise, the behavior is unspecified.

Threshold Limitations on Input Parameters
2EMIN/2≤ |scalea| ≤ 2(EMAX−2)/2
2EMIN/2≤ |scaleb| ≤ 2(EMAX−2)/2
|shifta| ≤ 2EMAX−2
|shiftb| ≤ 2EMAX−2
2EMIN/2≤a[i] ≤ 2(EMAX−2)/2
2EMIN/2≤b[i] ≤ 2(EMAX−2)/2
a[i] ≠ − (shifta/scalea)∗(1−δ1), \|δ1| ≤ 21−(p−1)/2
b[i] ≠ − (shiftb/scaleb)∗(1−δ2), \|δ2| ≤ 21−(p−1)/2

`EMIN`and `EMAX` are the minimum and maximum exponents and `p` is the number of significant bits (precision) for the corresponding data type according to the ANSI/IEEE Standard 754-2008 ([*Bibliography*]):

- for single precision`EMIN` = -126, `EMAX` = 127, `p` = 24
- **for double precision`EMIN` = -1022, `EMAX` = 1023, `p` =** 53

The thresholds become less strict for common cases with scalea=0 and/or scaleb=0:

- **ifscalea=0, there are no limitations for the values of `a[i]`** and shifta.
- **ifscaleb=0, there are no limitations for the values of `b[i]`** and shiftb.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

scalea Constant value for scaling multipliers of vector `a`

shifta Constant value for shifting addend of vector `a`

scaleb Constant value for scaling multipliers of vector `b`

shiftb Constant value for shifting addend of vector `b`

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The pointer `a` to the 1st input vector of size `n`.

b The pointer `b` to the 2nd input vector of size `n`.

scalea Constant value for scaling multipliers of vector `a`

shifta Constant value for shifting addend of vector `a`

scaleb Constant value for scaling multipliers of vector `b`

shiftb Constant value for shifting addend of vector `b`

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

ln

Computes natural logarithm of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event ln(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event ln(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`ln` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The ln(a)function computes natural logarithm of vector elements.

Argument	Result	Status code
+1	+0	
a <+0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	-∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i \cdot \frac{3\pi}{4}$	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	+∞-i·π					+∞+i·0	QNAN+i·QNAN
+i·0	+∞-i·π		-∞+i·π	-∞-i·0		+∞+i·0	QNAN+i·QNAN
-i·0	+∞-i·π		-∞+i·π	-∞-i·0		+∞-i·0	QNAN+i·QNAN
-i·Y	+∞-i·π					+∞-i·0	QNAN+i·QNAN
-i·∞	$+\infty - i \cdot \frac{3\pi}{4}$	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	+∞+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

log10

Computes the base 10 logarithm of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event log10(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event log10(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`log10` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The log10(a) function computes the base 10 logarithm of vector elements.

Argument	Result	Status code
+1	+0	
a <+0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	-∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i\frac{3}{4}\frac{\pi}{\ln 10}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty + i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty + i\frac{\pi}{4}\frac{1}{\ln 10}$	+∞+i·QNAN
+i·Y	$+\infty + i\frac{\pi}{\ln 10}$					+∞+i·0	QNAN+i·QNAN
+i·0	$+\infty + i\frac{\pi}{\ln 10}$		$-\infty + i\frac{\pi}{\ln 10}$	-∞+i·0		+∞+i·0	QNAN+i·QNAN
-i·0	$+\infty - i\frac{\pi}{\ln 10}$		$-\infty - i\frac{\pi}{\ln 10}$	-∞-i·0		+∞-i·0	QNAN-i·QNAN
-i·Y	$+\infty - i\frac{\pi}{\ln 10}$					+∞-i·0	QNAN+i·QNAN
-i·∞	$+\infty + i\frac{3}{4}\frac{\pi}{\ln 10}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty - i\frac{\pi}{2}\frac{1}{\ln 10}$	$+\infty - i\frac{\pi}{4}\frac{1}{\ln 10}$	+∞+i·QNAN
+i·NAN	+∞+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

log1p

Computes a natural logarithm of vector elements that are increased by 1.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event log1p(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event log1p(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`log1p` supports the following precisions.

T
`float`
`double`

Description

The log1p(a) function computes a natural logarithm of vector elements that are increased by 1.

Argument	Result	VM status code
-1	-∞	`oneapi::mkl::vm::status::sing`
a <-1	QNAN	`oneapi::mkl::vm::status::errdom`
+0	+0	
-0	-0	
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

log2

Computes the base 2 logarithm of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event log2(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event log2(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`log2` supports the following precisions.

T
`float`
`double`

Description

The log2(a) function computes the base 2 logarithm of vector elements.

Argument	Result	Status code
+1	+0	
a < +0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	-∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

logb

Computes the exponents of the elements of input vector `a`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event logb(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event logb(
        sycl::queue& exec_queue,
        std::int64_t n,
```

(continues on next page)

(continued from previous page)

```
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`logb` supports the following precisions.

T
`float`
`double`

Description

The logb(a) function computes the exponents of the elements of the input vector `a`. For each element a_i of vector `a`, this is the integral part of $\log_2|a_i|$. The returned value is exact and is independent of the current rounding direction mode.

Argument	Result	VM status code
+0	+∞	`oneapi::mkl::vm::status::errdom`
-0	-∞	`oneapi::mkl::vm::status::errdom`
-∞	+∞	
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

maxmag

Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event maxmag(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event maxmag(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
```

(continues on next page)

(continued from previous page)

```
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`maxmag` supports the following precisions.

T
`float`
`double`

Description

The maxmag(a, b) function returns a vector with element values equal to the element with the larger magnitude from each pair of corresponding elements of the two vectors `a` and `b`:

- **If |a| > |b| maxmag(a, b) returns `a`,** otherwise maxmag(a, b) returns `b`.
- **If |b| > |a| maxmag(a, b) returns `b`,** otherwise maxmag(a, b) returns `a`.
- Otherwise maxmag(a, b) behaves like fmax.

Argument 1	Argument 2	Result	Status code
`a` not NAN	NAN	`a`	
NAN	`b` not NAN	`b`	
NAN	NAN	NAN	

The maxmag(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

minmag

Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event minmag(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event minmag(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`minmag` supports the following precisions.

T
`float`
`double`

Description

The minmag(a, b) function returns a vector with element values equal to the element with the smaller magnitude from each pair of corresponding elements of the two vectors `a` and `b`:

- **If |a| < |b| minmag(a, b) returns a,** otherwise minmag(a, b) returns `b`.
- **If |b| < |a| minmag(a, b) returns b,** otherwise minmag(a, b) returns `a`.
- Otherwise minmag behaves like fmin.

Argument 1	Argument 2	Result	Status code
`a` not NAN	NAN	`a`	
NAN	`b` not NAN	`b`	
NAN	NAN	NAN	

The minmag(a, b) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

modf

Computes a truncated integer value and the remaining fraction part for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event modf(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        sycl::buffer<T,1>& z,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event modf(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        T* z,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`modf` supports the following precisions.

T
`float`
`double`

Description

The modf(a) function computes a truncated integer value and the remaining fraction part for each vector element.

$$a_i \geq 0, \begin{cases} y_i = \lfloor a_i \rfloor \\ z_i = a_i - \lfloor a_i \rfloor \end{cases}$$

$$a_i < 0, \begin{cases} y_i = \lceil a_i \rceil \\ z_i = a_i - \lceil a_i \rceil \end{cases}$$

Argument	Result 1	Result 2	Status code
+0	+0	+0	
-0	-0	-0	
$+\infty$	$+\infty$	+0	
$-\infty$	$-\infty$	-0	
SNAN	QNAN	QNAN	
QNAN	QNAN	QNAN	

The modf function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n` for truncated integer values.

z The buffer `z` containing the output vector of size `n` for remaining fraction parts.

USM API:

y Pointer `y` to the output vector of size `n` for truncated integer values.

z Pointer `z` to the output vector of size `n` for remaining fraction parts.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

mul

Performs element by element multiplication of vector `a` and vector `b`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event mul(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event mul(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});
```

(continues on next page)

(continued from previous page)

```
} // namespace oneapi::mkl::vm
```

`mul` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The mul(a, b) function performs element by element multiplication of vector `a` and vector `b`.

Argument 1	Argument 2	Result	Status code
+0	+0	+0	
+0	-0	-0	
-0	+0	-0	
-0	-0	+0	
+0	+∞	QNAN	
+0	-∞	QNAN	
-0	+∞	QNAN	
-0	-∞	QNAN	
+∞	+0	QNAN	
+∞	-0	QNAN	
-∞	+0	QNAN	
-∞	-0	QNAN	
+∞	+∞	+∞	
+∞	-∞	-∞	
-∞	+∞	-∞	
-∞	-∞	+∞	
SNAN	any value	QNAN	
any value	SNAN	QNAN	
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

```
mul(x1+i*y1, x2+i*y2) = (x1*x2-y1*y2) + i*(x1*y2+y1*x2)
```

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow` (overriding any possible `oneapi::mkl::vm::status::accuracy_warning` status).

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

mulbyconj

Performs element by element multiplication of vector `a` element and conjugated vector `b` element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event mulbyconj(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event mulbyconj(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`mulbyconj` supports the following precisions.

T
std::complex<float>
std::complex<double>

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

nearbyint

Computes a rounded integer value in the current rounding mode for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event nearbyint(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
```

(continues on next page)

(continued from previous page)

```
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event nearbyint(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`nearbyint` supports the following precisions.

T
`float`
`double`

Description

The nearbyint(a) function computes a rounded integer value in a current rounding mode for each vector element.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The nearbyint function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

nextafter

Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event nextafter(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event nextafter(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`nextafter` supports the following precisions.

T
`float`
`double`

Description

The nextafter(a, b) function returns a vector containing the next representable floating-point values following the first vector argument elements in the direction of the second vector argument's corresponding elements.

Arguments/Results	Status code
Input vector argument element is finite and the corresponding result vector element value is infinite	`oneapi::mkl::vm::status::overflow`
Result vector element value is subnormal or zero, and different from the corresponding input vector argument element	`oneapi::mkl::vm::status::underflow`

Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

pow

Computes `a` to the power `b` for elements of two vectors.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event pow(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event pow(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`pow` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The pow(a, b) function computes `a` to the power `b` for elements of two vectors.

The real function pow has certain limitations on the input range of `a` and `b` parameters. Specifically, if `a`[i] is positive, then `b`[i] may be arbitrary. For negative `a`[i], the value of `b`[i] must be an integer (either positive or negative).

The complex function pow has no input range limitations.

Argument 1	Argument 2	Result	Status code
+0	neg. odd integer	+∞	`oneapi::mkl::vm::status::errdom`
-0	neg. odd integer	-∞	`oneapi::mkl::vm::status::errdom`
+0	neg. even integer	+∞	`oneapi::mkl::vm::status::errdom`
-0	neg. even integer	+∞	`oneapi::mkl::vm::status::errdom`
+0	neg. non-integer	+∞	`oneapi::mkl::vm::status::errdom`
-0	neg. non-integer	+∞	`oneapi::mkl::vm::status::errdom`
-0	pos. odd integer	+0	
-0	pos. odd integer	-0	
+0	pos. even integer	+0	
-0	pos. even integer	+0	
+0	pos. non-integer	+0	
-0	pos. non-integer	+0	
-1	+∞	+1	
-1	-∞	+1	
+1	any value	+1	
+1	+0	+1	
+1	-0	+1	
+1	+∞	+1	
+1	-∞	+1	
+1	QNAN	+1	

continues on next page

Table 17 – continued from previous page

Argument 1	Argument 2	Result	Status code
any value	+0	+1	
+0	+0	+1	
-0	+0	+1	
+∞	+0	+1	
-∞	+0	+1	
QNAN	+0	+1	
any value	-0	+1	
+0	-0	+1	
-0	-0	+1	
+∞	-0	+1	
-∞	-0	+1	
QNAN	-0	+1	
a < +0	non-integer	QNAN	`oneapi::mkl::vm::status::errdom`
|a| < 1	-∞	+∞	
+0	-∞	+∞	`oneapi::mkl::vm::status::errdom`
-0	-∞	+∞	`oneapi::mkl::vm::status::errdom`
|a| > 1	-∞	+0	
+∞	-∞	+0	
-∞	-∞	+0	
|a| < 1	+∞	+0	
+0	+∞	+0	
-0	+∞	+0	
|a| > 1	+∞	+∞	
+∞	+∞	+∞	
-∞	+∞	+∞	
-∞	neg. odd integer	-0	
-∞	neg. even integer	+0	
-∞	neg. non-integer	+0	
-∞	pos. odd integer	-∞	
-∞	pos. even integer	+∞	
-∞	pos. non-integer	+∞	
+∞	b < +0	+0	
+∞	b > +0	+∞	
Big finite value*	Big finite value*	+/-∞	`oneapi::mkl::vm::status::overflow`
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	
SNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

* Overflow in a real function is supported only in the HA/LA accuracy modes. The overflow occurs when x and y are finite numbers, but the result is too large to fit the target precision. In this case, the function:

1. Returns ∞ in the result.
2. Sets the VM status code to `oneapi::mkl::vm::status::overflow`.

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow` (overriding any possible `oneapi::mkl::vm::status::accuracy_warning` status).

The complex double precision versions of this function are implemented in the EP accuracy mode only. If used in HA

or LA mode, the functions set the VM status code to `oneapi::mkl::vm::status::accuracy_warning`.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

pow2o3

Computes the cube root of the square of each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event pow2o3(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event pow2o3(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`pow2o3` supports the following precisions.

T
float
double

Description

The pow2o3(a)function computes the cube root of the square of each vector element.

Argument	Result	Status code
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

pow3o2

Computes the square root of the cube of each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event pow3o2(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event pow3o2(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`pow3o2` supports the following precisions.

T
float
double

Description

The pow3o2(a)function computes the square root of the cube of each vector element.

Data Type	Threshold Limitations on Input Parameters
single precision	$\|a_i\| <$ `(FLT_MAX)2/3`
double precision	$\|a_i\| <$ `(FLT_MAX)2/3`

Argument	Result	VM status code
a < +0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	+0	
-0	-0	
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

powr

Computes `a` to the power `b` for elements of two vectors, where the elements of vector argument `a` are all non-negative.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event powr(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event powr(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`powr` supports the following precisions.

T
`float`
`double`

Description

The powr(a, b) function raises each element of vector `a` by the corresponding element of vector `b`. The elements of `a` are all nonnegative ($a_i \geq 0$).

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < (FLT\_MAX)^{1/`b`i`}$
double precision	$a_i < (DBL\_MAX)^{1/`b`i`}$

Special values and VM status code treatment for v?Powr function are the same as for pow, unless otherwise indicated in this table:

Argument 1	Argument 2	Result	Status code
`a` < 0	any value `b`	NAN	`oneapi::mkl::vm::status::errdom`
0 < `a` < ∞	±0	1	
±0	-∞ < `b` < 0	+∞	
±0	-∞	+∞	
±0	`b` > 0	+0	
1	-∞ < `b` < ∞	1	
±0	±0	NAN	
+∞	±0	NAN	
1	+∞	NAN	
`a` ≥ 0	NAN	NAN	
NAN	any value `b`	NAN	
0 < `a` <1	-∞	+∞	
`a` > 1	-∞	+0	
0 ≤ `a` < 1	+∞	+0	
`a` > 1	+∞	+∞	
+∞	`b` < +0	+0	
+∞	`b` > +0	+∞	
QNAN	QNAN	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	SNAN	QNAN	`oneapi::mkl::vm::status::errdom`
SNAN	QNAN	QNAN	`oneapi::mkl::vm::status::errdom`
SNAN	SNAN	QNAN	`oneapi::mkl::vm::status::errdom`

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

powx

Computes vector `a` to the scalar power `b`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event powx(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        T b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event powx(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`powx` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The powx function computes `a` to the power `b` for a vector `a` and a scalar `b`.

The real function powx has certain limitations on the input range of `a` and `b` parameters. Specifically, if `a`[i] is positive, then `b` may be arbitrary. For negative `a`[i], the value of `b` must be an integer (either positive or negative).

The complex function powx has no input range limitations.

Special values and VM status code treatment are the same as for the pow function.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b Fixed value of power `b`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Fixed value of power `b`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

remainder

Performs element by element computation of the remainder function on the elements of vector `a` and the corresponding elements of vector `b`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event remainder(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event remainder(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* b,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`remainder` supports the following precisions.

T
float
double

Description

The `remainder(a)` function computes the remainder of each element of vector `a`, with respect to the corresponding elements of vector `b`: compute the values of `n` such that

```
n = ai - n*bi
```

where `n` is the integer nearest to the exact value of a_i/b_i. If two integers are equally close to a_i/b_i, `n` is the even one. If `n` is zero, it has the same sign as a_i.

Argument 1	Argument 2	Result	VM status code
`a` not NAN	± 0	NAN	oneapi::mkl::vm::status::errdom
$\pm\infty$	`b` not NAN	NAN	
± 0	`b` $\neq 0$, not NAN	± 0	
`a` finite	$\pm\infty$	`a`	
NAN	`b`	NAN	
`a`	NAN	NAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

rint

Computes a rounded integer value in the current rounding mode.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event rint(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event rint(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`rint` supports the following precisions.

T
float
double

Description

The rint(a) function computes a rounded floating-point integer value using the current rounding mode for each vector element.

The rounding mode affects the results computed for inputs that fall between consecutive integers. For example:

- **`f(0.5) = 0`, for rounding modes set to round to nearest round** toward zero or to minus infinity.
- `f(0.5) = 1`, for rounding modes set to plus infinity.
- **`f(-1.5) = -2`, for rounding modes set to round to nearest or to** minus infinity.
- **`f(-1.5) = -1`, for rounding modes set to round toward zero or to** plus infinity.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The rint function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

round

Computes a value rounded to the nearest integer for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event round(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event round(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`round` supports the following precisions.

T
float
double

Description

The round(a) function computes a value rounded to the nearest integer for each vector element. Input elements that are halfway between two consecutive integers are always rounded away from zero regardless of the rounding mode.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The round(a) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sin

Computes sine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sin(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sin(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sin` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The sin(a)function computes sine of vector elements.

Note that arguments `abs(a[i]) ≤ 213` and `abs(a[i]) ≤ 216` for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result	VM status code
+0	+0	
-0	-0	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`Sin(z) = -i*Sinh(i*z).`

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sincos

Computes sine and cosine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sincos(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        sycl::buffer<T,1>& z,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sincos(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        T* z,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sincos` supports the following precisions.

T
`float`
`double`

Description

The sincos(a) function computes sine and cosine of vector elements.

Note that arguments `abs(a[i])` $\leq$ `213`and `abs(a[i])` $\leq$ `216`for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result 1	Result 2	Status code
+0	+0	+1	
-0	-0	+1	
+∞	QNAN	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	QNAN	
SNAN	QNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output sine vector of size `n`.

z The buffer `z` containing the output cosine vector of size `n`.

USM API:

y Pointer `y` to the output sine vector of size `n`.

z The buffer `z` containing the output cosine vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sind

Computes the sine of vector elements multiplied by $\pi/180$.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sind(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sind(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sind` supports the following precisions.

T
`float`
`double`

Description

The sind(a) function is a degree argument trigonometric function. It computes the sine of vector elements multiplied by $\pi/180$. For an argument `a`, the function computes sin(π*`a`/180).

Note that arguments abs(a_i) $\leq 2^{24}$ for single precision or abs(a_i) $\leq 2^{52}$ for double precision, they belong to the *fast computational path*: trigonometric function arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sinh

Computes hyperbolic sine of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sinh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sinh(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sinh` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The sinh(a) function computes hyperbolic sine of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	`-Log(FLT_MAX)-Log(2) <a[i] < Log(FLT_MAX)+Log(2)`
double precision	`-Log(DBL_MAX)-Log(2) <a[i] < Log(DBL_MAX)+Log(2)`

Argument	Result	Status code
+0	+0	
-0	-0	
a > overflow	+∞	`oneapi::mkl::vm::status::overflow`
a < -overflow	-∞	`oneapi::mkl::vm::status::overflow`
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	-∞+i·QNAN	QNAN+i·QNAN	0+i·QNAN	+0+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN
+i·Y	-∞·Cos(Y)+ i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
+i·0	-∞+i·0		-0+i·0	+0+i·0		+∞+i·0	QNAN+i·0
-i·0	-∞-i·0		-0-i·0	+0-i·0		+∞-i·0	QNAN-i·0
-i·Y	-∞·Cos(Y)+ i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
-i·∞	-∞+i·QNAN	QNAN+i·QNAN	0+i·QNAN	+0+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN
+i·NAN	-∞+i·QNAN	QNAN+i·QNAN	0+i·QNAN	+0+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- **The complex sinh(a) function sets the VM status code to** oneapi::mkl::vm::status::overflow in the case of overflow, that is, when RE(a), IM(a) are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- `sinh(CONJ(a))=CONJ(sinh(a))`
- `sinh(-a)=-sinh(a).`

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sinpi

Computes the sine of vector elements multiplied by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sinpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sinpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sinpi` supports the following precisions.

T
float
double

Description

The sinpi(a) function computes the sine of vector elements multiplied by π. For an argument `a`, the function computes sin(π*a).

Argument	Result	Status code
+0	+0	
-0	-0	
+n, positive integer	+0	
-n, negative integer	-0	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

If arguments abs(a_i) $\leq 2^{22}$ for single precision or abs(a_i) $\leq 2^{51}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sqr

Performs element by element squaring of the vector.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sqr(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sqr(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`sqr` supports the following precisions.

T
float
double

Description

The sqr() function performs element by element squaring of the vector.

Argument	Result	Status code
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

The `sqr` function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing the input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sqrt

Computes a square root of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sqrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event sqrt(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`sqrt` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The sqrt function computes a square root of vector elements.

Argument	Result	VM status code
a< +0	QNAN	`oneapi::mkl::vm::status::errdom`
+0	+0	
-0	-0	
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞	+∞+i·∞
+i·Y	+0+i·∞					+∞+i·0	
+i·0	+0+i·∞		+0+i·0	+0+i·0		+∞+i·0	
-i·0	+0-i·∞		+0-i·0	+0-i·0		+∞-i·0	
-i·Y	+0-i·∞					+∞-i·0	
-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞
+i·NAN							

Notes:

- `Sqrt(CONJ(z))=CONJ(Sqrt(z))`.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

sub

Performs element by element subtraction of vector `b` from vector `a`.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event sub(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& b,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
.. code-block:: cpp
```

namespace oneapi::mkl::vm {

sycl::event sub(sycl::queue& exec_queue, std::int64_t n, T* a, T* b, T* y, sycl::vector_class<sycl::event> const & depends = {}, oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined, oneapi::mkl::vm::error_handler<T> errhandler - {});

} // namespace oneapi::mkl::vm

`sub` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The sub(a, b) function performs element by element subtraction of vector `a` and vector `b`.

Argument 1	Argument 2	Result	Status code
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
+∞	+∞	QNAN	
+∞	-∞	+∞	
-∞	+∞	-∞	
-∞	-∞	QNAN	
SNAN	any value	QNAN	
any value	SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

```
sub(x1+i*y1, x2+i*y2) = (x1-x2) + i*(y1-y2)
```

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, and sets the VM status code to `oneapi::mkl::vm::status::overflow` (overriding any possible `oneapi::mkl::vm::status::accuracy_warning` status).

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing 1st input vector of size `n`.

b The buffer `b` containing 2nd input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the 1st input vector of size `n`.

b Pointer `b` to the 2nd input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tan

Computes tangent of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tan(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tan(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`tan` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

The tan(a) function computes tangent of vector elements.

Note that arguments `abs(a[i])` $\leq$ `213`and `abs(a[i])` $\leq$ `216`for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

`Tan(z) = -i*Tanh(i*z).`

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tand

Computes the tangent of vector elements multiplied by $\pi/180$.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tand(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tand(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`tand` supports the following precisions.

T
float
double

Description

The tand(a) function computes the tangent of vector elements multiplied by $\pi/180$. For an argument `x`, the function computes tan(π*x/180).

Note that arguments abs(a_i) $\leq 2^{38}$ for single precision or abs(a_i) $\leq 2^{67}$ for double precision, they belong to the *fast computational path*: trigonometric function arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Argument	Result	Status code
+0	+1	
-0	+1	
±∞	QNAN	`oneapi::mkl::vm::status::errdom`
±∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tanh

Computes hyperbolic tangent of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tanh(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tanh(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`tanh` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

Description

The tanh(a) function computes hyperbolic tangent of vector elements.

Argument	Result	Erro Code
+0	+0	
-0	-0	
+∞	+1	
-∞	-1	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	-1+i·0	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+1+i·0	QNAN+i·QNAN
+i·Y	-1+i·0·Tan(Y)					+1+i·0·Tan(Y)	QNAN+i·QNAN
+i·0	-1+i·0		-0+i·0	+0+i·0		+1+i·0	QNAN+i·0
-i·0	-1-i·0		-0-i·0	+0-i·0		+1-i·0	QNAN-i·0
-i·Y	-1+i·0·Tan(Y)					+1+i·0·Tan(Y)	QNAN+i·QNAN
-i·∞	-1-i·0	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+1-i·0	QNAN+i·QNAN
+i·NAN	-1+i·0	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+1+i·0	QNAN+i·QNAN

Notes:

- `tanh(CONJ(a))=CONJ(tanh(a))`
- `tanh(-a)=-tanh(a).`

The tanh(a) function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tanpi

Computes the tangent of vector elements multiplied by π.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tanpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tanpi(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`tanpi` supports the following precisions.

T
`float`
`double`

Description

The tanpi(a) function computes the tangent of vector elements multiplied by π. For an argument `a`, the function computes tan(π*`a`).

Argument	Result	Status code
+0	+0	
-0	+0	
`n`, even integer	*copysign(0.0, `n`)	
`n`, odd integer	*copysign(0.0, -`n`)	
`n` + 0.5, for `n` even integer and `n` + 0.5 representable	+∞	
`n` + 0.5, for `n` odd integer and `n` + 0.5 representable	-∞	
+∞	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
QNAN	QNAN	
SNAN	QNAN	

The copysign(`x`, `y`) function returns the first vector argument `x` with the sign changed to match that of the second argument `y`.

If arguments abs(a_i) $\leq 2^{13}$ for single precision or abs(a_i) $\leq 2^{67}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

tgamma

Computes the gamma function of vector elements.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event tgamma(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event tgamma(
        sycl::queue& exec_queue,
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined,
        oneapi::mkl::vm::error_handler<T> errhandler = {});

} // namespace oneapi::mkl::vm
```

`tgamma` supports the following precisions.

T
`float`
`double`

Description

The tgamma(a) function computes the gamma function for elements of the input vector `a` and writes them to the output vector `y`. Precision overflow thresholds for the tgamma function are beyond the scope of this document. If the result does not meet the target precision, the function raises sets the VM status code to oneapi::mkl::vm::status::sing.

Argument	Result	Status code
+0	+∞	`oneapi::mkl::vm::status::sing`
-0	-∞	`oneapi::mkl::vm::status::sing`
negative integer	QNAN	`oneapi::mkl::vm::status::errdom`
-∞	QNAN	`oneapi::mkl::vm::status::errdom`
+∞	+∞	
a > overflow	+∞	`oneapi::mkl::vm::status::sing`
QNAN	QNAN	
SNAN	QNAN	

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

errhandler Sets local error handling mode for this function call. See the *create_error_handler* function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

trunc

Computes an integer value rounded towards zero for each vector element.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

sycl::event trunc(
        sycl::queue& exec_queue,
        std::int64_t n,
        sycl::buffer<T,1>& a,
        sycl::buffer<T,1>& y,
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

sycl::event trunc(
        sycl::queue& exec_queue,
```

(continues on next page)

(continued from previous page)

```
        std::int64_t n,
        T* a,
        T* y,
        sycl::vector_class<sycl::event> const & depends = {},
        oneapi::mkl::vm::mode mode = oneapi::mkl::vm::mode::not_defined);

} // namespace oneapi::mkl::vm
```

`trunc` supports the following precisions.

T
`float`
`double`

Description

The trunc(a) function computes an integer value rounded towards zero for each vector element.

$$y_i = \begin{cases} \lfloor a_i \rfloor, a_i \geq 0 \\ \lceil a_i \rceil, a_i < 0 \end{cases}$$

Argument	Result	Status code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The trunc function does not generate any errors.

Input Parameters

Buffer API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a The buffer `a` containing input vector of size `n`.

mode Overrides the global VM mode setting for this function call. See *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

USM API:

exec_queue The queue where the routine should be executed.

n Specifies the number of elements to be calculated.

a Pointer `a` to the input vector of size `n`.

depends Vector of dependent events (to wait for input data to be ready).

mode Overrides the global VM mode setting for this function call. See the *set_mode* function for possible values and their description. This is an optional parameter. The default value is `oneapi::mkl::vm::mode::not_defined`.

Output Parameters

Buffer API:

y The buffer `y` containing the output vector of size `n`.

USM API:

y Pointer `y` to the output vector of size `n`.

return value (event) Event, signifying availability of computed output and status code(s).

Exceptions

For list of generated exceptions please refer to *Exceptions*

Parent topic: *VM Mathematical Functions*

VM Service Functions

The VM Service functions enable you to set/get the accuracy mode and error code. These functions are available both in the Fortran and C interfaces. The table below lists available VM Service functions and their short description.

Function Short Name	Description
set_mode	Sets the VM mode for given queue
get_mode	Gets the VM mode for given queue
set_status	Sets the VM status code for given queue
get_status	Gets the VM status code for given queue
clear_status	Clears the VM status code for given queue
create_error_handler	Creates the local VM error handler for a function

Parent topic: *Vector Math*

set_mode

Sets a new mode for VM functions according to the `mode` parameter and returns the previous VM mode.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::mode set_mode(
        sycl::queue& exec_queue,
        oneapi::mkl::vm::mode new_mode);

} // namespace oneapi::mkl::vm
```

Description

The set_mode function sets a new mode for VM functions according to the `new_mode` parameter and returns the previous VM mode. The mode change has a global effect on all the VM functions within a queue.

The `mode` parameter is designed to control accuracy for a given queue.

Value of mode	Description
Accuracy Control	
`oneapi::mkl::vm::mode::ha`	High accuracy versions of VM functions.
`oneapi::mkl::vm::mode::la`	Low accuracy versions of VM functions.
`oneapi::mkl::vm::mode::ep`	Enhanced performance accuracy versions of VM functions.
`oneapi::mkl::vm::mode::not_defined`	VM mode not defined. This has no effect.

The assumed value of the `mode` parameter for a new queue, if `set_mode` is not called is `oneapi::mkl::vm::mode::ha`.

Input Parameters

exec_queue The queue where the routine should be executed.

new_mode Specifies the VM mode to be set.

Output Parameters

return value (old_mode) Specifies the former VM mode.

Parent topic: *VM Service Functions*

get_mode

Gets the VM mode.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::mode get_mode(
        sycl::queue& exec_queue);

} // namespace oneapi::mkl::vm
```

Description

The function get_mode function returns the global VM `mode` parameter that controls accuracy for a given queue.

Value of mode	Description
Accuracy Control	
`oneapi::mkl::vm::mode::ha`	High accuracy versions of VM functions.
`oneapi::mkl::vm::mode::la`	Low accuracy versions of VM functions.
`oneapi::mkl::vm::mode::ep`	Enhanced performance accuracy versions of VM functions.
`oneapi::mkl::vm::mode::not_defined`	VM mode not defined. It means that no special provisions for accuracy have been made for this queue. See *set_mode* for details.

Input Parameters

exec_queue The queue where the routine should be executed.

Output Parameters

return value The current global VM mode for the queue `exec_queue`.

Parent topic: *VM Service Functions*

set_status

Sets the global VM status according to new value and returns the previous VM status.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::status set_status(
        sycl::queue& exec_queue,
        oneapi::mkl::vm::status new_status);

} // namespace oneapi::mkl::vm
```

Description

The set_status function sets the global VM status to new value and returns the previous VM status code for a given queue.

The global VM status is a single value and it registers the bitwise-OR of status codes that happened inside VM functions run on the specific queue. For performance reasons, it might be done in non-atomic manner. The possible status codes are listed in the table below.

Status	Description
Successful Execution	
`oneapi::mkl::vm::status::success`	VM function execution completed successfully
`oneapi::mkl::vm::status::not_defined`	VM status not defined
Warnings	
`oneapi::mkl::vm::status::accuracy_warning`	VM function execution completed successfully in a different accuracy mode
Computational status codes	
`oneapi::mkl::vm::status::errdom`	Values are out of a range of definition producing invalid (QNaN) result
`oneapi::mkl::vm::status::sing`	Values cause divide-by-zero (singularity) computational errors and produce and invalid (QNaN or Inf) result
`oneapi::mkl::vm::status::overflow`	An overflow happened during the calculation process
`oneapi::mkl::vm::status::underflow`	An underflow happened during the calculation process

Input Parameters

exec_queue The queue where the routine should be executed.

new_status Specifies the VM status to be set.

Output Parameters

return value (old_status) Specifies the former VM status.

Parent topic: *VM Service Functions*

get_status

Gets the VM status.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::status get_status(
        sycl::queue& exec_queue);

} // namespace oneapi::mkl::vm
```

Description

The get_status function gets the VM status for a given queue.

The global VM status is a single value and it registers the bitwise-OR of status codes that happened inside VM functions run on the specific queue. For performance reasons, it might be done in non-atomic manner. The possible status codes are listed in the table below.

Status	Description
Successful Execution	
`oneapi::mkl::vm::status::success`	VM function execution completed successfully
`oneapi::mkl::vm::status::not_defined`	VM status not defined
Warnings	
`oneapi::mkl::vm::status::accuracy_warning`	VM function execution completed successfully in a different accuracy mode
Computational status codes	
`oneapi::mkl::vm::status::errdom`	Values are out of a range of definition producing invalid (QNaN) result
`oneapi::mkl::vm::status::sing`	Values cause divide-by-zero (singularity) computational errors and produce and invalid (QNaN or Inf) result
`oneapi::mkl::vm::status::overflow`	An overflow happened during the calculation process
`oneapi::mkl::vm::status::underflow`	An underflow happened during the calculation process

Input Parameters

exec_queue The queue where the routine should be executed.

Output Parameters

return value (status) Specifies the VM status.

Parent topic: *VM Service Functions*

clear_status

Resets the global VM status to `oneapi::mkl::vm::status::success` and returns the previous VM status code.

Syntax

```
namespace oneapi::mkl::vm {

    oneapi::mkl::vm::status clear_status(
        sycl::queue& exec_queue);

} // namespace oneapi::mkl::vm
```

Description

The clear_status function sets the VM status code to `oneapi::mkl::vm::status::success` and returns the previous VM status code for a given queue.

The global VM status is a single value and it registers the bitwise-OR of status codes that happened inside VM functions run on the specific queue. For performance reasons, it might be done in non-atomic manner. The possible status codes are listed in the table below.

Status code	Description
Successful Execution	
`oneapi::mkl::vm::status::success`	VM function execution completed successfully
`oneapi::mkl::vm::status::not_defined`	VM status not defined
Warnings	
`oneapi::mkl::vm::status::accuracy_warning`	VM function execution completed successfully in a different accuracy mode
Computational status codes	
`oneapi::mkl::vm::status::errdom`	Values are out of a range of definition producing invalid (QNaN) result
`oneapi::mkl::vm::status::sing`	Values cause divide-by-zero (singularity) computational errors and produce and invalid (QNaN or Inf) result
`oneapi::mkl::vm::status::overflow`	An overflow happened during the calculation process
`oneapi::mkl::vm::status::underflow`	An underflow happened during the calculation process

Input Parameters

exec_queue The queue where the routine should be executed.

Output Parameters

return value (old_status) Specifies the VM status code before the call.

Parent topic: *VM Service Functions*

create_error_handler

Creates an error handler for VM functions that support computational error handling.

Syntax

Buffer API:

```
namespace oneapi::mkl::vm {

oneapi::mkl::vm::error_handler<T> create_error_handler(
    sycl::buffer<oneapi::mkl::vm::status, 1> & status_array,
    int64_t length = 1,
    oneapi::mkl::vm::status status = oneapi::mkl::vm::status::not_defined,
    T fixup = {},
    bool copysign = false);

} // namespace oneapi::mkl::vm
```

USM API:

```
namespace oneapi::mkl::vm {

oneapi::mkl::vm::error_handler<T> create_error_handler(
    oneapi::mkl::vm::status* status_array,
    int64_t length = 1,
    oneapi::mkl::vm::status status = oneapi::mkl::vm::status::not_defined,
```

(continues on next page)

(continued from previous page)

```
    T fixup = {},
    bool copysign = false);

} // namespace oneapi::mkl::vm
```

`create_error_handler` supports the following precisions.

T
`float`
`double`
`std::complex<float>`
`std::complex<double>`

Description

create_error_handler creates an computational error handler to be passed to VM functions that support computational error handling.

A VM computational error handler supports three modes:

- **Single status mode**: all computational errors that happened during the execution of a function are being written into one single status variable.

  After the execution, the single value is either un-changed if no errors happened or contains bitwise-OR of initial value and non-success status codes occurred during computation.

  To enable this mode, `status_array` must point to any `status`-type array or buffer of 1 or more elements and `length` must be 1.

- **Multiple status mode**: each non-successful status code is saved in `status_array` at the same index as the argument causing the non-success status code.

  Success status codes are not written to `status_array`. This means the array needs to be allocated and initialized before function execution.

  To enable this mode, `status_array` must have at least the same length as the argument and result vectors, and `length` must be set to this length.

- **Fixup mode**: for all arguments that caused a specific error status, results are overwritten by a user-defined value.

  To enable this mode, the target `status` and `fixup` values must be set. The fixup value is written to results for each argument for which calculation resulted in the `status` status code.

  To fix multiple error status codes, `status` can be provided with bitwise-OR of status codes.

  If `copysign` is set to true then the sign of `fixup` is set to the same sign as the argument that caused the status code – a suitable option for symmetric math functions.

The following table lists the possible computational status code values.

Status code	Description
Successful Execution	
`oneapi::mkl::vm::status::success`	VM function execution completed successfully
`oneapi::mkl::vm::status::not_defined`	VM status not defined
Warnings	
`oneapi::mkl::vm::status::accuracy_warning`	VM function execution completed successfully in a different accuracy mode
Computational Errors	
`oneapi::mkl::vm::status::errdom`	Values are out of a range of definition producing invalid (QNaN) result
`oneapi::mkl::vm::status::sing`	Values cause divide-by-zero (singularity) computational errors and produce and invalid (QNaN or Inf) result
`oneapi::mkl::vm::status::overflow`	An overflow happened during the calculation process
`oneapi::mkl::vm::status::underflow`	An underflow happened during the calculation process

Notes:

- `status_array` must be allocated and initialized before calling VM functions in multiple status error handling mode.

  The array should be large enough to contain `n` status codes, where `n` is the same as the input/output vector size for the VM function.

- If no arguments are passed to `create_error_handler`, then an empty object is created with all three error handling modes disabled.

  In this case, the VM math functions set the global status code only.

Input Parameters

status_array Array to store status codes (should be a buffer for buffer API).

length Length of the errarray. This is an optional argument, default value is `1`.

status_code Status code to match and fix the results. This is an optional argument, default value is `oneapi::mkl::vm::status::not_defined`.

fixup Fixup value for results. This is an optional argument, default value is `0.0`.

copysign Flag for setting the fixup value's sign the same as the argument's. This is an optional argument, default value `false`.

Output Parameters

return value Specifies the error handler object to be created.

Parent topic: *VM Service Functions*

Exceptions

All VM mathematical functions throw exceptions in exceptional cases. The following table summarizes the conditions.

exception	when thrown

oneapi::mkl::invalid_argument	**buffer API:** n < 0;
	y.get_count() < n;
	z.get_count() < n; // for sincos

	USM API: n < 0; any pointer argument is nullptr
oneapi::mkl::host_bad_alloc	**USM API:** when internal copying to and from host memory is used and corresponding allocation fails
oneapi::mkl::device_bad_alloc	**USM API:** when internal copying to and from device memory is used and corresponding allocation fails

Bibliography

For more information about the VM functionality, refer to the following publications:

- **VM**

  [IEEE754] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-2008.

12.3 oneMKL Appendix

12.3.1 Future considerations

The following items are being considered for future versions of this specification:

- Encapsulation of matrix and vector information in classes. Matrix storage information could also be encapsulated.
- More human-readable names for linear algebra functionality, aligned with the P1673 C++ proposal.
- Broader support for row major layout.
- Alternative handling of computational failures.

12.3.2 Acknowledgment

The oneMKL Technical Advisory Board members provided valuable feedback to the specification and are thanked for their contributions.

CHAPTER

THIRTEEN

ADVANCED RENDERING

13.1 Overview

oneAPI Advanced Ray Tracing defines a set of Ray Tracing and high-fidelity Rendering and computation routines for use in a wide variety of 3D graphics uses including, film and television photorealistic visual effects and animation rendering, scientific visualization, high-performance computing computations, gaming, and more. Advanced rendering is designed to allow cooperative execution on a wide variety of computational devices: CPUs, GPUs, FPGAs, and other accelerators, termed "XPU" computation. The functionality is subdivided into several domains: geometric ray tracing computations, volumetric computation and rendering, path guided ray tracing, image denoising, and an integrated rendering infrastructure and API utilizing all the individual kernel capabilities integrated into a highly capable, easy to use rendering engine.

The individual components and their APIs are described. Other design considerations and related components that are not necessarily part of the advanced rendering specification but that are worth mentioning will be discussed in the appendix.

13.1.1 Component Libraries

There are 4 domains.

Embree

Embree is a collection of high-performance ray tracing kernels. The Embree target users are graphics application engineers who want to improve the performance of their photo-realistic rendering application by leveraging Embree's performance-optimized ray tracing kernels. Embree supports runtime code selection to choose the traversal and build algorithms that best matches the instruction set of your CPU.

Embree supports applications written with the Intel® SPMD Program Compiler (ISPC, https://ispc.github.io/) by also providing an ISPC interface to the core ray tracing algorithms. This makes it possible to write a renderer in ISPC that automatically vectorizes and leverages SSE, AVX, AVX2, and AVX-512 instructions. ISPC also supports runtime code selection, thus ISPC will select the best code path for your application.

Embree contains algorithms optimized for incoherent workloads (e.g. Monte Carlo ray tracing algorithms) and coherent workloads (e.g. primary visibility and hard shadow rays).

The single-ray traversal kernels of Embree provide high performance for incoherent workloads and are very easy to integrate into existing rendering applications. Using the stream kernels, even higher performance for incoherent rays is possible, but integration might require significant code changes to the application to use the stream paradigm. In general for coherent workloads, the stream mode with coherent flag set gives the best performance.

Embree also supports dynamic scenes by implementing high-performance two-level spatial index structure construction algorithms.

Introduction

The Embree API is a low-level C99 ray tracing API which can be used to construct 3D scenes and perform ray queries of different types inside these scenes. All API calls carry the prefix `rtc` (or `RTC` for types) which stands for **r**ay **t**racing **c**ore.

The API also exists in an ISPC version, which is almost identical but contains additional functions that operate on ray packets with a size of the native SIMD width used by ISPC. For simplicity this document refers to the C99 version of the API functions. For changes when upgrading from the Embree 2 to the current Embree 3 API see Section [Upgrading from Embree 2 to Embree 3].

The API supports scenes consisting of different geometry types such as triangle meshes, quad meshes (triangle pairs), grid meshes, flat curves, round curves, oriented curves, subdivision meshes, instances, and user-defined geometries. See Section *Scene Object* for more information.

Finding the closest hit of a ray segment with the scene (`rtcIntersect`-type functions), and determining whether any hit between a ray segment and the scene exists (`rtcOccluded`-type functions) are both supported. The API supports queries for single rays, ray packets, and ray streams. See Section *Ray Queries* for more information.

The API is designed in an object-oriented manner, e.g. it contains device objects (`RTCDevice` type), scene objects (`RTCScene` type), geometry objects (`RTCGeometry` type), buffer objects (`RTCBuffer` type), and BVH objects (`RTCBVH` type). All objects are reference counted, and handles can be released by calling the appropriate release function (e.g. `rtcReleaseDevice`) or retained by incrementing the reference count (e.g. `rtcRetainDevice`). In general, API calls that access the same object are not thread-safe, unless specified differently. However, attaching geometries to the same scene and performing ray queries in a scene is thread-safe.

Device Object

Embree supports a device concept, which allows different components of the application to use the Embree API without interfering with each other. An application typically first creates a device using the *rtcNewDevice* function. This device can then be used to construct further objects, such as scenes and geometries. Before the application exits, it should release all devices by invoking *rtcReleaseDevice*. An application typically creates only a single device. If required differently, it should only use a small number of devices at any given time.

Each user thread has its own error flag per device. If an error occurs when invoking an API function, this flag is set to an error code (if it isn't already set by a previous error). See Section *rtcGetDeviceError* for information on how to read the error code and Section *rtcSetDeviceErrorFunction* on how to register a callback that is invoked for each error encountered. It is recommended to always set a error callback function, to detect all errors.

Scene Object

A scene is a container for a set of geometries, and contains a spatial acceleration structure which can be used to perform different types of ray queries.

A scene is created using the `rtcNewScene` function call, and released using the `rtcReleaseScene` function call. To populate a scene with geometries use the `rtcAttachGeometry` call, and to detach them use the `rtcDetachGeometry` call. Once all scene geometries are attached, an `rtcCommitScene` call (or `rtcJoinCommitScene` call) will finish the scene description and trigger building of internal data structures. After the scene got committed, it is safe to perform ray queries (see Section *Ray Queries*) or to query the scene bounding box (see *rtcGetSceneBounds* and *rtcGetSceneLinearBounds*).

If scene geometries get modified or attached or detached, the `rtcCommitScene` call must be invoked before performing any further ray queries for the scene; otherwise the effect of the ray query is undefined. The modification of a geometry, committing the scene, and tracing of rays must always happen sequentially, and never at the same

time. Any API call that sets a property of the scene or geometries contained in the scene count as scene modification, e.g. including setting of intersection filter functions.

Scene flags can be used to configure a scene to use less memory (`RTC_SCENE_FLAG_COMPACT`), use more robust traversal algorithms (`RTC_SCENE_FLAG_ROBUST`), and to optimize for dynamic content. See Section *rtcSetSceneFlags* for more details.

A build quality can be specified for a scene to balance between acceleration structure build performance and ray query performance. See Section *rtcSetSceneBuildQuality* for more details on build quality.

Geometry Object

A new geometry is created using the `rtcNewGeometry` function. Depending on the geometry type, different buffers must be bound (e.g. using `rtcSetSharedGeometryBuffer`) to set up the geometry data. In most cases, binding of a vertex and index buffer is required. The number of primitives and vertices of that geometry is typically inferred from the size of these bound buffers.

Changes to the geometry always must be committed using the `rtcCommitGeometry` call before using the geometry. After committing, a geometry is not included in any scene. A geometry can be added to a scene by using the `rtcAttachGeometry` function (to automatically assign a geometry ID) or using the `rtcAttachGeometryById` function (to specify the geometry ID manually). A geometry can get attached to multiple scenes.

All geometry types support multi-segment motion blur with an arbitrary number of equidistant time steps (in the range of 2 to 129) inside a user specified time range. Each geometry can have a different number of time steps and a different time range. The motion blur geometry is defined by linearly interpolating the geometries of neighboring time steps. To construct a motion blur geometry, first the number of time steps of the geometry must be specified using the `rtcSetGeometryTimeStepCount` function, and then a vertex buffer for each time step must be bound, e.g. using the `rtcSetSharedGeometryBuffer` function. Optionally, a time range defining the start (and end time) of the first (and last) time step can be set using the `rtcSetGeometryTimeRange` function. This feature will also allow geometries to appear and disappear during the camera shutter time if the time range is a sub range of [0,1].

The API supports per-geometry filter callback functions (see `rtcSetGeometryIntersectFilterFunction` and `rtcSetGeometryOccludedFilterFunction`) that are invoked for each intersection found during the `rtcIntersect`-type or `rtcOccluded`-type calls. The former ones are called geometry intersection filter functions, the latter ones geometry occlusion filter functions. These filter functions are designed to be used to ignore intersections outside of a user-defined silhouette of a primitive, e.g. to model tree leaves using transparency textures.

Ray Queries

The API supports finding the closest hit of a ray segment with the scene (`rtcIntersect`-type functions), and determining whether any hit between a ray segment and the scene exists (`rtcOccluded`-type functions).

Supported are single ray queries (`rtcIntersect1` and `rtcOccluded1`) as well as ray packet queries for ray packets of size 4 (`rtcIntersect4` and `rtcOccluded4`), ray packets of size 8 (`rtcIntersect8` and `rtcOccluded8`), and ray packets of size 16 (`rtcIntersect16` and `rtcOccluded16`).

Ray streams in a variety of layouts are supported as well, such as streams of single rays (`rtcIntersect1M` and `rtcOccluded1M`), streams of pointers to single rays (`rtcIntersect1p` and `rtcOccluded1p`), streams of ray packets (`rtcIntersectNM` and `rtcOccludedNM`), and large packet-like streams in structure of pointer layout (`rtcIntersectNp` and `rtcOccludedNp`).

See Sections *rtcIntersect1* and *rtcOccluded1* for a detailed description of how to set up and trace a ray.

See tutorial Triangle Geometry for a complete example of how to trace single rays and ray packets. Also have a look at the tutorial Stream Viewer for an example of how to trace ray streams.

Point Queries

The API supports traversal of the BVH using a point query object that specifies a location and a query radius. For all primitives intersecting the according domain, a user defined callback function is called which allows queries such as finding the closest point on the surface geometries of the scene (see Tutorial Closest Point) or nearest neighbour queries (see Tutorial Voronoi).

See Section *rtcPointQuery* for a detailed description of how to set up point queries.

Collision Detection

The Embree API also supports collision detection queries between two scenes consisting only of user geometries. Embree only performs broadphase collision detection, the narrow phase detection can be performed through a callback function.

See Section *rtcCollide* for a detailed description of how to set up collision detection.

Seen tutorial *Collision Detection* for a complete example of collsion detection being used on a simple cloth solver.

Miscellaneous

A context filter function, which can be set per ray query is supported (see `rtcInitIntersectContext`). This filter function is designed to change the semantics of the ray query, e.g. to accumulate opacity for transparent shadows, count the number of surfaces along a ray, collect all hits along a ray, etc.

The internal algorithms to build a BVH are exposed through the `RTCBVH` object and `rtcBuildBVH` call. This call makes it possible to build a BVH in a user-specified format over user-specified primitives. See the documentation of the `rtcBuildBVH` call for more details.

For getting the most performance out of Embree, see the Section [Performance Recommendations].

Embree API

rtcNewDevice

NAME

```
rtcNewDevice - creates a new device
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCDevice rtcNewDevice(const char* config);
```

DESCRIPTION

This function creates a new device and returns a handle to this device. The device object is reference counted with an initial reference count of 1. The handle can be released using the `rtcReleaseDevice` API call.

The device object acts as a class factory for all other object types. All objects created from the device (like scenes, geometries, etc.) hold a reference to the device, thus the device will not be destroyed unless these objects are destroyed first.

Objects are only compatible if they belong to the same device, e.g it is not allowed to create a geometry in one device and attach it to a scene created with a different device.

A configuration string (`config` argument) can be passed to the device construction. This configuration string can be `NULL` to use the default configuration.

When creating the device, Embree reads configurations for the device from the following locations in order:

1) `config` string passed to the `rtcNewDevice` function
2) `.embree3` file in the application folder
3) `.embree3` file in the home folder

Settings performed later overwrite previous settings. This way the configuration for the application can be changed globally (either through the `rtcNewDevice` call or through the `.embree3` file in the application folder), and each user has the option to modify the configuration to fit their needs.

The following configuration is supported:

- `threads=[int]`: Specifies a number of build threads to use. A value of 0 enables all detected hardware threads. By default all hardware threads are used.
- `user_threads=[int]`: Sets the number of user threads that can be used to join and participate in a scene commit using `rtcJoinCommitScene`. The tasking system will only use threads-user_threads many worker threads, thus if the app wants to solely use its threads to commit scenes, just set threads equal to user_threads. This option only has effect with the Intel(R) Threading Building Blocks (TBB) tasking system.
- `set_affinity=[0/1]`: When enabled, build threads are affinitized to hardware threads. This option is disabled by default on standard CPUs, and enabled by default on Xeon Phi Processors.
- `start_threads=[0/1]`: When enabled, the build threads are started upfront. This can be useful for benchmarking to exclude thread creation time. This option is disabled by default.

- `isa=[sse2,sse4.2,avx,avx2,avx512]`: Use specified ISA. By default the ISA is selected automatically.
- `max_isa=[sse2,sse4.2,avx,avx2,avx512]`: Configures the automated ISA selection to use maximally the specified ISA.
- `hugepages=[0/1]`: Enables or disables usage of huge pages. Under Linux huge pages are used by default but under Windows and macOS they are disabled by default.
- `enable_selockmemoryprivilege=[0/1]`: When set to 1, this enables the `SeLockMemoryPrivilege` privilege with is required to use huge pages on Windows. This option has an effect only under Windows and is ignored on other platforms. See Section [Huge Page Support] for more details.
- `ignore_config_files=[0/1]`: When set to 1, configuration files are ignored. Default is 0.
- `verbose=[0,1,2,3]`: Sets the verbosity of the output. When set to 0, no output is printed by Embree, when set to a higher level more output is printed. By default Embree does not print anything on the console.
- `frequency_level=[simd128,simd256,simd512]`: Specifies the frequency level the application want to run on, which can be either:
  a) simd128 to run at highest frequency
  b) simd256 to run at AVX2-heavy frequency level
  c) simd512 to run at heavy AVX512 frequency level. When some frequency level is specified, Embree will avoid doing optimizations that may reduce the frequency level below the level specified. E.g. if your app does not use AVX instructions setting "frequency_level=simd128" will cause some CPUs to run at highest frequency, which may result in higher application performance if you do much shading. If you application heavily uses AVX code, you should best set the frequency level to simd256. Per default Embree tries to avoid reducing the frequency of the CPU by setting the simd256 level only when the CPU has no significant down clocking.

Different configuration options should be separated by commas, e.g.:

```
rtcNewDevice("threads=1,isa=avx");
```

EXIT STATUS

On success returns a handle of the created device. On failure returns `NULL` as device and sets a per-thread error code that can be queried using `rtcGetDeviceError(NULL)`.

SEE ALSO

rtcRetainDevice, *rtcReleaseDevice*

rtcRetainDevice

NAME

```
rtcRetainDevice - increments the device reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcRetainDevice(RTCDevice device);
```

DESCRIPTION

Device objects are reference counted. The `rtcRetainDevice` function increments the reference count of the passed device object (`device` argument). This function together with `rtcReleaseDevice` allows to use the internal reference counting in a C++ wrapper class to manage the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewDevice, *rtcReleaseDevice*

rtcReleaseDevice

NAME

```
rtcReleaseDevice - decrements the device reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcReleaseDevice(RTCDevice device);
```

DESCRIPTION

Device objects are reference counted. The `rtcReleaseDevice` function decrements the reference count of the passed device object (`device` argument). When the reference count falls to 0, the device gets destroyed.

All objects created from the device (like scenes, geometries, etc.) hold a reference to the device, thus the device will not get destroyed unless these objects are destroyed first.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewDevice, *rtcRetainDevice*

rtcGetDeviceProperty

NAME

```
rtcGetDeviceProperty - queries properties of the device
```

SYNOPSIS

```
#include <embree3/rtcore.h>

ssize_t rtcGetDeviceProperty(
  RTCDevice device,
  enum RTCDeviceProperty prop
);
```

DESCRIPTION

The `rtcGetDeviceProperty` function can be used to query properties (`prop` argument) of a device object (`device` argument). The returned property is an integer of type `ssize_t`.

Possible properties to query are:

- `RTC_DEVICE_PROPERTY_VERSION`: Queries the combined version number (MAJOR.MINOR.PATCH) with two decimal digits per component. E.g. for Embree 2.8.3 the integer 208003 is returned.
- `RTC_DEVICE_PROPERTY_VERSION_MAJOR`: Queries the major version number of Embree.
- `RTC_DEVICE_PROPERTY_VERSION_MINOR`: Queries the minor version number of Embree.
- `RTC_DEVICE_PROPERTY_VERSION_PATCH`: Queries the patch version number of Embree.
- `RTC_DEVICE_PROPERTY_NATIVE_RAY4_SUPPORTED`: Queries whether the `rtcIntersect4` and `rtcOccluded4` functions preserve packet size and ray order when invoking callback functions. This is only the case if Embree is compiled with `EMBREE_RAY_PACKETS` and `SSE2` (or `SSE4.2`) enabled, and if the machine it is running on supports `SSE2` (or `SSE4.2`).
- `RTC_DEVICE_PROPERTY_NATIVE_RAY8_SUPPORTED`: Queries whether the `rtcIntersect8` and `rtcOccluded8` functions preserve packet size and ray order when invoking callback functions. This is only the case if Embree is compiled with `EMBREE_RAY_PACKETS` and `AVX` (or `AVX2`) enabled, and if the machine it is running on supports `AVX` (or `AVX2`).
- `RTC_DEVICE_PROPERTY_NATIVE_RAY16_SUPPORTED`: Queries whether the `rtcIntersect16` and `rtcOccluded16` functions preserve packet size and ray order when invoking callback functions. This is only the case if Embree is compiled with `EMBREE_RAY_PACKETS` and `AVX512` enabled, and if the machine it is running on supports `AVX512`.
- `RTC_DEVICE_PROPERTY_RAY_STREAM_SUPPORTED`: Queries whether `rtcIntersect1M`, `rtcIntersect1Mp`, `rtcIntersectNM`, `rtcIntersectNp`, `rtcOccluded1M`, `rtcOccluded1Mp`, `rtcOccludedNM`, and `rtcOccludedNp` are supported. This is only the case if Embree is compiled with `EMBREE_RAY_PACKETS` enabled.
- `RTC_DEVICE_PROPERTY_RAY_MASK_SUPPORTED`: Queries whether ray masks are supported. This is only the case if Embree is compiled with `EMBREE_RAY_MASK` enabled.
- `RTC_DEVICE_PROPERTY_BACKFACE_CULLING_ENABLED`: Queries whether back face culling is enabled. This is only the case if Embree is compiled with `EMBREE_BACKFACE_CULLING` enabled.

- RTC_DEVICE_PROPERTY_COMPACT_POLYS_ENABLED: Queries whether compact polys is enabled. This is only the case if Embree is compiled with EMBREE_COMPACT_POLYS enabled.
- RTC_DEVICE_PROPERTY_FILTER_FUNCTION_SUPPORTED: Queries whether filter functions are supported, which is the case if Embree is compiled with EMBREE_FILTER_FUNCTION enabled.
- RTC_DEVICE_PROPERTY_IGNORE_INVALID_RAYS_ENABLED: Queries whether invalid rays are ignored, which is the case if Embree is compiled with EMBREE_IGNORE_INVALID_RAYS enabled.
- RTC_DEVICE_PROPERTY_TRIANGLE_GEOMETRY_SUPPORTED: Queries whether triangles are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_TRIANGLE enabled.
- RTC_DEVICE_PROPERTY_QUAD_GEOMETRY_SUPPORTED: Queries whether quads are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_QUAD enabled.
- RTC_DEVICE_PROPERTY_SUBDIVISION_GEOMETRY_SUPPORTED: Queries whether subdivision meshes are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_SUBDIVISION enabled.
- RTC_DEVICE_PROPERTY_CURVE_GEOMETRY_SUPPORTED: Queries whether curves are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_CURVE enabled.
- RTC_DEVICE_PROPERTY_POINT_GEOMETRY_SUPPORTED: Queries whether points are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_POINT enabled.
- RTC_DEVICE_PROPERTY_USER_GEOMETRY_SUPPORTED: Queries whether user geometries are supported, which is the case if Embree is compiled with EMBREE_GEOMETRY_USER enabled.
- RTC_DEVICE_PROPERTY_TASKING_SYSTEM: Queries the tasking system Embree is compiled with. Possible return values are:

  0. internal tasking system
  1. Intel Threading Building Blocks (TBB)
  2. Parallel Patterns Library (PPL)

- RTC_DEVICE_PROPERTY_JOIN_COMMIT_SUPPORTED: Queries whether rtcJoinCommitScene is supported. This is not the case when Embree is compiled with PPL or older versions of TBB.
- RTC_DEVICE_PROPERTY_PARALLEL_COMMIT_SUPPORTED: Queries whether rtcCommitScene can get invoked from multiple TBB worker threads concurrently. This feature is only supported starting with TBB 2019 Update 9.

EXIT STATUS

On success returns the value of the queried property. For properties returning a boolean value, the return value 0 denotes false and 1 denotes true.

On failure zero is returned and an error code is set that can be queried using rtcGetDeviceError.

rtcGetDeviceError

NAME

```
rtcGetDeviceError - returns the error code of the device
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCError rtcGetDeviceError(RTCDevice device);
```

DESCRIPTION

Each thread has its own error code per device. If an error occurs when calling an API function, this error code is set to the occurred error if it stores no previous error. The `rtcGetDeviceError` function reads and returns the currently stored error and clears the error code. This assures that the returned error code is always the first error occurred since the last invocation of `rtcGetDeviceError`.

Possible error codes returned by `rtcGetDeviceError` are:

- `RTC_ERROR_NONE`: No error occurred.
- `RTC_ERROR_UNKNOWN`: An unknown error has occurred.
- `RTC_ERROR_INVALID_ARGUMENT`: An invalid argument was specified.
- `RTC_ERROR_INVALID_OPERATION`: The operation is not allowed for the specified object.
- `RTC_ERROR_OUT_OF_MEMORY`: There is not enough memory left to complete the operation.
- `RTC_ERROR_UNSUPPORTED_CPU`: The CPU is not supported as it does not support the lowest ISA Embree is compiled for.
- `RTC_ERROR_CANCELLED`: The operation got canceled by a memory monitor callback or progress monitor callback function.

When the device construction fails, `rtcNewDevice` returns `NULL` as device. To detect the error code of a such a failed device construction, pass `NULL` as device to the `rtcGetDeviceError` function. For all other invocations of `rtcGetDeviceError`, a proper device pointer must be specified.

EXIT STATUS

Returns the error code for the device.

SEE ALSO

rtcSetDeviceErrorFunction

rtcSetDeviceErrorFunction

NAME

```
rtcSetDeviceErrorFunction - sets an error callback function for the device
```

SYNOPSIS

```
#include <embree3/rtcore.h>

typedef void (*RTCErrorFunction)(
  void* userPtr,
  RTCError code,
  const char* str
);

void rtcSetDeviceErrorFunction(
  RTCDevice device,
  RTCErrorFunction error,
  void* userPtr
);
```

DESCRIPTION

Using the `rtcSetDeviceErrorFunction` call, it is possible to set a callback function (`error` argument) with payload (`userPtr` argument), which is called whenever an error occurs for the specified device (`device` argument).

Only a single callback function can be registered per device, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

When the registered callback function is invoked, it gets passed the user-defined payload (`userPtr` argument as specified at registration time), the error code (`code` argument) of the occurred error, as well as a string (`str` argument) that further describes the error.

The error code is also set if an error callback function is registered.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetDeviceError

rtcSetDeviceMemoryMonitorFunction

NAME

```
rtcSetDeviceMemoryMonitorFunction - registers a callback function
  to track memory consumption
```

SYNOPSIS

```
#include <embree3/rtcore.h>

typedef bool (*RTCMemoryMonitorFunction)(
  void* userPtr,
  ssize_t bytes,
  bool post
);

void rtcSetDeviceMemoryMonitorFunction(
  RTCDevice device,
  RTCMemoryMonitorFunction memoryMonitor,
  void* userPtr
);
```

DESCRIPTION

Using the `rtcSetDeviceMemoryMonitorFunction` call, it is possible to register a callback function (`memoryMonitor` argument) with payload (`userPtr` argument) for a device (`device` argument), which is called whenever internal memory is allocated or deallocated by objects of that device. Using this memory monitor callback mechanism, the application can track the memory consumption of an Embree device, and optionally terminate API calls that consume too much memory.

Only a single callback function can be registered per device, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

Once registered, the Embree device will invoke the memory monitor callback function before or after it allocates or frees important memory blocks. The callback function gets passed the payload as specified at registration time (`userPtr` argument), the number of bytes allocated or deallocated (`bytes` argument), and whether the callback is invoked after the allocation or deallocation took place (`post` argument). The callback function might get called from multiple threads concurrently.

The application can track the current memory usage of the Embree device by atomically accumulating the `bytes` input parameter provided to the callback function. This parameter will be >0 for allocations and <0 for deallocations.

Embree will continue its operation normally when returning `true` from the callback function. If `false` is returned, Embree will cancel the current operation with the `RTC_ERROR_OUT_OF_MEMORY` error code. Issuing multiple cancel requests from different threads is allowed. Canceling will only happen when the callback was called for allocations (bytes > 0), otherwise the cancel request will be ignored.

If a callback to cancel was invoked before the allocation happens (`post == false`), then the `bytes` parameter should not be accumulated, as the allocation will never happen. If the callback to cancel was invoked after the allocation happened (`post == true`), then the `bytes` parameter should be accumulated, as the allocation properly happened and a deallocation will later free that data block.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewDevice

rtcNewScene

NAME

```
rtcNewScene - creates a new scene
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCScene rtcNewScene(RTCDevice device);
```

DESCRIPTION

This function creates a new scene bound to the specified device (`device` argument), and returns a handle to this scene. The scene object is reference counted with an initial reference count of 1. The scene handle can be released using the `rtcReleaseScene` API call.

EXIT STATUS

On success a scene handle is returned. On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcRetainScene, *rtcReleaseScene*

rtcGetSceneDevice

NAME

```
rtcGetSceneDevice - returns the device the scene got created in
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCDevice rtcGetSceneDevice(RTCScene scene);
```

DESCRIPTION

This function returns the device object the scene got created in. The returned handle own one additional reference to the device object, thus you should need to call `rtcReleaseDevice` when the returned handle is no longer required.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcReleaseDevice

rtcRetainScene

NAME

```
rtcRetainScene - increments the scene reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcRetainScene(RTCScene scene);
```

DESCRIPTION

Scene objects are reference counted. The `rtcRetainScene` function increments the reference count of the passed scene object (`scene` argument). This function together with `rtcReleaseScene` allows to use the internal reference counting in a C++ wrapper class to handle the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewScene, *rtcReleaseScene*

rtcReleaseScene

NAME

```
rtcReleaseScene - decrements the scene reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcReleaseScene(RTCScene scene);
```

DESCRIPTION

Scene objects are reference counted. The `rtcReleaseScene` function decrements the reference count of the passed scene object (`scene` argument). When the reference count falls to 0, the scene gets destroyed.

The scene holds a reference to all attached geometries, thus if the scene gets destroyed, all geometries get detached and their reference count decremented.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewScene, *rtcRetainScene*

rtcAttachGeometry

NAME

```
rtcAttachGeometry - attaches a geometry to the scene
```

SYNOPSIS

```
#include <embree3/rtcore.h>

unsigned int rtcAttachGeometry(
  RTCScene scene,
  RTCGeometry geometry
);
```

DESCRIPTION

The `rtcAttachGeometry` function attaches a geometry (`geometry` argument) to a scene (`scene` argument) and assigns a geometry ID to that geometry. All geometries attached to a scene are defined to be included inside the scene. A geometry can get attached to multiplee scene. The geometry ID is unique for the scene, and is used to identify the geometry when hit by a ray during ray queries.

This function is thread-safe, thus multiple threads can attach geometries to a scene in parallel.

The geometry IDs are assigned sequentially, starting from 0, as long as no geometry got detached. If geometries got detached, the implementation will reuse IDs in an implementation dependent way. Consequently sequential assignment is no longer guaranteed, but a compact range of IDs.

These rules allow the application to manage a dynamic array to efficiently map from geometry IDs to its own geometry representation. Alternatively, the application can also use per-geometry user data to map to its geometry representation. See `rtcSetGeometryUserData` and `rtcGetGeometryUserData` for more information.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryUserData, *rtcGetGeometryUserData*

rtcAttachGeometryByID

NAME

```
rtcAttachGeometryByID - attaches a geometry to the scene
  using a specified geometry ID
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcAttachGeometryByID(
  RTCScene scene,
  RTCGeometry geometry,
  unsigned int geomID
);
```

DESCRIPTION

The `rtcAttachGeometryByID` function attaches a geometry (`geometry` argument) to a scene (`scene` argument) and assigns a user provided geometry ID (`geomID` argument) to that geometry. All geometries attached to a scene are defined to be included inside the scene. A geometry can get attached to multiple scenes. The passed user-defined geometry ID is used to identify the geometry when hit by a ray during ray queries. Using this function, it is possible to share the same IDs to refer to geometries inside the application and Embree.

This function is thread-safe, thus multiple threads can attach geometries to a scene in parallel.

The user-provided geometry ID must be unused in the scene, otherwise the creation of the geometry will fail. Further, the user-provided geometry IDs should be compact, as Embree internally creates a vector which size is equal to the largest geometry ID used. Creating very large geometry IDs for small scenes would thus cause a memory consumption and performance overhead.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcAttachGeometry

rtcDetachGeometry

NAME

```
rtcDetachGeometry - detaches a geometry from the scene
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcDetachGeometry(RTCScene scene, unsigned int geomID);
```

DESCRIPTION

This function detaches a geometry identified by its geometry ID (`geomID` argument) from a scene (`scene` argument). When detached, the geometry is no longer contained in the scene.

This function is thread-safe, thus multiple threads can detach geometries from a scene at the same time.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcAttachGeometry, *rtcAttachGeometryByID*

rtcGetGeometry

NAME

```
rtcGetGeometry - returns the geometry bound to
  the specified geometry ID
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCGeometry rtcGetGeometry(RTCScene scene, unsigned int geomID);
```

DESCRIPTION

The `rtcGetGeometry` function returns the geometry that is bound to the specified geometry ID (`geomID` argument) for the specified scene (`scene` argument). This function just looks up the handle and does *not* increment the reference count. If you want to get ownership of the handle, you need to additionally call `rtcRetainGeometry`. For this reason, this function is fast and can be used during rendering. However, it is generally recommended to store the geometry handle inside the application's geometry representation and look up the geometry handle from that representation directly.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcAttachGeometry, *rtcAttachGeometryByID*

rtcCommitScene

NAME

```
rtcCommitScene - commits scene changes
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcCommitScene(RTCScene scene);
```

DESCRIPTION

The `rtcCommitScene` function commits all changes for the specified scene (`scene` argument). This internally triggers building of a spatial acceleration structure for the scene using all available worker threads. Ray queries can be performed only after committing all scene changes.

If the application uses TBB 2019 Update 9 or later for parallelization of rendering, lazy scene construction during rendering is supported by `rtcCommitScene`. Therefore `rtcCommitScene` can get called from multiple TBB worker threads concurrently for the same scene. The `rtcCommitScene` function will then internally isolate the scene construction using a tbb::isolated_task_group. The alternative approach of using `rtcJoinCommitScene` which uses an tbb:task_arena internally, is not recommended due to it's high runtime overhead.

If scene geometries get modified or attached or detached, the `rtcCommitScene` call must be invoked before performing any further ray queries for the scene; otherwise the effect of the ray query is undefined. The modification of a geometry, committing the scene, and tracing of rays must always happen sequentially, and never at the same time. Any API call that sets a property of the scene or geometries contained in the scene count as scene modification, e.g. including setting of intersection filter functions.

The kind of acceleration structure built can be influenced using scene flags (see `rtcSetSceneFlags`), and the quality can be specified using the `rtcSetSceneBuildQuality` function.

Embree silently ignores primitives during spatial acceleration structure construction that would cause numerical issues, e.g. primitives containing NaNs, INFs, or values greater than 1.844E18f (as no reasonable calculations can be performed with such values without causing overflows).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcJoinCommitScene

rtcJoinCommitScene

NAME

```
rtcJoinCommitScene - commits the scene from multiple threads
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcJoinCommitScene(RTCScene scene);
```

DESCRIPTION

The `rtcJoinCommitScene` function commits all changes for the specified scene (`scene` argument). The scene commit internally triggers building of a spatial acceleration structure for the scene. Ray queries can be performed after scene changes got properly committed.

The `rtcJoinCommitScene` function can get called from multiple user threads which will all cooperate in the build operation. All threads calling into this function will return from `rtcJoinCommitScene` after the scene commit is finished. All threads must consistently call `rtcJoinCommitScene` and not `rtcCommitScene`.

In contrast to the `rtcCommitScene` function, the `rtcJoinCommitScene` function can be called from multiple user threads, while the `rtcCommitScene` can only get called from multiple TBB worker threads when used concurrently. For optimal performance we strongly recommend using TBB inside the application together with the `rtcCommitScene` function and to avoid using the `rtcJoinCommitScene` function.

The `rtcJoinCommitScene` feature allows a flexible way to lazily create hierarchies during rendering. A thread reaching a not-yet-constructed sub-scene of a two-level scene can generate the sub-scene geometry and call `rtcJoinCommitScene` on that just generated scene. During construction, further threads reaching the not-yet-built scene can join the build operation by also invoking `rtcJoinCommitScene`. A thread that calls `rtcJoinCommitScene` after the build finishes will directly return from the `rtcJoinCommitScene` call.

Multiple scene commit operations on different scenes can be running at the same time, hence it is possible to commit many small scenes in parallel, distributing the commits to many threads.

When using Embree with the Intel® Threading Building Blocks (which is the default), threads that call `rtcJoinCommitScene` will join the build operation, but other TBB worker threads might also participate in the build. To avoid thread oversubscription, we recommend using TBB also inside the application. Further, the join mode only works properly starting with TBB v4.4 Update 1. For earlier TBB versions, threads that call `rtcJoinCommitScene` to join a running build will just trigger the build and wait for the build to finish. Further, old TBB versions with `TBB_INTERFACE_VERSION_MAJOR < 8` do not support `rtcJoinCommitScene`, and invoking this function will result in an error.

When using Embree with the internal tasking system, only threads that call `rtcJoinCommitScene` will perform the build operation, and no additional worker threads will be scheduled.

When using Embree with the Parallel Patterns Library (PPL), `rtcJoinCommitScene` is not supported and calling that function will result in an error.

To detect whether `rtcJoinCommitScene` is supported, use the `rtcGetDeviceProperty` function.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcCommitScene, *rtcGetDeviceProperty*

rtcSetSceneProgressMonitorFunction

NAME

```
rtcSetSceneProgressMonitorFunction - registers a callback
  to track build progress
```

SYNOPSIS

```
#include <embree3/rtcore.h>

typedef bool (*RTCProgressMonitorFunction)(
  void* ptr,
  double n
);

void rtcSetSceneProgressMonitorFunction(
  RTCScene scene,
  RTCProgressMonitorFunction progress,
  void* userPtr
);
```

DESCRIPTION

Embree supports a progress monitor callback mechanism that can be used to report progress of hierarchy build operations and to cancel build operations.

The `rtcSetSceneProgressMonitorFunction` registers a progress monitor callback function (`progress` argument) with payload (`userPtr` argument) for the specified scene (`scene` argument).

Only a single callback function can be registered per scene, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

Once registered, Embree will invoke the callback function multiple times during hierarchy build operations of the scene, by passing the payload as set at registration time (`userPtr` argument), and a double in the range $[0, 1]$ which estimates the progress of the operation (`n` argument). The callback function might be called from multiple threads concurrently.

When returning `true` from the callback function, Embree will continue the build operation normally. When returning `false`, Embree will cancel the build operation with the `RTC_ERROR_CANCELLED` error code. Issuing multiple cancel requests for the same build operation is allowed.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewScene

rtcSetSceneBuildQuality

NAME

```
rtcSetSceneBuildQuality - sets the build quality for
  the scene
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetSceneBuildQuality(
  RTCScene scene,
  enum RTCBuildQuality quality
);
```

DESCRIPTION

The `rtcSetSceneBuildQuality` function sets the build quality (`quality` argument) for the specified scene (`scene` argument). Possible values for the build quality are:

- `RTC_BUILD_QUALITY_LOW`: Create lower quality data structures, e.g. for dynamic scenes. A two-level spatial index structure is built when enabling this mode, which supports fast partial scene updates, and allows for setting a per-geometry build quality through the `rtcSetGeometryBuildQuality` function.
- `RTC_BUILD_QUALITY_MEDIUM`: Default build quality for most usages. Gives a good compromise between build and render performance.
- `RTC_BUILD_QUALITY_HIGH`: Create higher quality data structures for final-frame rendering. For certain geometry types this enables a spatial split BVH.

Selecting a higher build quality results in better rendering performance but slower scene commit times. The default build quality for a scene is `RTC_BUILD_QUALITY_MEDIUM`.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryBuildQuality

rtcSetSceneFlags

NAME

```
rtcSetSceneFlags - sets the flags for the scene
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetSceneFlags(RTCScene scene, enum RTCSceneFlags flags);
```

DESCRIPTION

The `rtcSetSceneFlags` function sets the scene flags (`flags` argument) for the specified scene (`scene` argument). Possible scene flags are:

- `RTC_SCENE_FLAG_NONE`: No flags set.
- `RTC_SCENE_FLAG_DYNAMIC`: Provides better build performance for dynamic scenes (but also higher memory consumption).
- `RTC_SCENE_FLAG_COMPACT`: Uses compact acceleration structures and avoids algorithms that consume much memory.
- `RTC_SCENE_FLAG_ROBUST`: Uses acceleration structures that allow for robust traversal, and avoids optimizations that reduce arithmetic accuracy. This mode is typically used for avoiding artifacts caused by rays shooting through edges of neighboring primitives.
- `RTC_SCENE_FLAG_CONTEXT_FILTER_FUNCTION`: Enables support for a filter function inside the intersection context for this scene. See Section *rtcInitIntersectContext* for more details.

Multiple flags can be enabled using an `or` operation, e.g. `RTC_SCENE_FLAG_COMPACT | RTC_SCENE_FLAG_ROBUST`.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetSceneFlags

rtcGetSceneFlags

NAME

```
rtcGetSceneFlags - returns the flags of the scene
```

SYNOPSIS

```
#include <embree3/rtcore.h>

enum RTCSceneFlags rtcGetSceneFlags(RTCScene scene);
```

DESCRIPTION

Queries the flags of a scene. This function can be useful when setting individual flags, e.g. to just set the robust mode without changing other flags the following way:

```
RTCSceneFlags flags = rtcGetSceneFlags(scene);
rtcSetSceneFlags(scene, RTC_SCENE_FLAG_ROBUST | flags);
```

EXIT STATUS

On failure `RTC_SCENE_FLAG_NONE` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetSceneFlags

rtcGetSceneBounds

NAME

```
rtcGetSceneBounds - returns the axis-aligned bounding box of the scene
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCORE_ALIGN(16) RTCBounds
{
  float lower_x, lower_y, lower_z, align0;
  float upper_x, upper_y, upper_z, align1;
};

void rtcGetSceneBounds(
  RTCScene scene,
  struct RTCBounds* bounds_o
);
```

DESCRIPTION

The `rtcGetSceneBounds` function queries the axis-aligned bounding box of the specified scene (`scene` argument) and stores that bounding box to the provided destination pointer (`bounds_o` argument). The stored bounding box consists of lower and upper bounds for the x, y, and z dimensions as specified by the `RTCBounds` structure.

The provided destination pointer must be aligned to 16 bytes. The function may be invoked only after committing the scene; otherwise the result is undefined.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetSceneLinearBounds, *rtcCommitScene*, *rtcJoinCommitScene*

rtcGetSceneLinearBounds

NAME

```
rtcGetSceneLinearBounds - returns the linear bounds of the scene
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCORE_ALIGN(16) RTCLinearBounds
{
  RTCBounds bounds0;
  RTCBounds bounds1;
};

void rtcGetSceneLinearBounds(
  RTCScene scene,
  struct RTCLinearBounds* bounds_o
);
```

DESCRIPTION

The `rtcGetSceneLinearBounds` function queries the linear bounds of the specified scene (`scene` argument) and stores them to the provided destination pointer (`bounds_o` argument). The stored linear bounds consist of bounding boxes for time 0 (`bounds0` member) and time 1 (`bounds1` member) as specified by the `RTCLinearBounds` structure. Linearly interpolating these bounds to a specific time `t` yields bounds for the geometry at that time.

The provided destination pointer must be aligned to 16 bytes. The function may be called only after committing the scene, otherwise the result is undefined.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetSceneBounds, *rtcCommitScene*, *rtcJoinCommitScene*

rtcNewGeometry

NAME

```
rtcNewGeometry - creates a new geometry object
```

SYNOPSIS

```
#include <embree3/rtcore.h>

enum RTCGeometryType
{
 RTC_GEOMETRY_TYPE_TRIANGLE,
 RTC_GEOMETRY_TYPE_QUAD,
 RTC_GEOMETRY_TYPE_SUBDIVISION,
 RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE,
 RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE,
 RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE,
 RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE,
 RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE,
 RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BEZIER_CURVE,
 RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BSPLINE_CURVE,
 RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_HERMITE_CURVE,
 RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_CATMULL_ROM_CURVE,
 RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE,
 RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE,
 RTC_GEOMETRY_TYPE_GRID,
 RTC_GEOMETRY_TYPE_SPHERE_POINT,
 RTC_GEOMETRY_TYPE_DISC_POINT,
 RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT,
 RTC_GEOMETRY_TYPE_USER,
 RTC_GEOMETRY_TYPE_INSTANCE
};

RTCGeometry rtcNewGeometry(
  RTCDevice device,
  enum RTCGeometryType type
);
```

DESCRIPTION

Geometries are objects that represent an array of primitives of the same type. The `rtcNewGeometry` function creates a new geometry of specified type (`type` argument) bound to the specified device (`device` argument) and returns a handle to this geometry. The geometry object is reference counted with an initial reference count of 1. The geometry handle can be released using the `rtcReleaseGeometry` API call.

Supported geometry types are triangle meshes (`RTC_GEOMETRY_TYPE_TRIANGLE` type), quad meshes (triangle pairs) (`RTC_GEOMETRY_TYPE_QUAD` type), Catmull-Clark subdivision surfaces

(RTC_GEOMETRY_TYPE_SUBDIVISION type), curve geometries with different bases (RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE, RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE, RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE, RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE, RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BEZIER_CURVE, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BSPLINE_CURVE, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_HERMITE_CURVE, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_CATMULL_ROM_CURVE, RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE, RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE, RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE, RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE, RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE, RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE types) grid meshes (RTC_GEOMETRY_TYPE_GRID), point geometries (RTC_GEOMETRY_TYPE_SPHERE_POINT, RTC_GEOMETRY_TYPE_DISC_POINT, RTC_TYPE_ORIENTED_DISC_POINT), user-defined geometries (RTC_GEOMETRY_TYPE_USER), and instances (RTC_GEOMETRY_TYPE_INSTANCE).

The types RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE, RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE, and RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE will treat the curve as a sweep surface of a varying-radius circle swept tangentially along the curve. The types RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE, RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE, and RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE use ray-facing ribbons as a faster-to-intersect approximation.

After construction, geometries are enabled by default and not attached to any scene. Geometries can be disabled (rtcDisableGeometry call), and enabled again (rtcEnableGeometry call). A geometry can be attached to multiple scenes using the rtcAttachGeometry call (or rtcAttachGeometryByID call), and detached using the rtcDetachGeometry call. During attachment, a geometry ID is assigned to the geometry (or assigned by the user when using the rtcAttachGeometryByID call), which uniquely identifies the geometry inside that scene. This identifier is returned when primitives of the geometry are hit in later ray queries for the scene.

Geometries can also be modified, including their vertex and index buffers. After modifying a buffer, rtcUpdateGeometryBuffer must be called to notify that the buffer got modified.

The application can use the rtcSetGeometryUserData function to set a user data pointer to its own geometry representation, and later read out this pointer using the rtcGetGeometryUserData function.

After setting up the geometry or modifying it, rtcCommitGeometry must be called to finish the geometry setup. After committing the geometry, vertex data interpolation can be performed using the rtcInterpolate and rtcInterpolateN functions.

A build quality can be specified for a geometry using the rtcSetGeometryBuildQuality function, to balance between acceleration structure build performance and ray query performance. The build quality per geometry will be used if a two-level acceleration structure is built internally, which is the case if the RTC_BUILD_QUALITY_LOW is set as the scene build quality. See Section *rtcSetSceneBuildQuality* for more details.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcEnableGeometry, *rtcDisableGeometry*, *rtcAttachGeometry*, *rtcAttachGeometryByID*, *rtcUpdateGeometryBuffer*, *rtcSetGeometryUserData*, *rtcGetGeometryUserData*, *rtcCommitGeometry*, *rtcInterpolate*, *rtcInterpolateN*, *rtcSetGeometryBuildQuality*, *rtcSetSceneBuildQuality*, *RTC_GEOMETRY_TYPE_TRIANGLE*, *RTC_GEOMETRY_TYPE_QUAD*, *RTC_GEOMETRY_TYPE_SUBDIVISION*, *RTC_GEOMETRY_TYPE_CURVE*, *RTC_GEOMETRY_TYPE_GRID*, *RTC_GEOMETRY_TYPE_POINT*, *RTC_GEOMETRY_TYPE_USER*, *RTC_GEOMETRY_TYPE_INSTANCE*

RTC_GEOMETRY_TYPE_TRIANGLE

NAME

```
RTC_GEOMETRY_TYPE_TRIANGLE - triangle geometry type
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_TRIANGLE);
```

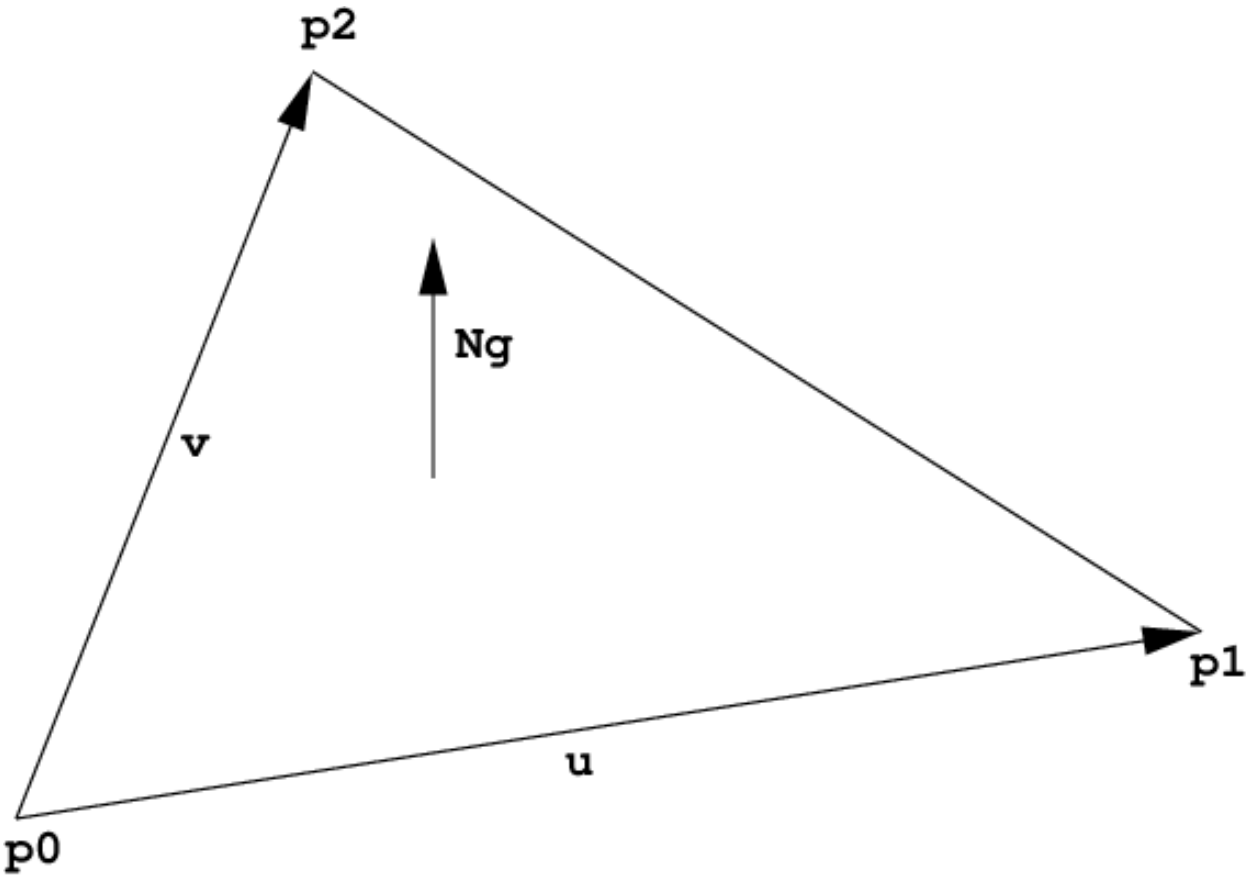
DESCRIPTION

Triangle meshes are created by passing `RTC_GEOMETRY_TYPE_TRIANGLE` to the `rtcNewGeometry` function call. The triangle indices can be specified by setting an index buffer (`RTC_BUFFER_TYPE_INDEX` type) and the triangle vertices by setting a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` type). See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers. The index buffer must contain an array of three 32-bit indices per triangle (`RTC_FORMAT_UINT3` format) and the number of primitives is inferred from the size of that buffer. The vertex buffer must contain an array of single precision `x`, `y`, `z` floating point coordinates (`RTC_FORMAT_FLOAT3` format), and the number of vertices are inferred from the size of that buffer. The vertex buffer can be at most 16 GB large.

The parametrization of a triangle uses the first vertex `p0` as base point, the vector `p1 - p0` as u-direction and the vector `p2 - p0` as v-direction. Thus vertex attributes `t0,t1,t2` can be linearly interpolated over the triangle the following way:

```
t_uv = (1-u-v)*t0 + u*t1 + v*t2
     = t0 + u*(t1-t0) + v*(t2-t0)
```

A triangle whose vertices are laid out counter-clockwise has its geometry normal pointing upwards outside the front face, like illustrated in the following picture:



For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers have to have the same stride and size.

Also see tutorial Triangle Geometry for an example of how to create triangle meshes.

EXIT STATUS

On failure `NULL` is returned and an error code is set that be get queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry

RTC_GEOMETRY_TYPE_QUAD

NAME

```
RTC_GEOMETRY_TYPE_QUAD - quad geometry type
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_QUAD);
```

DESCRIPTION

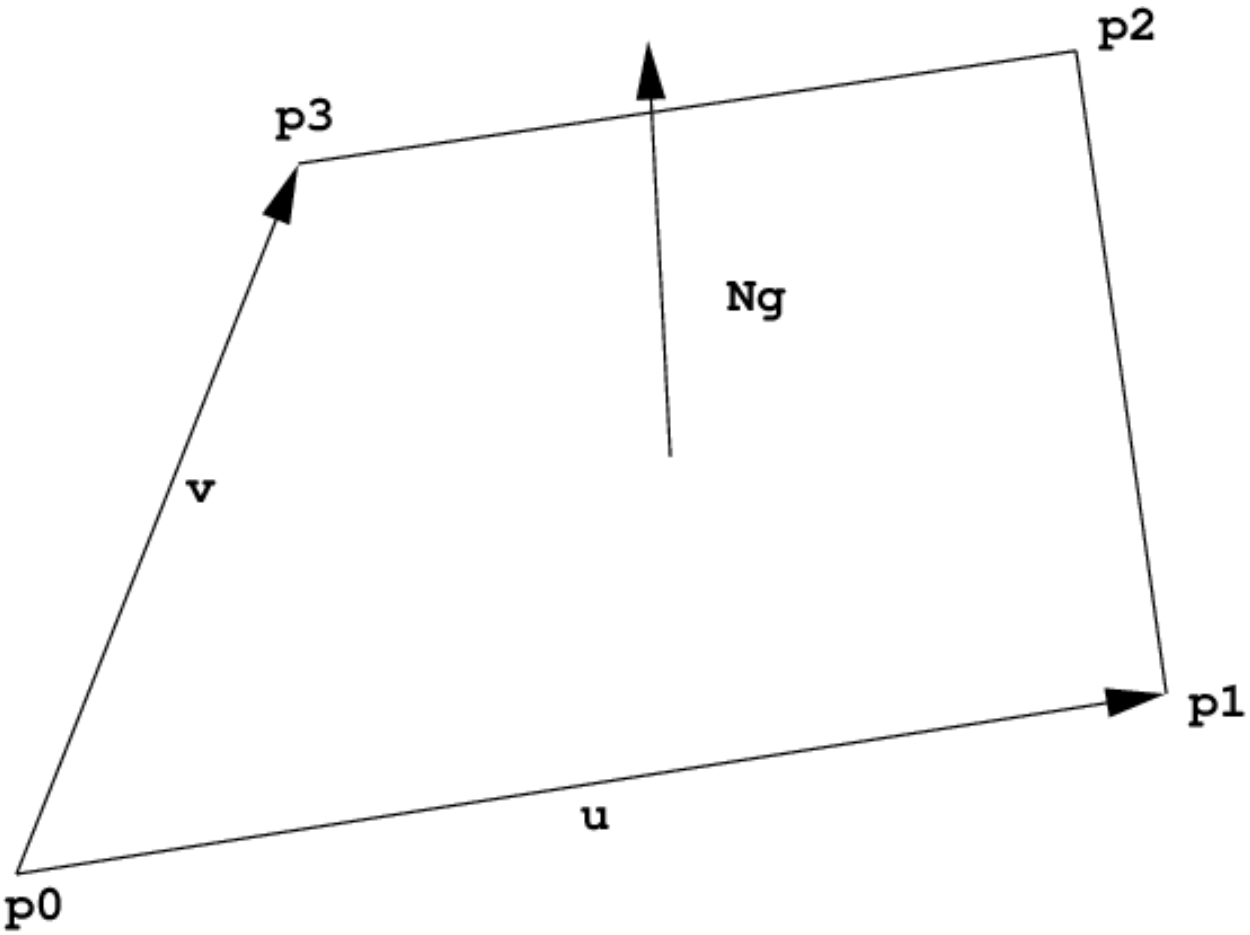
Quad meshes are created by passing `RTC_GEOMETRY_TYPE_QUAD` to the `rtcNewGeometry` function call. The quad indices can be specified by setting an index buffer (`RTC_BUFFER_TYPE_INDEX` type) and the quad vertices by setting a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` type). See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers. The index buffer contains an array of four 32-bit indices per quad (`RTC_FORMAT_UINT4` format), and the number of primitives is inferred from the size of that buffer. The vertex buffer contains an array of single precision `x`, `y`, `z` floating point coordinates (`RTC_FORMAT_FLOAT3` format), and the number of vertices is inferred from the size of that buffer. The vertex buffer can be at most 16 GB large.

A quad is internally handled as a pair of two triangles `v0,v1,v3` and `v2,v3,v1`, with the `u'/v'` coordinates of the second triangle corrected by `u = 1-u'` and `v = 1-v'` to produce a quad parametrization where `u` and `v` are in the range 0 to 1. Thus the parametrization of a quad uses the first vertex `p0` as base point, and the vector `p1 - p0` as `u`-direction, and `p3 - p0` as v-direction. Thus vertex attributes `t0,t1,t2,t3` can be bilinearly interpolated over the quadrilateral the following way:

```
t_uv = (1-v)((1-u)*t0 + u*t1) + v*((1-u)*t3 + u*t2)
```

Mixed triangle/quad meshes are supported by encoding a triangle as a quad, which can be achieved by replicating the last triangle vertex (`v0,v1,v2` -> `v0,v1,v2,v2`). This way the second triangle is a line (which can never get hit), and the parametrization of the first triangle is compatible with the standard triangle parametrization.

A quad whose vertices are laid out counter-clockwise has its geometry normal pointing upwards outside the front face, like illustrated in the following picture.



For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers must have the same stride and size.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry

RTC_GEOMETRY_TYPE_GRID

NAME

```
RTC_GEOMETRY_TYPE_GRID - grid geometry type
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_GRID);
```

DESCRIPTION

Grid meshes are created by passing `RTC_GEOMETRY_TYPE_GRID` to the `rtcNewGeometry` function call, and contain an array of grid primitives. This array of grids can be specified by setting up a grid buffer (with `RTC_BUFFER_TYPE_GRID` type and `RTC_FORMAT_GRID` format) and the grid mesh vertices by setting a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` type). See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers. The number of grid primitives in the grid mesh is inferred from the size of the grid buffer.

The vertex buffer contains an array of single precision `x`, `y`, `z` floating point coordinates (`RTC_FORMAT_FLOAT3` format), and the number of vertices is inferred from the size of that buffer.

Each grid in the grid buffer is of the type `RTCGrid`:

```
struct RTCGrid
{
  unsigned int startVertexID;
  unsigned int stride;
  unsigned short width,height;
};
```

The `RTCGrid` structure describes a 2D grid of vertices (with respect to the vertex buffer of the grid mesh). The `width` and `height` members specify the number of vertices in u and v direction, e.g. setting both `width` and `height` to 3 sets up a 3×3 vertex grid. The maximum allowed `width` and `height` is 32767. The `startVertexID` specifies the ID of the top-left vertex in the vertex grid, while the `stride` parameter specifies a stride (in number of vertices) used to step to the next row.

A vertex grid of dimensions `width` and `height` is treated as a `(width-1)` x `(height-1)` grid of `quads` (triangle-pairs), with the same shared edge handling as for regular quad meshes. However, the `u`/`v` coordinates have the uniform range `[0..1]` for an entire vertex grid. The `u` direction follows the `width` of the grid while the `v` direction the `height`.

For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers must have the same stride and size.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry

RTC_GEOMETRY_TYPE_SUBDIVISION

NAME

```
RTC_GEOMETRY_TYPE_SUBDIVISION - subdivision geometry type
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_SUBDIVISION);
```

DESCRIPTION

Catmull-Clark subdivision meshes are supported, including support for edge creases, vertex creases, holes, non-manifold geometry, and face-varying interpolation. The number of vertices per face can be in the range of 3 to 15 vertices (triangles, quadrilateral, pentagons, etc).

Subdivision meshes are created by passing `RTC_GEOMETRY_TYPE_SUBDIVISION` to the `rtcNewGeometry` function. Various buffers need to be set by the application to set up the subdivision mesh. See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers. The face buffer (`RTC_BUFFER_TYPE_FACE` type and `RTC_FORMAT_UINT` format) contains the number of edges/indices of each face (3 to 15), and the number of faces is inferred from the size of this buffer. The index buffer (`RTC_BUFFER_TYPE_INDEX` type) contains multiple (3 to 15) 32-bit vertex indices (`RTC_FORMAT_UINT` format) for each face, and the number of edges is inferred from the size of this buffer. The vertex buffer (`RTC_BUFFER_TYPE_VERTEX` type) stores an array of single precision `x`, `y`, `z` floating point coordinates (`RTC_FORMAT_FLOAT3` format), and the number of vertices is inferred from the size of this buffer.

Optionally, the application may set additional index buffers using different buffer slots if multiple topologies are required for face-varying interpolation. The standard vertex buffers (`RTC_BUFFER_TYPE_VERTEX`) are always bound to the geometry topology (topology 0) thus use `RTC_BUFFER_TYPE_INDEX` with buffer slot 0. User vertex data interpolation may use different topologies as described later.

Optionally, the application can set up the hole buffer (`RTC_BUFFER_TYPE_HOLE`) which contains an array of 32-bit indices (`RTC_FORMAT_UINT` format) of faces that should be considered non-existing in all topologies. The number of holes is inferred from the size of this buffer.

Optionally, the application can fill the level buffer (`RTC_BUFFER_TYPE_LEVEL`) with a tessellation rate for each of the edges of each face. This buffer must have the same size as the index buffer. The tessellation level is a positive floating point value (`RTC_FORMAT_FLOAT` format) that specifies how many quads along the edge should be generated during tessellation. If no level buffer is specified, a level of 1 is used. The maximally supported edge level is 4096, and larger levels are clamped to that value. Note that edges may be shared between (typically 2) faces. To guarantee a watertight tessellation, the level of these shared edges should be identical. A uniform tessellation rate for an entire subdivision mesh can be set by using the `rtcSetGeometryTessellationRate` function. The existence of a level buffer has precedence over the uniform tessellation rate.

Optionally, the application can fill the sparse edge crease buffers to make edges appear sharper. The edge crease index buffer (`RTC_BUFFER_TYPE_EDGE_CREASE_INDEX`) contains an array of pairs of 32-bit vertex indices (`RTC_FORMAT_UINT2` format) that specify unoriented edges in the geometry topology. The edge crease weight buffer (`RTC_BUFFER_TYPE_EDGE_CREASE_WEIGHT`) stores for each of these crease edges a positive floating point weight (`RTC_FORMAT_FLOAT` format). The number of edge creases is inferred from the size of these buffers, which has to be identical. The larger a weight, the sharper the edge. Specifying a weight of infinity is supported and

marks an edge as infinitely sharp. Storing an edge multiple times with the same crease weight is allowed, but has lower performance. Storing an edge multiple times with different crease weights results in undefined behavior. For a stored edge (i,j), the reverse direction edges (j,i) do not have to be stored, as both are considered the same unoriented edge. Edge crease features are shared between all topologies.

Optionally, the application can fill the sparse vertex crease buffers to make vertices appear sharper. The vertex crease index buffer (`RTC_BUFFER_TYPE_VERTEX_CREASE_INDEX`), contains an array of 32-bit vertex indices (`RTC_FORMAT_UINT` format) to specify a set of vertices from the geometry topology. The vertex crease weight buffer (`RTC_BUFFER_TYPE_VERTEX_CREASE_WEIGHT`) specifies for each of these vertices a positive floating point weight (`RTC_FORMAT_FLOAT` format). The number of vertex creases is inferred from the size of these buffers, and has to be identical. The larger a weight, the sharper the vertex. Specifying a weight of infinity is supported and makes the vertex infinitely sharp. Storing a vertex multiple times with the same crease weight is allowed, but has lower performance. Storing a vertex multiple times with different crease weights results in undefined behavior. Vertex crease features are shared between all topologies.

Subdivision modes can be used to force linear interpolation for parts of the subdivision mesh; see `rtcSetGeometrySubdivisionMode` for more details.

For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers have to have the same stride and size.

Also see tutorial Subdivision Geometry for an example of how to create subdivision surfaces.

Parametrization

The parametrization for subdivision faces is different for quadrilaterals and non-quadrilateral faces.

The parametrization of a quadrilateral face uses the first vertex `p0` as base point, and the vector `p1 - p0` as u-direction and `p3 - p0` as v-direction.

The parametrization for all other face types (with number of vertices not equal 4), have a special parametrization where the subpatch ID `n` (of the `n`-th quadrilateral that would be obtained by a single subdivision step) and the local hit location inside this quadrilateral are encoded in the UV coordinates. The following code extracts the sub-patch ID `i` and local UVs of this subpatch:

```
unsigned int l = floorf(0.5f*U);
unsigned int h = floorf(0.5f*V);
unsigned int i = 4*h+l;
float u = 2.0f*fracf(0.5f*U)-0.5f;
float v = 2.0f*fracf(0.5f*V)-0.5f;
```

This encoding allows local subpatch UVs to be in the range `[-0.5,1.5[` thus negative subpatch UVs can be passed to `rtcInterpolate` to sample subpatches slightly out of bounds. This can be useful to calculate derivatives using finite differences if required. The encoding further has the property that one can just move the value `u` (or `v`) on a subpatch by adding `du` (or `dv`) to the special UV encoding as long as it does not fall out of the `[-0.5,1.5[` range.

To smoothly interpolate vertex attributes over the subdivision surface we recommend using the `rtcInterpolate` function, which will apply the standard subdivision rules for interpolation and automatically takes care of the special UV encoding for non-quadrilaterals.

Face-Varying Data

Face-varying interpolation is supported through multiple topologies per subdivision mesh and binding such topologies to vertex attribute buffers to interpolate. This way, texture coordinates may use a different topology with additional boundaries to construct separate UV regions inside one subdivision mesh.

Each such topology `i` has a separate index buffer (specified using `RTC_BUFFER_TYPE_INDEX` with buffer slot `i`) and separate subdivision mode that can be set using `rtcSetGeometrySubdivisionMode`. A vertex attribute buffer `RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE` bound to a buffer slot `j` can be assigned to use a topology for interpolation using the `rtcSetGeometryVertexAttributeTopology` call.

The face buffer (`RTC_BUFFER_TYPE_FACE` type) is shared between all topologies, which means that the `n`-th primitive always has the same number of vertices (e.g. being a triangle or a quad) for each topology. However, the indices of the topologies themselves may be different.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry

RTC_GEOMETRY_TYPE_CURVE

NAME

```
RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE -
  flat curve geometry with linear basis

RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE -
  flat curve geometry with cubic Bézier basis

RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE -
  flat curve geometry with cubic B-spline basis

RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE -
  flat curve geometry with cubic Hermite basis

RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE -
  flat curve geometry with Catmull-Rom basis

RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BEZIER_CURVE -
  flat normal oriented curve geometry with cubic Bézier basis

RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BSPLINE_CURVE -
  flat normal oriented curve geometry with cubic B-spline basis

RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_HERMITE_CURVE -
  flat normal oriented curve geometry with cubic Hermite basis

RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_CATMULL_ROM_CURVE -
  flat normal oriented curve geometry with Catmull-Rom basis

RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE -
  capped cone curve geometry with linear basis - discontinous at edge boundaries

RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE -
  capped cone curve geometry with linear basis and spherical ending

RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE -
  swept surface curve geometry with cubic Bézier basis

RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE -
  swept surface curve geometry with cubic B-spline basis

RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE -
  swept surface curve geometry with cubic Hermite basis

RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE -
  swept surface curve geometry with Catmull-Rom basis
```

SYNOPSIS

```
#include <embree3/rtcore.h>

rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BEZIER_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_BSPLINE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_HERMITE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_CATMULL_ROM_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE);
```

DESCRIPTION

Curves with per vertex radii are supported with linear, cubic Bézier, cubic B-spline, and cubic Hermite bases. Such curve geometries are created by passing `RTC_GEOMETRY_TYPE_FLAT_LINEAR_CURVE`, `RTC_GEOMETRY_TYPE_FLAT_BEZIER_CURVE`, `RTC_GEOMETRY_TYPE_FLAT_BSPLINE_CURVE`, `RTC_GEOMETRY_TYPE_FLAT_HERMITE_CURVE`, `RTC_GEOMETRY_TYPE_FLAT_CATMULL_ROM_CURVE`, `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_FLAT_BEZIER_CURVE`, `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_FLAT_` `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_FLAT_HERMITE_CURVE`, `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_FLAT` `RTC_GEOMETRY_TYPE_CONE_LINEAR_CURVE`, `RTC_GEOMETRY_TYPE_ROUND_LINEAR_CURVE`, `RTC_GEOMETRY_TYPE_ROUND_BEZIER_CURVE`, `RTC_GEOMETRY_TYPE_ROUND_BSPLINE_CURVE`, `RTC_GEOMETRY_TYPE_ROUND_HERMITE_CURVE`, or `RTC_GEOMETRY_TYPE_ROUND_CATMULL_ROM_CURVE` to the `rtcNewGeometry` function. The curve indices can be specified through an index buffer (`RTC_BUFFER_TYPE_INDEX`) and the curve vertices through a vertex buffer (`RTC_BUFFER_TYPE_VERTEX`). For the Hermite basis a tangent buffer (`RTC_BUFFER_TYPE_TANGENT`), normal oriented curves a normal buffer (`RTC_BUFFER_TYPE_NORMAL`), and for normal oriented Hermite curves a normal derivative buffer (`RTC_BUFFER_TYPE_NORMAL_DERIVATIVE`) has to get specified additionally. See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers.

The index buffer contains an array of 32-bit indices (`RTC_FORMAT_UINT` format), each pointing to the first control vertex in the vertex buffer, but also to the first tangent in the tangent buffer, and first normal in the normal buffer if these buffers are present.

The vertex buffer stores each control vertex in the form of a single precision position and radius stored in (`x`, `y`, `z`, `r`) order in memory (`RTC_FORMAT_FLOAT4` format). The number of vertices is inferred from the size of this buffer. The radii may be smaller than zero but the interpolated radii should always be greater or equal to zero. Similarly, the tangent buffer stores the derivative of each control vertex (`x`, `y`, `z`, `r` order and `RTC_FORMAT_FLOAT4` format) and the normal buffer stores a single precision normal per control vertex (`x`, `y`, `z` order and `RTC_FORMAT_FLOAT3` format).

Linear Basis

For the linear basis the indices point to the first of 2 consecutive control points in the vertex buffer. The first control point is the start and the second control point the end of the line segment. When constructing hair strands in this basis, the end-point can be shared with the start of the next line segment.

For the linear basis the user optionally can provide a flags buffer of type `RTC_BUFFER_TYPE_FLAGS` which contains bytes that encode if the left neighbor segment (`RTC_CURVE_FLAG_NEIGHBOR_LEFT` flag) and/or right neighbor segment (`RTC_CURVE_FLAG_NEIGHBOR_RIGHT` flags) exist (see *RTCCurveFlags*). If this buffer is not set, than the left/right neighbor bits are automatically calculated base on the index buffer (left segment exists if segment(id-1)+1 == segment(id) and right segment exists if segment(id+1)-1 == segment(id)).

A left neighbor segment is assumed to end at the start vertex of the current segement, and to start at the previous vertex in the vertex buffer. Similarly, the right neighbor segment is assumed to start at the end vertex of the current segment, and to end at the next vertex in the vertex buffer.

Only when the left and right bits are properly specified the current segment can properly attach to the left and/or right neighbor, otherwise the touching area may not get rendererd properly.

Bézier Basis

For the cubic Bézier basis the indices point to the first of 4 consecutive control points in the vertex buffer. These control points use the cubic Bézier basis, where the first control point represents the start point of the curve, and the 4th control point the end point of the curve. The Bézier basis is interpolating, thus the curve does go exactly through the first and fourth control vertex.

B-spline Basis

For the cubic B-spline basis the indices point to the first of 4 consecutive control points in the vertex buffer. These control points make up a cardinal cubic B-spline (implicit equidistant knot vector). This basis is not interpolating, thus the curve does in general not go through any of the control points directly. A big advantage of this basis is that 3 control points can be shared for two continuous neighboring curve segments, e.g. the curves (p0,p1,p2,p3) and (p1,p2,p3,p4) are C1 continuous. This feature make this basis a good choise to construct continuous multi-segment curves, as memory consumption can be kept minimal.

Hermite Basis

For the cubic Hermite basis the indices point to the first of 2 consecutive points in the vertex buffer, and the first of 2 consecutive tangents in the tangent buffer. These two points and two tangents make up a cubic Hermite curve. This basis is interpolating, thus does exactly go through the first and second control point, and the first order derivative at the begin and end matches exactly the value specified in the tangent buffer. When connecting two segments continuously, the end point and tangent of the previous segment can be shared. Different versions of Catmull-Rom splines can be easily constructed usig the Hermite basis, by calculating a proper tangent buffer from the control points.

Catmull-Rom Basis

For the Catmull-Rom basis the indices point to the first of 4 consecutive control points in the vertex buffer. This basis goes through p1 and p2, with tangents (p2-p0)/2 and (p3-p1)/2.

Flat Curves

The `RTC_GEOMETRY_TYPE_FLAT_*` flat mode is a fast mode designed to render distant hair. In this mode the curve is rendered as a connected sequence of ray facing quads. Individual quads are considered to have subpixel size, and zooming onto the curve might show geometric artifacts. The number of quads to subdivide into can be specified through the `rtcSetGeometryTessellationRate` function. By default the tessellation rate is 4.

Normal Oriented Curves

The `RTC_GEOMETRY_TYPE_NORMAL_ORIENTED_*` mode is a mode designed to render blades of grass. In this mode a vertex spline has to get specified as for the previous modes, but additionally a normal spline is required. If the Hermite basis is used, the `RTC_BUFFER_TYPE_NORMAL` and `RTC_BUFFER_TYPE_NORMAL_DERIVATIVE` buffers have both to be set.

The curve is rendered as a flat band whose center approximately follows the provided vertex spline, whose half width approximately follows the provided radius spline, and whose normal orientation approximately follows the provided normal spline.

To intersect the normal oriented curve, we perform a newton-raphson style intersection of a ray with a tensor product surface of a linear basis (perpendicular to the curve) and cubic Bézier basis (along the curve). We use a guide curve and its derivatives to construct the control points of that surface. The guide curve is defined by a sweep surface defined by sweeping a line centered at the vertex spline location along the curve. At each parameter value the half width of the line matches the radius spline, and the direction matches the cross product of the normal from the normal spline and tangent of the vertex spline. Note that this construction does not work when the provided normals are parallel to the curve direction. For this reason the provided normals should best be kept as perpendicular to the curve direction as possible.

Round Curves

In the `RTC_GEOMETRY_TYPE_ROUND_*` round mode, a real geometric surface is rendered for the curve, which is more expensive but allows closeup views.

For the linear basis the round mode renders a cone that tangentially touches a start-sphere and end-sphere. The start sphere is rendered when no previous segments is indicated by the neighbor bits. The end sphere is always rendered but parts that lie inside the next segment are clipped away (if that next segment exists). This way a curve is closed on both ends and the interiour will render properly as long as only neighboring segments penetrate into a segment. For this to work properly it is important that the flags buffer is properly populated with neighbor information.

For the cubic polynomial bases, the round mode renders a sweep surface by sweeping a varying radius circle tangential along the curve. As a limitation, the radius of the curve has to be smaller than the curvature radius of the curve at each location on the curve.

The intersection with the curve segment stores the parametric hit location along the curve segment as u-coordinate (range 0 to +1).

For flat curves, the v-coordinate is set to the normalized distance in the range -1 to +1. For normal oriented curves the v-coordinate is in the range 0 to 1. For the linear basis and in round mode the v-coordinate is set to zero.

In flat mode, the geometry normal `Ng` is set to the tangent of the curve at the hit location. In round mode and for normal oriented curves, the geometry normal `Ng` is set to the non-normalized geometric normal of the surface.

For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers must have the same stride and size. For the Hermite basis also a tangent buffer has to be set for each time step and for normal oriented curves a normal buffer has to get specified for each time step.

Also see tutorials Hair and Curves for examples of how to create and use curve geometries.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *RTCCurveFlags*

RTC_GEOMETRY_TYPE_POINT

NAME

```
RTC_GEOMETRY_TYPE_SPHERE_POINT -
  point geometry spheres

RTC_GEOMETRY_TYPE_DISC_POINT -
  point geometry with ray-oriented discs

RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT -
  point geometry with normal-oriented discs
```

SYNOPSIS

```
#include <embree3/rtcore.h>

rtcNewGeometry(device, RTC_GEOMETRY_TYPE_SPHERE_POINT);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_DISC_POINT);
rtcNewGeometry(device, RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT);
```

DESCRIPTION

Points with per vertex radii are supported with sphere, ray-oriented discs, and normal-oriented discs geometric representations. Such point geometries are created by passing `RTC_GEOMETRY_TYPE_SPHERE_POINT`, `RTC_GEOMETRY_TYPE_DISC_POINT`, or `RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT` to the `rtcNewGeometry` function. The point vertices can be specified t through a vertex buffer (`RTC_BUFFER_TYPE_VERTEX`). For the normal oriented discs a normal buffer (`RTC_BUFFER_TYPE_NORMAL`) has to get specified additionally. See `rtcSetGeometryBuffer` and `rtcSetSharedGeometryBuffer` for more details on how to set buffers.

The vertex buffer stores each control vertex in the form of a single precision position and radius stored in (`x`, `y`, `z`, `r`) order in memory (`RTC_FORMAT_FLOAT4` format). The number of vertices is inferred from the size of this buffer. Similarly, the normal buffer stores a single precision normal per control vertex (`x`, `y`, `z` order and `RTC_FORMAT_FLOAT3` format).

In the `RTC_GEOMETRY_TYPE_SPHERE_POINT` mode, a real geometric surface is rendered for the curve, which is more expensive but allows closeup views.

The `RTC_GEOMETRY_TYPE_DISC_POINT` flat mode is a fast mode designed to render distant points. In this mode the point is rendered as a ray facing disc.

The `RTC_GEOMETRY_TYPE_ORIENTED_DISC_POINT` mode is a mode designed as a midpoint geometrically between ray facing discs and spheres. In this mode the point is rendered as a normal oriented disc.

For all point types, only the hit distance and geometry normal is returned as hit information, u and v are set to zero.

For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` call. Then a vertex buffer for each time step can be set using different buffer slots, and all these buffers must have the same stride and size.

Also see tutorial [Points] for an example of how to create and use point geometries.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry

RTC_GEOMETRY_TYPE_USER

NAME

```
RTC_GEOMETRY_TYPE_USER - user geometry type
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCGeometry geometry =
  rtcNewGeometry(device, RTC_GEOMETRY_TYPE_USER);
```

DESCRIPTION

User-defined geometries contain a number of user-defined primitives, just like triangle meshes contain multiple triangles. The shape of the user-defined primitives is specified through registered callback functions, which enable extending Embree with arbitrary types of primitives.

User-defined geometries are created by passing `RTC_GEOMETRY_TYPE_USER` to the `rtcNewGeometry` function call. One has to set the number of primitives (see `rtcSetGeometryUserPrimitiveCount`), a user data pointer (see `rtcSetGeometryUserData`), a bounding function closure (see `rtcSetGeometryBoundsFunction`), as well as user-defined intersect (see `rtcSetGeometryIntersectFunction`) and occluded (see `rtcSetGeometryOccludedFunction`) callback functions. The bounding function is used to query the bounds of all time steps of a user primitive, while the intersect and occluded callback functions are called to intersect the primitive with a ray. The user data pointer is passed to each callback invocation and can be used to point to the application's representation of the user geometry.

The creation of a user geometry typically looks the following:

```
RTCGeometry geometry = rtcNewGeometry(device, RTC_GEOMETRY_TYPE_USER);
rtcSetGeometryUserPrimitiveCount(geometry, numPrimitives);
rtcSetGeometryUserData(geometry, userGeometryRepresentation);
rtcSetGeometryBoundsFunction(geometry, boundsFunction);
rtcSetGeometryIntersectFunction(geometry, intersectFunction);
rtcSetGeometryOccludedFunction(geometry, occludedFunction);
```

Please have a look at the `rtcSetGeometryBoundsFunction`, `rtcSetGeometryIntersectFunction`, and `rtcSetGeometryOccludedFunction` functions on the implementation of the callback functions.

Primitives of a user geometry are ignored during rendering when their bounds are empty, thus bounds have lower>upper in at least one dimension.

See tutorial User Geometry for an example of how to use the user-defined geometries.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *rtcSetGeometryUserPrimitiveCount*, *rtcSetGeometryUserData*, *rtcSetGeometryBoundsFunction*, *rtcSetGeometryIntersectFunction*, *rtcSetGeometryOccludedFunction*

RTC_GEOMETRY_TYPE_INSTANCE

NAME

```
RTC_GEOMETRY_TYPE_INSTANCE - instance geometry type
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCGeometry geometry =
   rtcNewGeometry(device, RTC_GEOMETRY_TYPE_INSTANCE);
```

DESCRIPTION

Embree supports instancing of scenes using affine transformations (3×3 matrix plus translation). As the instanced scene is stored only a single time, even if instanced to multiple locations, this feature can be used to create very complex scenes with small memory footprint.

Embree supports both single-level instancing and multi-level instancing. The maximum instance nesting depth is `RTC_MAX_INSTANCE_LEVEL_COUNT`; it can be configured at compile-time using the constant `EMBREE_MAX_INSTANCE_LEVEL_COUNT`. Users should adapt this constant to their needs: instances nested any deeper are silently ignored in release mode, and cause assertions in debug mode.

Instances are created by passing `RTC_GEOMETRY_TYPE_INSTANCE` to the `rtcNewGeometry` function call. The instanced scene can be set using the `rtcSetGeometryInstancedScene` call, and the affine transformation can be set using the `rtcSetGeometryTransform` function.

Please note that `rtcCommitScene` on the instanced scene should be called first, followed by `rtcCommitGeometry` on the instance, followed by `rtcCommitScene` for the top-level scene containing the instance.

If a ray hits the instance, the `geomID` and `primID` members of the hit are set to the geometry ID and primitive ID of the hit primitive in the instanced scene, and the `instID` member of the hit is set to the geometry ID of the instance in the top-level scene.

The instancing scheme can also be implemented using user geometries. To achieve this, the user geometry code should set the `instID` member of the intersection context to the geometry ID of the instance, then trace the transformed ray, and finally set the `instID` field of the intersection context again to -1. The `instID` field is copied automatically by each primitive intersector into the `instID` field of the hit structure when the primitive is hit. See the User Geometry tutorial for an example.

For multi-segment motion blur, the number of time steps must be first specified using the `rtcSetGeometryTimeStepCount` function. Then a transformation for each time step can be specified using the `rtcSetGeometryTransform` function.

See tutorials Instanced Geometry and Multi Level Instancing for examples of how to use instances.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *rtcSetGeometryInstancedScene*, *rtcSetGeometryTransform*

RTCCurveFlags

NAME

```
RTCCurveFlags - per segment flags for curve geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>
```

enum RTCCurveFlags { RTC_CURVE_FLAG_NEIGHBOR_LEFT = (1 << 0), RTC_CURVE_FLAG_NEIGHBOR_RIGHT = (1 << 1) };

DESCRIPTION

The RTCCurveFlags type is used for linear curves to determine if the left and/or right neighbor segment exist. Therefore one attaches a buffer of type RTC_BUFFER_TYPE_FLAGS to the curve geometry which stores an individual byte per curve segment.

If the RTC_CURVE_FLAG_NEIGHBOR_LEFT flag in that byte is enabled for a curve segment, then the left segment exists (which starts one vertex before the start vertex of the current curve) and the current segment is rendered to properly attach to that segment.

If the RTC_CURVE_FLAG_NEIGHBOR_RIGHT flag in that byte is enabled for a curve segment, then the right segment exists (which ends one vertex after the end vertex of the current curve) and the current segment is rendered to properly attach to that segment.

When not properly specifying left and right flags for linear curves, the rendering at the ending of these curves may not look correct, in particular when round linear curves are viewed from the inside.

EXIT STATUS

SEE ALSO

RTC_GEOMETRY_TYPE_CURVE

rtcRetainGeometry

NAME

```
rtcRetainGeometry - increments the geometry reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcRetainGeometry(RTCGeometry geometry);
```

DESCRIPTION

Geometry objects are reference counted. The `rtcRetainGeometry` function increments the reference count of the passed geometry object (`geometry` argument). This function together with `rtcReleaseGeometry` allows to use the internal reference counting in a C++ wrapper class to handle the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *rtcReleaseGeometry*

rtcReleaseGeometry

NAME

```
rtcReleaseGeometry - decrements the geometry reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcReleaseGeometry(RTCGeometry geometry);
```

DESCRIPTION

Geometry objects are reference counted. The `rtcReleaseGeometry` function decrements the reference count of the passed geometry object (`geometry` argument). When the reference count falls to 0, the geometry gets destroyed.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *rtcRetainGeometry*

rtcCommitGeometry

NAME

```
rtcCommitGeometry - commits geometry changes
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcCommitGeometry(RTCGeometry geometry);
```

DESCRIPTION

The `rtcCommitGeometry` function is used to commit all geometry changes performed to a geometry (`geometry` parameter). After a geometry gets modified, this function must be called to properly update the internal state of the geometry to perform interpolations using `rtcInterpolate` or to commit a scene containing the geometry using `rtcCommitScene`.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcInterpolate, *rtcCommitScene*

rtcEnableGeometry

NAME

```
rtcEnableGeometry - enables the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcEnableGeometry(RTCGeometry geometry);
```

DESCRIPTION

The `rtcEnableGeometry` function enables the specified geometry (`geometry` argument). Only enabled geometries are rendered. Each geometry is enabled by default at construction time.

After enabling a geometry, the scene containing that geometry must be committed using `rtcCommitScene` for the change to have effect.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *rtcDisableGeometry*, *rtcCommitScene*

rtcDisableGeometry

NAME

```
rtcDisableGeometry - disables the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcDisableGeometry(RTCGeometry geometry);
```

DESCRIPTION

The `rtcDisableGeometry` function disables the specified geometry (`geometry` argument). A disabled geometry is not rendered. Each geometry is enabled by default at construction time.

After disabling a geometry, the scene containing that geometry must be committed using `rtcCommitScene` for the change to have effect.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *rtcEnableGeometry*, *rtcCommitScene*

rtcSetGeometryTimeStepCount

NAME

```
rtcSetGeometryTimeStepCount - sets the number of time steps of the
  geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryTimeStepCount(
  RTCGeometry geometry,
  unsigned int timeStepCount
);
```

DESCRIPTION

The `rtcSetGeometryTimeStepCount` function sets the number of time steps for multi-segment motion blur (`timeStepCount` parameter) of the specified geometry (`geometry` parameter).

For triangle meshes (`RTC_GEOMETRY_TYPE_TRIANGLE`), quad meshes (`RTC_GEOMETRY_TYPE_QUAD`), curves (`RTC_GEOMETRY_TYPE_CURVE`), points (`RTC_GEOMETRY_TYPE_POINT`), and subdivision geometries (`RTC_GEOMETRY_TYPE_SUBDIVISION`), the number of time steps directly corresponds to the number of vertex buffer slots available (`RTC_BUFFER_TYPE_VERTEX` buffer type). For these geometries, one vertex buffer per time step must be specified when creating multi-segment motion blur geometries.

For instance geometries (`RTC_GEOMETRY_TYPE_INSTANCE`), a transformation must be specified for each time step (see `rtcSetGeometryTransform`).

For user geometries, the registered bounding callback function must provide a bounding box per primitive and time step, and the intersection and occlusion callback functions should properly intersect the motion-blurred geometry at the ray time.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *rtcSetGeometryTimeRange*

rtcSetGeometryTimeRange

NAME

```
rtcSetGeometryTimeRange - sets the time range for a motion blur geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryTimeRange(
  RTCGeometry geometry,
  float startTime,
  float endTime
);
```

DESCRIPTION

The `rtcSetGeometryTimeRange` function sets a time range which defines the start (and end time) of the first (and last) time step of a motion blur geometry. The time range is defined relative to the camera shutter interval [0,1] but it can be arbitrary. Thus the startTime can be smaller, equal, or larger 0, indicating a geometry whose animation definition start before, at, or after the camera shutter opens. Similar the endTime can be smaller, equal, or larger than 1, indicating a geometry whose animation definition ends after, at, or before the camera shutter closes. The startTime has to be smaller or equal to the endTime.

The default time range when this function is not called is the entire camera shutter [0,1]. For best performance at most one time segment of the piece wise linear definition of the motion should fall outside the shutter window to the left and to the right. Thus do not set the startTime or endTime too far outside the [0,1] interval for best performance.

This time range feature will also allow geometries to appear and disappear during the camera shutter time if the specified time range is a sub range of [0,1].

Please also have a look at the `rtcSetGeometryTimeStepCount` function to see how to define the time steps for the specified time range.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryTimeStepCount

rtcSetGeometryVertexAttributeCount

NAME

```
rtcSetGeometryVertexAttributeCount - sets the number of vertex
  attributes of the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryVertexAttributeCount(
  RTCGeometry geometry,
  unsigned int vertexAttributeCount
);
```

DESCRIPTION

The `rtcSetGeometryVertexAttributeCount` function sets the number of slots (`vertexAttributeCount` parameter) for vertex attribute buffers (`RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`) that can be used for the specified geometry (`geometry` parameter).

This function is supported only for triangle meshes (`RTC_GEOMETRY_TYPE_TRIANGLE`), quad meshes (`RTC_GEOMETRY_TYPE_QUAD`), curves (`RTC_GEOMETRY_TYPE_CURVE`), points (`RTC_GEOMETRY_TYPE_POINT`), and subdivision geometries (`RTC_GEOMETRY_TYPE_SUBDIVISION`).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *RTCBufferType*

rtcSetGeometryMask

NAME

```
rtcSetGeometryMask - sets the geometry mask
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryMask(
  RTCGeometry geometry,
  unsigned int mask
);
```

DESCRIPTION

The `rtcSetGeometryMask` function sets a 32-bit geometry mask (`mask` argument) for the specified geometry (`geometry` argument).

This geometry mask is used together with the ray mask stored inside the `mask` field of the ray. The primitives of the geometry are hit by the ray only if the bitwise `and` operation of the geometry mask with the ray mask is not 0. This feature can be used to disable selected geometries for specifically tagged rays, e.g. to disable shadow casting for certain geometries.

Ray masks are disabled in Embree by default at compile time, and can be enabled through the `EMBREE_RAY_MASK` parameter in CMake. One can query whether ray masks are enabled by querying the `RTC_DEVICE_PROPERTY_RAY_MASK_SUPPORTED` device property using `rtcGetDeviceProperty`.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTCRay, *rtcGetDeviceProperty*

rtcSetGeometryBuildQuality

NAME

```
rtcSetGeometryBuildQuality - sets the build quality for the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryBuildQuality(
  RTCGeometry geometry,
  enum RTCBuildQuality quality
);
```

DESCRIPTION

The `rtcSetGeometryBuildQuality` function sets the build quality (`quality` argument) for the specified geometry (`geometry` argument). The per-geometry build quality is only a hint and may be ignored. Embree currently uses the per-geometry build quality when the scene build quality is set to `RTC_BUILD_QUALITY_LOW`. In this mode a two-level acceleration structure is build, and geometries build a separate acceleration structure using the geometry build quality. The per-geometry build quality can be one of:

- `RTC_BUILD_QUALITY_LOW`: Creates lower quality data structures, e.g. for dynamic scenes.
- `RTC_BUILD_QUALITY_MEDIUM`: Default build quality for most usages. Gives a good compromise between build and render performance.
- `RTC_BUILD_QUALITY_HIGH`: Creates higher quality data structures for final-frame rendering. Enables a spatial split builder for certain primitive types.
- `RTC_BUILD_QUALITY_REFIT`: Uses a BVH refitting approach when changing only the vertex buffer.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetSceneBuildQuality

rtcSetGeometryBuffer

NAME

```
rtcSetGeometryBuffer - assigns a view of a buffer to the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryBuffer(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot,
  enum RTCFormat format,
  RTCBuffer buffer,
  size_t byteOffset,
  size_t byteStride,
  size_t itemCount
);
```

DESCRIPTION

The `rtcSetGeometryBuffer` function binds a view of a buffer object (`buffer` argument) to a geometry buffer type and slot (`type` and `slot` argument) of the specified geometry (`geometry` argument).

One can specify the start of the first buffer element in bytes (`byteOffset` argument), the byte stride between individual buffer elements (`byteStride` argument), the format of the buffer elements (`format` argument), and the number of elements to bind (`itemCount`).

The start address (`byteOffset` argument) and stride (`byteStride` argument) must be both aligned to 4 bytes, otherwise the `rtcSetGeometryBuffer` function will fail.

After successful completion of this function, the geometry will hold a reference to the buffer object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetSharedGeometryBuffer, *rtcSetNewGeometryBuffer*

rtcSetSharedGeometryBuffer

NAME

```
rtcSetSharedGeometryBuffer - assigns a view of a shared data buffer
  to a geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetSharedGeometryBuffer(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot,
  enum RTCFormat format,
  const void* ptr,
  size_t byteOffset,
  size_t byteStride,
  size_t itemCount
);
```

DESCRIPTION

The `rtcSetSharedGeometryBuffer` function binds a view of a shared user-managed data buffer (`ptr` argument) to a geometry buffer type and slot (`type` and `slot` argument) of the specified geometry (`geometry` argument).

One can specify the start of the first buffer element in bytes (`byteOffset` argument), the byte stride between individual buffer elements (`byteStride` argument), the format of the buffer elements (`format` argument), and the number of elements to bind (`itemCount`).

The start address (`byteOffset` argument) and stride (`byteStride` argument) must be both aligned to 4 bytes; otherwise the `rtcSetGeometryBuffer` function will fail.

When the buffer will be used as a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` and `RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`), the last buffer element must be readable using 16-byte SSE load instructions, thus padding the last element is required for certain layouts. E.g. a standard `float3` vertex buffer layout should add storage for at least one more float to the end of the buffer.

The buffer data must remain valid for as long as the buffer may be used, and the user is responsible for freeing the buffer data when no longer required.

Sharing buffers can significantly reduce the memory required by the application, thus we recommend using this feature. When enabling the `RTC_SCENE_COMPACT` scene flag, the spatial index structures index into the vertex buffer, resulting in even higher memory savings.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryBuffer, *rtcSetNewGeometryBuffer*

rtcSetNewGeometryBuffer

NAME

```
rtcSetNewGeometryBuffer - creates and assigns a new data buffer to
  the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void* rtcSetNewGeometryBuffer(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot,
  enum RTCFormat format,
  size_t byteStride,
  size_t itemCount
);
```

DESCRIPTION

The `rtcSetNewGeometryBuffer` function creates a new data buffer of specified format (`format` argument), byte stride (`byteStride` argument), and number of items (`itemCount` argument), and assigns it to a geometry buffer slot (`type` and `slot` argument) of the specified geometry (`geometry` argument). The buffer data is managed internally and automatically freed when the geometry is destroyed.

The byte stride (`byteStride` argument) must be aligned to 4 bytes; otherwise the `rtcSetNewGeometryBuffer` function will fail.

The allocated buffer will be automatically over-allocated slightly when used as a vertex buffer, where a requirement is that each buffer element should be readable using 16-byte SSE load instructions.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryBuffer, *rtcSetSharedGeometryBuffer*

RTCFormat

NAME

```
RTCFormat - specifies format of data in buffers
```

SYNOPSIS

```
#include <embree3/rtcore_ray.h>

enum RTCFormat
{
  RTC_FORMAT_UINT,
  RTC_FORMAT_UINT2,
  RTC_FORMAT_UINT3,
  RTC_FORMAT_UINT4,

  RTC_FORMAT_FLOAT,
  RTC_FORMAT_FLOAT2,
  RTC_FORMAT_FLOAT3,
  RTC_FORMAT_FLOAT4,
  RTC_FORMAT_FLOAT5,
  RTC_FORMAT_FLOAT6,
  RTC_FORMAT_FLOAT7,
  RTC_FORMAT_FLOAT8,
  RTC_FORMAT_FLOAT9,
  RTC_FORMAT_FLOAT10,
  RTC_FORMAT_FLOAT11,
  RTC_FORMAT_FLOAT12,
  RTC_FORMAT_FLOAT13,
  RTC_FORMAT_FLOAT14,
  RTC_FORMAT_FLOAT15,
  RTC_FORMAT_FLOAT16,

  RTC_FORMAT_FLOAT3X4_ROW_MAJOR,
  RTC_FORMAT_FLOAT4X4_ROW_MAJOR,

  RTC_FORMAT_FLOAT3X4_COLUMN_MAJOR,
  RTC_FORMAT_FLOAT4X4_COLUMN_MAJOR,

  RTC_FORMAT_GRID,
```

};

DESCRIPTION

The `RTFormat` structure defines the data format stored in data buffers provided to Embree using the *rtcSetGeometryBuffer*, *rtcSetSharedGeometryBuffer*, and *rtcSetNewGeometryBuffer* API calls.

The `RTC_FORMAT_UINT/2/3/4` format are used to specify that data buffers store unsigned integers, or unsigned integer vectors of size 2,3 or 4. This format has typically to get used when specifying index buffers, e.g. `RTC_FORMAT_UINT3` for triangle meshes.

The `RTC_FORMAT_FLOAT/2/3/4...` format are used to specify that data buffers store single precision floating point values, or vectors there of (size 2,3,4, etc.). This format is typcally used to specify to format of vertex buffers, e.g. the `RTC_FORMAT_FLOAT3` type for vertex buffers of triangle meshes.

The `RTC_FORMAT_FLOAT3X4_ROW_MAJOR` and `RTC_FORMAT_FLOAT3X4_COLUMN_MAJOR` formats, specify a 3x4 floating point matrix layed out either row major or column major. The `RTC_FORMAT_FLOAT4X4_ROW_MAJOR` and `RTC_FORMAT_FLOAT4X4_COLUMN_MAJOR` formats, specify a 4x4 floating point matrix layed out either row major or column major. These matrix formats are used in the *rtcSetGeometryTransform* function in order to set a transformation matrix for geometries.

The `RTC_FORMAT_GRID` is a special data format used to specify grid primitives of layout RTCGrid when creating grid geometries (see *RTC_GEOMETRY_TYPE_GRID*).

EXIT STATUS

SEE ALSO

rtcSetGeometryBuffer, *rtcSetSharedGeometryBuffer*, *rtcSetNewGeometryBuffer*, *rtcSetGeometryTransform*

RTCBufferType

NAME

```
RTCFormat - specifies format of data in buffers
```

SYNOPSIS

```
#include <embree3/rtcore_ray.h>

enum RTCBufferType
{
  RTC_BUFFER_TYPE_INDEX            = 0,
  RTC_BUFFER_TYPE_VERTEX           = 1,
  RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE = 2,
  RTC_BUFFER_TYPE_NORMAL           = 3,
  RTC_BUFFER_TYPE_TANGENT          = 4,
  RTC_BUFFER_TYPE_NORMAL_DERIVATIVE = 5,

  RTC_BUFFER_TYPE_GRID                = 8,

  RTC_BUFFER_TYPE_FACE                = 16,
  RTC_BUFFER_TYPE_LEVEL               = 17,
  RTC_BUFFER_TYPE_EDGE_CREASE_INDEX   = 18,
  RTC_BUFFER_TYPE_EDGE_CREASE_WEIGHT  = 19,
  RTC_BUFFER_TYPE_VERTEX_CREASE_INDEX  = 20,
  RTC_BUFFER_TYPE_VERTEX_CREASE_WEIGHT = 21,
  RTC_BUFFER_TYPE_HOLE                = 22,

  RTC_BUFFER_TYPE_FLAGS = 32
};
```

DESCRIPTION

The `RTBufferType` structure defines slots to assign data buffers to using the *rtcSetGeometryBuffer*, *rtcSetSharedGeometryBuffer*, and *rtcSetNewGeometryBuffer* API calls.

For most geometry types the `RTC_BUFFER_TYPE_INDEX` slot is used to assign an index buffer, while the `RTC_BUFFER_TYPE_VERTEX` is used to assign the corresponding vertex buffer.

The `RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE` slot can get used to assign arbitrary additional vertex data which can get interpolated using the *rtcInterpolate* API call.

The `RTC_BUFFER_TYPE_NORMAL`, `RTC_BUFFER_TYPE_TANGENT`, and `RTC_BUFFER_TYPE_NORMAL_DERIVATIVE` are special buffers required to assign per vertex normals, tangents, and normal derivatives for some curve types.

The `RTC_BUFFER_TYPE_GRID` buffer is used to assign the grid primitive buffer for grid geometries (see *RTC_GEOMETRY_TYPE_GRID*).

The `RTC_BUFFER_TYPE_FACE`, `RTC_BUFFER_TYPE_LEVEL`, `RTC_BUFFER_TYPE_EDGE_CREASE_INDEX`, `RTC_BUFFER_TYPE_EDGE_CREASE_WEIGHT`, `RTC_BUFFER_TYPE_VERTEX_CREASE_INDEX`, `RTC_BUFFER_TYPE_VERTEX_CREASE_WEIGHT`, and `RTC_BUFFER_TYPE_HOLE` are special buffers required to create subdivision meshes (see *RTC_GEOMETRY_TYPE_SUBDIVISION*).

The `RTC_BUFFER_TYPE_FLAGS` can get used to add additional flag per primitive of a geometry, and is currently only used for linear curves.

EXIT STATUS

SEE ALSO

rtcSetGeometryBuffer, *rtcSetSharedGeometryBuffer*, *rtcSetNewGeometryBuffer*

rtcGetGeometryBufferData

NAME

```
rtcGetGeometryBufferData - gets pointer to
  the first buffer view element
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void* rtcGetGeometryBufferData(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot
);
```

DESCRIPTION

The `rtcGetGeometryBufferData` function returns a pointer to the first element of the buffer view attached to the specified buffer type and slot (`type` and `slot` argument) of the geometry (`geometry` argument).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryBuffer, *rtcSetSharedGeometryBuffer*, *rtcSetNewGeometryBuffer*

rtcUpdateGeometryBuffer

NAME

```
rtcUpdateGeometryBuffer - marks a buffer view bound to the geometry
  as modified
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcUpdateGeometryBuffer(
  RTCGeometry geometry,
  enum RTCBufferType type,
  unsigned int slot
);
```

DESCRIPTION

The `rtcUpdateGeometryBuffer` function marks the buffer view bound to the specified buffer type and slot (`type` and `slot` argument) of a geometry (`geometry` argument) as modified.

If a data buffer is changed by the application, the `rtcUpdateGeometryBuffer` call must be invoked for that buffer. Each buffer view assigned to a buffer slot is initially marked as modified, thus this function needs to be called only when doing buffer modifications after the first `rtcCommitScene`.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewGeometry, *rtcCommitScene*

rtcSetGeometryIntersectFilterFunction

NAME

```
rtcSetGeometryIntersectFilterFunction - sets the intersection filter
  for the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCFilterFunctionNArguments
{
  int* valid;
  void* geometryUserPtr;
  const struct RTCIntersectContext* context;
  struct RTCRayN* ray;
  struct RTCHitN* hit;
  unsigned int N;
};

typedef void (*RTCFilterFunctionN)(
  const struct RTCFilterFunctionNArguments* args
);

void rtcSetGeometryIntersectFilterFunction(
  RTCGeometry geometry,
  RTCFilterFunctionN filter
);
```

DESCRIPTION

The `rtcSetGeometryIntersectFilterFunction` function registers an intersection filter callback function (`filter` argument) for the specified geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered intersection filter function is invoked for every hit encountered during the `rtcIntersect`-type ray queries and can accept or reject that hit. The feature can be used to define a silhouette for a primitive and reject hits that are outside the silhouette. E.g. a tree leaf could be modeled with an alpha texture that decides whether hit points lie inside or outside the leaf.

If the `RTC_BUILD_QUALITY_HIGH` mode is set, the filter functions may be called multiple times for the same primitive hit. Further, rays hitting exactly the edge might also report two hits for the same surface. For certain use cases, the application may have to work around this limitation by collecting already reported hits (`geomID`/`primID` pairs) and ignoring duplicates.

The filter function callback of type `RTCFilterFunctionN` gets passed a number of arguments through the `RTCFilterFunctionNArguments` structure. The `valid` parameter of that structure points to an integer valid mask (0 means invalid and -1 means valid). The `geometryUserPtr` member is a user pointer optionally set per geometry through the `rtcSetGeometryUserData` function. The `context` member points to the intersection context passed to the ray query function. The `ray` parameter points to `N` rays in SOA layout. The `hit` parameter points to `N` hits in SOA layout to test. The `N` parameter is the number of rays and hits in `ray` and `hit`. The hit distance

is provided as the `tfar` value of the ray. If the hit geometry is instanced, the `instID` member of the ray is valid, and the ray and the potential hit are in object space.

The filter callback function has the task to check for each valid ray whether it wants to accept or reject the corresponding hit. To reject a hit, the filter callback function just has to write `0` to the integer valid mask of the corresponding ray. To accept the hit, it just has to leave the valid mask set to `-1`. The filter function is further allowed to change the hit and decrease the `tfar` value of the ray but it should not modify other ray data nor any inactive components of the ray or hit.

When performing ray queries using `rtcIntersect1`, it is guaranteed that the packet size is 1 when the callback is invoked. When performing ray queries using the `rtcIntersect4/8/16` functions, it is not generally guaranteed that the ray packet size (and order of rays inside the packet) passed to the callback matches the initial ray packet. However, under some circumstances these properties are guaranteed, and whether this is the case can be queried using `rtcGetDeviceProperty`. When performing ray queries using the stream API such as `rtcIntersect1M`, `rtcIntersect1Mp`, `rtcIntersectNM`, or `rtcIntersectNp` the order of rays and ray packet size of the callback function might change to either 1, 4, 8, or 16.

For many usage scenarios, repacking and re-ordering of rays does not cause difficulties in implementing the callback function. However, algorithms that need to extend the ray with additional data must use the `rayID` component of the ray to identify the original ray to access the per-ray data.

The implementation of the filter function can choose to implement a single code path that uses the ray access helper functions `RTCRay_XXX` and hit access helper functions `RTCHit_XXX` to access ray and hit data. Alternatively the code can branch to optimized implementations for specific sizes of `N` and cast the `ray` and `hit` inputs to the proper packet types.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryOccludedFilterFunction

rtcSetGeometryOccludedFilterFunction

NAME

```
rtcSetGeometryOccludedFilterFunction - sets the occlusion filter
  for the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryOccludedFilterFunction(
  RTCGeometry geometry,
  RTCFilterFunctionN filter
);
```

DESCRIPTION

The `rtcSetGeometryOccludedFilterFunction` function registers an occlusion filter callback function (`filter` argument) for the specified geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered intersection filter function is invoked for every hit encountered during the `rtcOccluded`-type ray queries and can accept or reject that hit. The feature can be used to define a silhouette for a primitive and reject hits that are outside the silhouette. E.g. a tree leaf could be modeled with an alpha texture that decides whether hit points lie inside or outside the leaf.

Please see the description of the `rtcSetGeometryIntersectFilterFunction` for a description of the filter callback function.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryIntersectFilterFunction

rtcFilterIntersection

NAME

```
rtcFilterIntersection - invokes the intersection filter function
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcFilterIntersection(
  const struct RTCIntersectFunctionNArguments* args,
  const struct RTCFilterFunctionNArguments* filterArgs
);
```

DESCRIPTION

The `rtcFilterIntersection` function can be called inside an `RTCIntersectFunctionN` callback function to invoke the intersection filter registered to the geometry and stored inside the context. For this an `RTCFilterFunctionNArguments` structure must be created (see `rtcSetGeometryIntersectFilterFunction`) which basically consists of a valid mask, a hit packet to filter, the corresponding ray packet, and the packet size. After the invocation of `rtcFilterIntersection`, only rays that are still valid (valid mask set to -1) should update a hit.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcFilterOcclusion, *rtcSetGeometryIntersectFunction*

rtcFilterOcclusion

NAME

```
rtcFilterOcclusion - invokes the occlusion filter function
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcFilterOcclusion(
  const struct RTCOccludedFunctionNArguments* args,
  const struct RTCFilterFunctionNArguments* filterArgs
);
```

DESCRIPTION

The `rtcFilterOcclusion` function can be called inside an `RTCOccludedFunctionN` callback function to invoke the occlusion filter registered to the geometry and stored inside the context. For this an `RTCFilterFunctionNArguments` structure must be created (see `rtcSetGeometryIntersectFilterFunction`) which basically consists of a valid mask, a hit packet to filter, the corresponding ray packet, and the packet size. After the invocation of `rtcFilterOcclusion` only rays that are still valid (valid mask set to -1) should signal an occlusion.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcFilterIntersection, *rtcSetGeometryOccludedFunction*

rtcSetGeometryUserData

NAME

```
rtcSetGeometryUserData - sets the user-defined data pointer of the
  geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryUserData(RTCGeometry geometry, void* userPtr);
```

DESCRIPTION

The `rtcSetGeometryUserData` function sets the user-defined data pointer (`userPtr` argument) for a geometry (`geometry` argument). This user data pointer is intended to be pointing to the application's representation of the geometry, and is passed to various callback functions. The application can use this pointer inside the callback functions to access its geometry representation.

The `rtcGetGeometryUserData` function can be used to query an already set user data pointer of a geometry.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryUserData

rtcGetGeometryUserData

NAME

```
rtcGetGeometryUserData - returns the user data pointer
  of the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void* rtcGetGeometryUserData(RTCGeometry geometry);
```

DESCRIPTION

The `rtcGetGeometryUserData` function queries the user data pointer previously set with `rtcSetGeometryUserData`. When `rtcSetGeometryUserData` was not called yet, `NULL` is returned.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryUserData

rtcSetGeometryUserPrimitiveCount

NAME

```
rtcSetGeometryUserPrimitiveCount - sets the number of primitives
  of a user-defined geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryUserPrimitiveCount(
  RTCGeometry geometry,
  unsigned int userPrimitiveCount
);
```

DESCRIPTION

The `rtcSetGeometryUserPrimitiveCount` function sets the number of user-defined primitives (`userPrimitiveCount` parameter) of the specified user-defined geometry (`geometry` parameter).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_USER

rtcSetGeometryBoundsFunction

NAME

```
rtcSetGeometryBoundsFunction - sets a callback to query the
  bounding box of user-defined primitives
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCBoundsFunctionArguments
{
  void* geometryUserPtr;
  unsigned int primID;
  unsigned int timeStep;
  struct RTCBounds* bounds_o;
};

typedef void (*RTCBoundsFunction)(
  const struct RTCBoundsFunctionArguments* args
);

void rtcSetGeometryBoundsFunction(
  RTCGeometry geometry,
  RTCBoundsFunction bounds,
  void* userPtr
);
```

DESCRIPTION

The `rtcSetGeometryBoundsFunction` function registers a bounding box callback function (`bounds` argument) with payload (`userPtr` argument) for the specified user geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered bounding box callback function is invoked to calculate axis-aligned bounding boxes of the primitives of the user-defined geometry during spatial acceleration structure construction. The bounding box callback of `RTCBoundsFunction` type is invoked with a pointer to a structure of type `RTCBoundsFunctionArguments` which contains various arguments, such as: the user data of the geometry (`geometryUserPtr` member), the ID of the primitive to calculate the bounds for (`primID` member), the time step at which to calculate the bounds (`timeStep` member), and a memory location to write the calculated bound to (`bounds_o` member).

In a typical usage scenario one would store a pointer to the internal representation of the user geometry object using `rtcSetGeometryUserData`. The callback function can then read that pointer from the `geometryUserPtr` field and calculate the proper bounding box for the requested primitive and time, and store that bounding box to the destination structure (`bounds_o` member).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_USER

rtcSetGeometryIntersectFunction

NAME

```
rtcSetGeometryIntersectFunction - sets the callback function to
  intersect a user geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCIntersectFunctionNArguments
{
  int* valid;
  void* geometryUserPtr;
  unsigned int primID;
  struct RTCIntersectContext* context;
  struct RTCRayHitN* rayhit;
  unsigned int N;
  unsigned int geomID;
};

typedef void (*RTCIntersectFunctionN)(
  const struct RTCIntersectFunctionNArguments* args
);

void rtcSetGeometryIntersectFunction(
  RTCGeometry geometry,
  RTCIntersectFunctionN intersect
);
```

DESCRIPTION

The `rtcSetGeometryIntersectFunction` function registers a ray/primitive intersection callback function (`intersect` argument) for the specified user geometry (`geometry` argument).

Only a single callback function can be registered per geometry and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered callback function is invoked by `rtcIntersect`-type ray queries to calculate the intersection of a ray packet of variable size with one user-defined primitive. The callback function of type `RTCIntersectFunctionN` gets passed a number of arguments through the `RTCIntersectFunctionNArguments` structure. The value `N` specifies the ray packet size, `valid` points to an array of integers that specify whether the corresponding ray is valid (-1) or invalid (0), the `geometryUserPtr` member points to the geometry user data previously set through `rtcSetGeometryUserData`, the `context` member points to the intersection context passed to the ray query, the `rayhit` member points to a ray and hit packet of variable size `N`, and the `geomID` and `primID` member identifies the geometry ID and primitive ID of the primitive to intersect.

The `ray` component of the `rayhit` structure contains valid data, in particular the `tfar` value is the current closest hit distance found. All data inside the `hit` component of the `rayhit` structure are undefined and should not be read by the function.

The task of the callback function is to intersect each active ray from the ray packet with the specified user primitive. If the user-defined primitive is missed by a ray of the ray packet, the function should return without modifying the

ray or hit. If an intersection of the user-defined primitive with the ray was found in the valid range (from `tnear` to `tfar`), it should update the hit distance of the ray (`tfar` member) and the hit (`u`, `v`, `Ng`, `instID`, `geomID`, `primID` members). In particular, the currently intersected instance is stored in the `instID` field of the intersection context, which must be deep copied into the `instID` member of the hit.

As a primitive might have multiple intersections with a ray, the intersection filter function needs to be invoked by the user geometry intersection callback for each encountered intersection, if filtering of intersections is desired. This can be achieved through the `rtcFilterIntersection` call.

Within the user geometry intersect function, it is safe to trace new rays and create new scenes and geometries.

When performing ray queries using `rtcIntersect1`, it is guaranteed that the packet size is 1 when the callback is invoked. When performing ray queries using the `rtcIntersect4/8/16` functions, it is not generally guaranteed that the ray packet size (and order of rays inside the packet) passed to the callback matches the initial ray packet. However, under some circumstances these properties are guaranteed, and whether this is the case can be queried using `rtcGetDeviceProperty`. When performing ray queries using the stream API such as `rtcIntersect1M`, `rtcIntersect1Mp`, `rtcIntersectNM`, or `rtcIntersectNp` the order of rays and ray packet size of the callback function might change to either 1, 4, 8, or 16.

For many usage scenarios, repacking and re-ordering of rays does not cause difficulties in implementing the callback function. However, algorithms that need to extend the ray with additional data must use the `rayID` component of the ray to identify the original ray to access the per-ray data.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryOccludedFunction, *rtcSetGeometryUserData*, *rtcFilterIntersection*

rtcSetGeometryOccludedFunction

NAME

```
rtcSetGeometryOccludedFunction - sets the callback function to
  test a user geometry for occlusion
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCOccludedFunctionNArguments
{
  int* valid;
  void* geometryUserPtr;
  unsigned int primID;
  struct RTCIntersectContext* context;
  struct RTCRayN* ray;
  unsigned int N;
  unsigned int geomID;
};

typedef void (*RTCOccludedFunctionN)(
  const struct RTCOccludedFunctionNArguments* args
);

void rtcSetGeometryOccludedFunction(
  RTCGeometry geometry,
  RTCOccludedFunctionN filter
);
```

DESCRIPTION

The `rtcSetGeometryOccludedFunction` function registers a ray/primitive occlusion callback function (`filter` argument) for the specified user geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered callback function is invoked by `rtcOccluded`-type ray queries to test whether the rays of a packet of variable size are occluded by a user-defined primitive. The callback function of type `RTCOccludedFunctionN` gets passed a number of arguments through the `RTCOccludedFunctionNArguments` structure. The value `N` specifies the ray packet size, `valid` points to an array of integers which specify whether the corresponding ray is valid (-1) or invalid (0), the `geometryUserPtr` member points to the geometry user data previously set through `rtcSetGeometryUserData`, the `context` member points to the intersection context passed to the ray query, the `ray` member points to a ray packet of variable size `N`, and the `geomID` and `primID` member identifies the geometry ID and primitive ID of the primitive to intersect.

The task of the callback function is to intersect each active ray from the ray packet with the specified user primitive. If the user-defined primitive is missed by a ray of the ray packet, the function should return without modifying the ray. If an intersection of the user-defined primitive with the ray was found in the valid range (from `tnear` to `tfar`), it should set the `tfar` member of the ray to `-inf`.

As a primitive might have multiple intersections with a ray, the occlusion filter function needs to be invoked by the user geometry occlusion callback for each encountered intersection, if filtering of intersections is desired. This can be achieved through the `rtcFilterOcclusion` call.

Within the user geometry occlusion function, it is safe to trace new rays and create new scenes and geometries.

When performing ray queries using `rtcOccluded1`, it is guaranteed that the packet size is 1 when the callback is invoked. When performing ray queries using the `rtcOccluded4/8/16` functions, it is not generally guaranteed that the ray packet size (and order of rays inside the packet) passed to the callback matches the initial ray packet. However, under some circumstances these properties are guaranteed, and whether this is the case can be queried using `rtcGetDeviceProperty`. When performing ray queries using the stream API such as `rtcOccluded1M`, `rtcOccluded1Mp`, `rtcOccludedNM`, or `rtcOccludedNp` the order of rays and ray packet size of the callback function might change to either 1, 4, 8, or 16.

For many usage scenarios, repacking and re-ordering of rays does not cause difficulties in implementing the callback function. However, algorithms that need to extend the ray with additional data must use the `rayID` component of the ray to identify the original ray to access the per-ray data.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometryIntersectFunction, *rtcSetGeometryUserData*, *rtcFilterOcclusion*

rtcSetGeometryPointQueryFunction

NAME

```
rtcSetGeometryPointQueryFunction - sets the point query callback function
  for a geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCPointQueryFunctionArguments
{
  // the (world space) query object that was passed as an argument of rtcPointQuery.
  struct RTCPointQuery* query;

  // used for user input/output data. Will not be read or modified internally.
  void* userPtr;

  // primitive and geometry ID of primitive
  unsigned int  primID;
  unsigned int  geomID;

  // the context with transformation and instance ID stack
  struct RTCPointQueryContext* context;

  // scaling factor indicating whether the current instance transformation
  // is a similarity transformation.
  float similarityScale;
};

typedef bool (*RTCPointQueryFunction)(
  struct RTCPointQueryFunctionArguments* args
);

void rtcSetGeometryPointQueryFunction(
  RTCGeometry geometry,
  RTCPointQueryFunction queryFunc
);
```

DESCRIPTION

The `rtcSetGeometryPointQueryFunction` function registers a point query callback function (`queryFunc` argument) for the specified geometry (`geometry` argument).

Only a single callback function can be registered per geometry and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered callback function is invoked by *rtcPointQuery* for every primitive of the geometry that intersects the corresponding point query domain. The callback function of type `RTCPointQueryFunction` gets passed a number of arguments through the `RTCPointQueryFunctionArguments` structure. The `query` object is the original point query object passed into *rtcPointQuery*, `usrPtr` is an arbitrary pointer to pass input into and store results of the callback function. The `primID`, `geomID` and `context` (see *rtcInitPointQueryContext* for details) can be used to identify the geometry data of the primitive.

A `RTCPointQueryFunction` can also be passed directly as an argument to *rtcPointQuery*. In this case the callback is invoked for all primitives in the scene that intersect the query domain. If a callback function is passed as an argument to *rtcPointQuery* and (a potentially different) callback function is set for a geometry with *rtcSetGeometryPointQueryFunction* both callback functions are invoked and the callback function passed to *rtcPointQuery* will be called before the geometry specific callback function.

If instancing is used, the parameter `simliarityScale` indicates whether the current instance transform (top element of the stack in `context`) is a similarity transformation or not. Similarity transformations are composed of translation, rotation and uniform scaling and if a matrix M defines a similarity transformation, there is a scaling factor D such that for all x,y: dist(Mx, My) = D * dist(x, y). In this case the parameter `scalingFactor` is this scaling factor D and otherwise it is 0. A valid similarity scale (`similarityScale` > 0) allows to compute distance information in instance space and scale the distances into world space (for example, to update the query radius, see below) by dividing the instance space distance with the similarity scale. If the current instance transform is not a similarity transform (`similarityScale` is 0), the distance computation has to be performed in world space to ensure correctness. In this case the instance to world transformations given with the `context` should be used to transform the primitive data into world space. Otherwise, the query location can be transformed into instance space which can be more efficient. If there is no instance transform, the similarity scale is 1.

The callback function will potentially be called for primitives outside the query domain for two resons: First, the callback is invoked for all primitives inside a BVH leaf node since no geometry data of primitives is determined internally and therefore individual primitives are not culled (only their (aggregated) bounding boxes). Second, in case non similarity transformations are used, the resulting ellipsoidal query domain (in instance space) is approximated by its axis aligned bounding box internally and therefore inner nodes that do not intersect the original domain might intersect the approximative bounding box which results in unneccessary callbacks. In any case, the callbacks are conservative, i.e. if a primitive is inside the query domain a callback will be invoked but the reverse is not neccessarily true.

For efficiency, the radius of the `query` object can be decreased (in world space) inside the callback function to improve culling of geometry during BVH traversal. If the query radius was updated, the callback function should return `true` to issue an update of internal traversal information. Increasing the radius or modifying the time or position of the query results in undefined behaviour.

Within the callback function, it is safe to call *rtcPointQuery* again, for example when implementing instancing manually. In this case the instance transformation should be pushed onto the stack in `context`. Embree will internally compute the point query information in instance space using the top element of the stack in `context` when *rtcPointQuery* is called.

For a reference implementation of a closest point traversal of triangle meshes using instancing and user defined instancing see the tutorial [ClosestPoint].

SEE ALSO

rtcPointQuery, *rtcInitPointQueryContext*

rtcSetGeometryInstancedScene

NAME

```
rtcSetGeometryInstancedScene - sets the instanced scene of
  an instance geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryInstancedScene(
  RTCGeometry geometry,
  RTCScene scene
);
```

DESCRIPTION

The `rtcSetGeometryInstancedScene` function sets the instanced scene (`scene` argument) of the specified instance geometry (`geometry` argument).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_INSTANCE, *rtcSetGeometryTransform*

rtcSetGeometryTransform

NAME

```
rtcSetGeometryTransform - sets the transformation for a particular
  time step of an instance geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryTransform(
  RTCGeometry geometry,
  unsigned int timeStep,
  enum RTCFormat format,
  const float* xfm
);
```

DESCRIPTION

The `rtcSetGeometryTransform` function sets the local-to-world affine transformation (`xfm` parameter) of an instance geometry (`geometry` parameter) for a particular time step (`timeStep` parameter). The transformation is specified as a 3×4 matrix (3×3 linear transformation plus translation), for which the following formats (`format` parameter) are supported:

- `RTC_FORMAT_FLOAT3X4_ROW_MAJOR`: The 3×4 float matrix is laid out in row-major form.
- `RTC_FORMAT_FLOAT3X4_COLUMN_MAJOR`: The 3×4 float matrix is laid out in column-major form.
- `RTC_FORMAT_FLOAT4X4_COLUMN_MAJOR`: The 3×4 float matrix is laid out in column-major form as a 4×4 homogeneous matrix with the last row being equal to (0, 0, 0, 1).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_INSTANCE

rtcSetGeometryTransformQuaternion

NAME

```
rtcSetGeometryTransformQuaternion - sets the transformation for a particular
  time step of an instance geometry as a decomposition of the
  transformation matrix using quaternions to represent the rotation.
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryTransformQuaternion(
  RTCGeometry geometry,
  unsigned int timeStep,
  const struct RTCQuaternionDecomposition* qd
);
```

DESCRIPTION

The `rtcSetGeometryTransformQuaternion` function sets the local-to-world affine transformation (`qd` parameter) of an instance geometry (`geometry` parameter) for a particular time step (`timeStep` parameter). The transformation is specified as a *RTCQuaternionDecomposition*, which is a decomposition of an affine transformation that represents the rotational component of an affine transformation as a quaternion. This allows interpolating rotational transformations exactly using spherical linear interpolation (such as a turning wheel).

For more information about the decomposition see *RTCQuaternionDecomposition*. The quaternion given in the `RTCQuaternionDecomposition` struct will be normalized internally.

For correct results, the transformation matrices for all time steps must be set either using `rtcSetGeometryTransform` or `rtcSetGeometryTransformQuaternion`. Mixing both representations is not allowed. Spherical linear interpolation will be used, iff the transformation matizes are set with `rtcSetGeometryTransformQuaternion`.

For an example of this feature see the tutorial Quaternion Motion Blur.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcInitQuaternionDecomposition, *rtcSetGeometryTransform*

rtcGetGeometryTransform

NAME

```
rtcGetGeometryTransform - returns the interpolated instance
  transformation for the specified time
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcGetGeometryTransform(
  RTCGeometry geometry,
  float time,
  enum RTCFormat format,
  void* xfm
);
```

DESCRIPTION

The `rtcGetGeometryTransform` function returns the interpolated local to world transformation (`xfm` parameter) of an instance geometry (`geometry` parameter) for a particular time (`time` parameter in range $[0, 1]$) in the specified format (`format` parameter).

Possible formats for the returned matrix are:

- `RTC_FORMAT_FLOAT3X4_ROW_MAJOR`: The 3×4 float matrix is laid out in row-major form.
- `RTC_FORMAT_FLOAT3X4_COLUMN_MAJOR`: The 3×4 float matrix is laid out in column-major form.
- `RTC_FORMAT_FLOAT4X4_COLUMN_MAJOR`: The 3×4 float matrix is laid out in column-major form as a 4×4 homogeneous matrix with last row equal to (0, 0, 0, 1).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_INSTANCE, *rtcSetGeometryTransform*

rtcSetGeometryTessellationRate

NAME

```
rtcSetGeometryTessellationRate - sets the tessellation rate of the
  geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryTessellationRate(
  RTCGeometry geometry,
  float tessellationRate
);
```

DESCRIPTION

The `rtcSetGeometryTessellationRate` function sets the tessellation rate (`tessellationRate` argument) for the specified geometry (`geometry` argument). The tessellation rate can only be set for flat curves and subdivision geometries. For curves, the tessellation rate specifies the number of ray-facing quads per curve segment. For subdivision surfaces, the tessellation rate specifies the number of quads along each edge.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_CURVE, *RTC_GEOMETRY_TYPE_SUBDIVISION*

rtcSetGeometryTopologyCount

NAME

```
rtcSetGeometryTopologyCount - sets the number of topologies of
  a subdivision geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryTopologyCount(
  RTCGeometry geometry,
  unsigned int topologyCount
);
```

DESCRIPTION

The `rtcSetGeometryTopologyCount` function sets the number of topologies (`topologyCount` parameter) for the specified subdivision geometry (`geometry` parameter). The number of topologies of a subdivision geometry must be greater or equal to 1.

To use multiple topologies, first the number of topologies must be specified, then the individual topologies can be configured using `rtcSetGeometrySubdivisionMode` and by setting an index buffer (`RTC_BUFFER_TYPE_INDEX`) using the topology ID as the buffer slot.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_SUBDIVISION, *rtcSetGeometrySubdivisionMode*

rtcSetGeometrySubdivisionMode

NAME

```
rtcSetGeometrySubdivisionMode - sets the subdivision mode
  of a subdivision geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometrySubdivisionMode(
  RTCGeometry geometry,
  unsigned int topologyID,
  enum RTCSubdivisionMode mode
);
```

DESCRIPTION

The `rtcSetGeometrySubdivisionMode` function sets the subdivision mode (`mode` parameter) for the topology (`topologyID` parameter) of the specified subdivision geometry (`geometry` parameter).

The subdivision modes can be used to force linear interpolation for certain parts of the subdivision mesh:

- `RTC_SUBDIVISION_MODE_NO_BOUNDARY`: Boundary patches are ignored. This way each rendered patch has a full set of control vertices.
- `RTC_SUBDIVISION_MODE_SMOOTH_BOUNDARY`: The sequence of boundary control points are used to generate a smooth B-spline boundary curve (default mode).
- `RTC_SUBDIVISION_MODE_PIN_CORNERS`: Corner vertices are pinned to their location during subdivision.
- `RTC_SUBDIVISION_MODE_PIN_BOUNDARY`: All vertices at the border are pinned to their location during subdivision. This way the boundary is interpolated linearly. This mode is typically used for texturing to also map texels at the border of the texture to the mesh.
- `RTC_SUBDIVISION_MODE_PIN_ALL`: All vertices at the border are pinned to their location during subdivision. This way all patches are linearly interpolated.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

RTC_GEOMETRY_TYPE_SUBDIVISION

rtcSetGeometryVertexAttributeTopology

NAME

```
rtcSetGeometryVertexAttributeTopology - binds a vertex
  attribute to a topology of the geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcSetGeometryVertexAttributeTopology(
  RTCGeometry geometry,
  unsigned int vertexAttributeID,
  unsigned int topologyID
);
```

DESCRIPTION

The `rtcSetGeometryVertexAttributeTopology` function binds a vertex attribute buffer slot (`vertexAttributeID` argument) to a topology (`topologyID` argument) for the specified subdivision geometry (`geometry` argument). Standard vertex buffers are always bound to the default topology (topology 0) and cannot be bound differently. A vertex attribute buffer always uses the topology it is bound to when used in the `rtcInterpolate` and `rtcInterpolateN` calls.

A topology with ID `i` consists of a subdivision mode set through `rtcSetGeometrySubdivisionMode` and the index buffer bound to the index buffer slot `i`. This index buffer can assign indices for each face of the subdivision geometry that are different to the indices of the default topology. These new indices can for example be used to introduce additional borders into the subdivision mesh to map multiple textures onto one subdivision geometry.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcSetGeometrySubdivisionMode, *rtcInterpolate*, *rtcInterpolateN*

rtcSetGeometryDisplacementFunction

NAME

```
rtcSetGeometryDisplacementFunction - sets the displacement function
  for a subdivision geometry
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCDisplacementFunctionNArguments
{
  void* geometryUserPtr;
  RTCGeometry geometry;
  unsigned int primID;
  unsigned int timeStep;
  const float* u;
  const float* v;
  const float* Ng_x;
  const float* Ng_y;
  const float* Ng_z;
  float* P_x;
  float* P_y;
  float* P_z;
  unsigned int N;
};

typedef void (*RTCDisplacementFunctionN)(
   const struct RTCDisplacementFunctionNArguments* args
);

void rtcSetGeometryDisplacementFunction(
  RTCGeometry geometry,
  RTCDisplacementFunctionN displacement
);
```

DESCRIPTION

The `rtcSetGeometryDisplacementFunction` function registers a displacement callback function (`displacement` argument) for the specified subdivision geometry (`geometry` argument).

Only a single callback function can be registered per geometry, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function.

The registered displacement callback function is invoked to displace points on the subdivision geometry during spatial acceleration structure construction, during the `rtcCommitScene` call.

The callback function of type `RTCDisplacementFunctionN` is invoked with a number of arguments stored inside the `RTCDisplacementFunctionNArguments` structure. The provided user data pointer of the geometry (`geometryUserPtr` member) can be used to point to the application's representation of the subdivision mesh. A number `N` of points to displace are specified in a structure of array layout. For each point to displace, the local patch UV coordinates (`u` and `v` arrays), the normalized geometry normal (`Ng_x`, `Ng_y`, and `Ng_z` arrays), and the position

(P_x, P_y, and P_z arrays) are provided. The task of the displacement function is to use this information and change the position data.

The geometry handle (geometry member) and primitive ID (primID member) of the patch to displace are additionally provided as well as the time step timeStep, which can be important if the displacement is time-dependent and motion blur is used.

All passed arrays must be aligned to 64 bytes and properly padded to make wide vector processing inside the displacement function easily possible.

Also see tutorial Displacement Geometry for an example of how to use the displacement mapping functions.

EXIT STATUS

On failure an error code is set that can be queried using rtcGetDeviceError.

SEE ALSO

RTC_GEOMETRY_TYPE_SUBDIVISION

rtcGetGeometryFirstHalfEdge

NAME

```
rtcGetGeometryFirstHalfEdge - returns the first half edge of a face
```

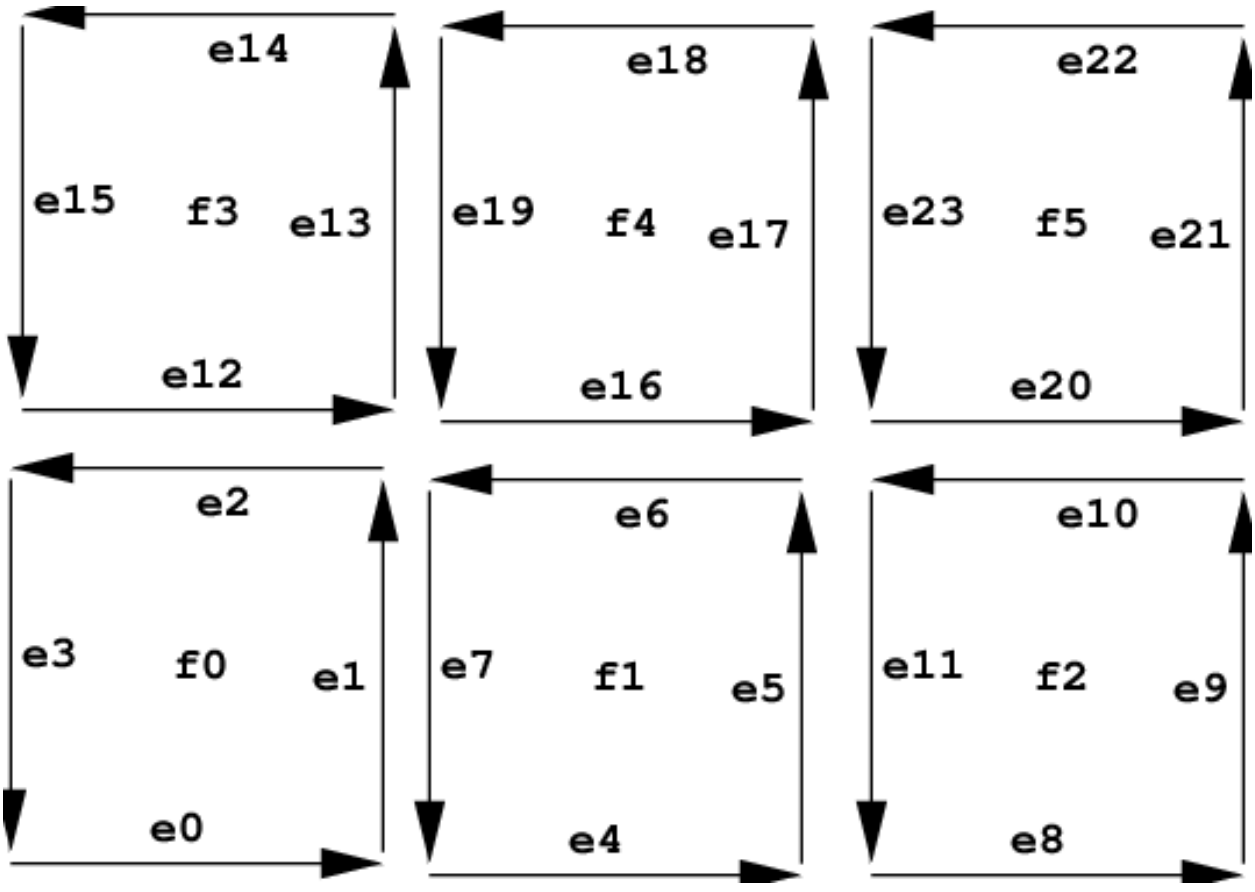
SYNOPSIS

```
#include <embree3/rtcore.h>

unsigned int rtcGetGeometryFirstHalfEdge(
  RTCGeometry geometry,
  unsigned int faceID
);
```

DESCRIPTION

The `rtcGetGeometryFirstHalfEdge` function returns the ID of the first half edge belonging to the specified face (`faceID` argument). For instance in the following example the first half edge of face `f1` is `e4`.



This function can only be used for subdivision geometries. As all topologies of a subdivision geometry share the same face buffer the function does not depend on the topology ID.

Here f0 to f7 are 8 quadrilateral faces with 4 vertices each. The edges e0 to e23 of these faces are shown with their orientation. For each face the ID of the edges corresponds to the slots the face occupies in the index array of the geometry. E.g. as the indices of face f1 start at location 4 of the index array, the first edge is edge e4, the next edge e5, etc.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, *rtcGetGeometryFace*, *rtcGetGeometryOppositeHalfEdge*, *rtcGetGeometryNextHalfEdge*, *rtcGetGeometryPreviousHalfEdge*

rtcGetGeometryFace

NAME

```
rtcGetGeometryFace - returns the face of some half edge
```

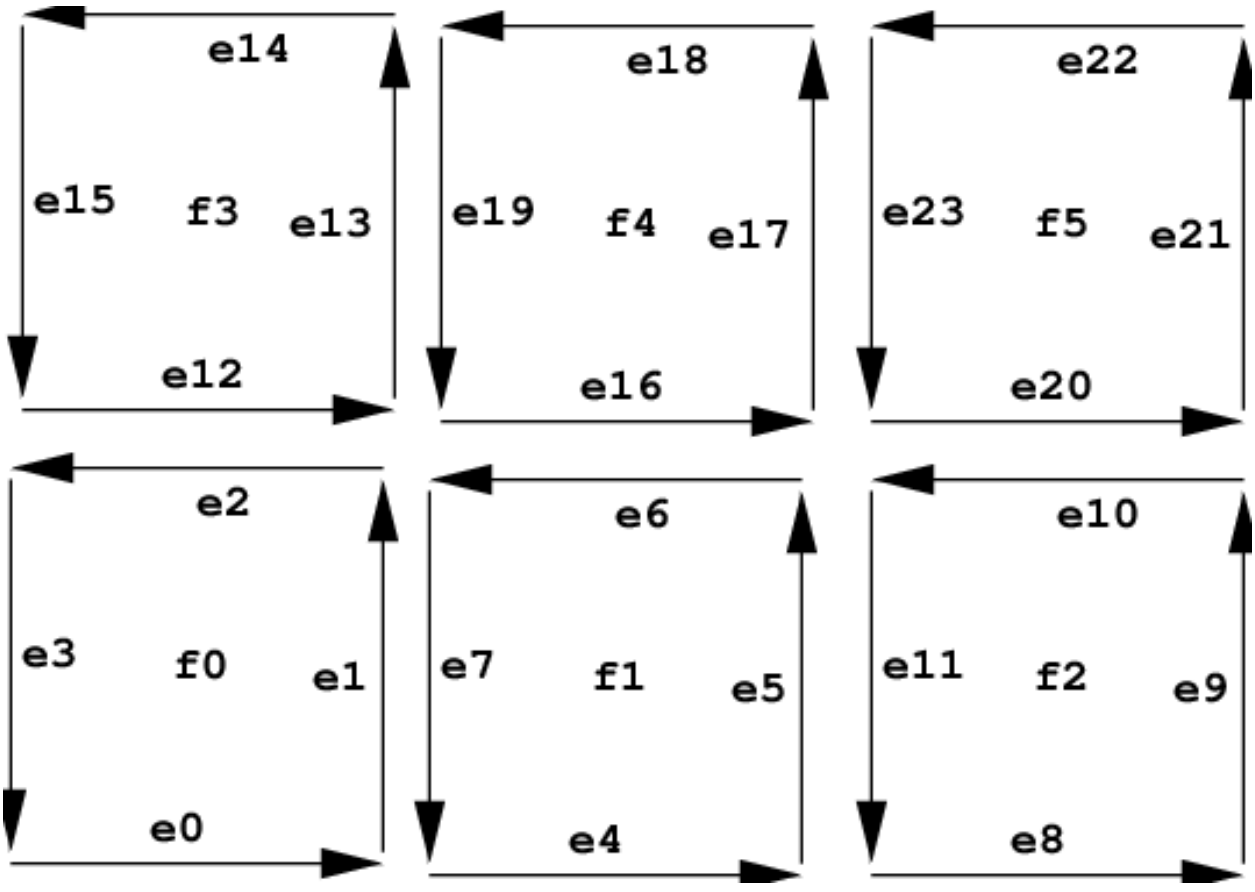
SYNOPSIS

```
#include <embree3/rtcore.h>

unsigned int rtcGetGeometryFace(
  RTCGeometry geometry,
  unsigned int edgeID
);
```

DESCRIPTION

The `rtcGetGeometryFace` function returns the ID of the face the specified half edge (`edgeID` argument) belongs to. For instance in the following example the face `f1` is returned for edges `e4`, `e5`, `e6`, and `e7`.



This function can only be used for subdivision geometries. As all topologies of a subdivision geometry share the same face buffer the function does not depend on the topology ID.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, *rtcGetGeometryFace*, *rtcGetGeometryOppositeHalfEdge*, *rtcGetGeometryNextHalfEdge*, *rtcGetGeometryPreviousHalfEdge*

rtcGetGeometryNextHalfEdge

NAME

```
rtcGetGeometryNextHalfEdge - returns the next half edge
```

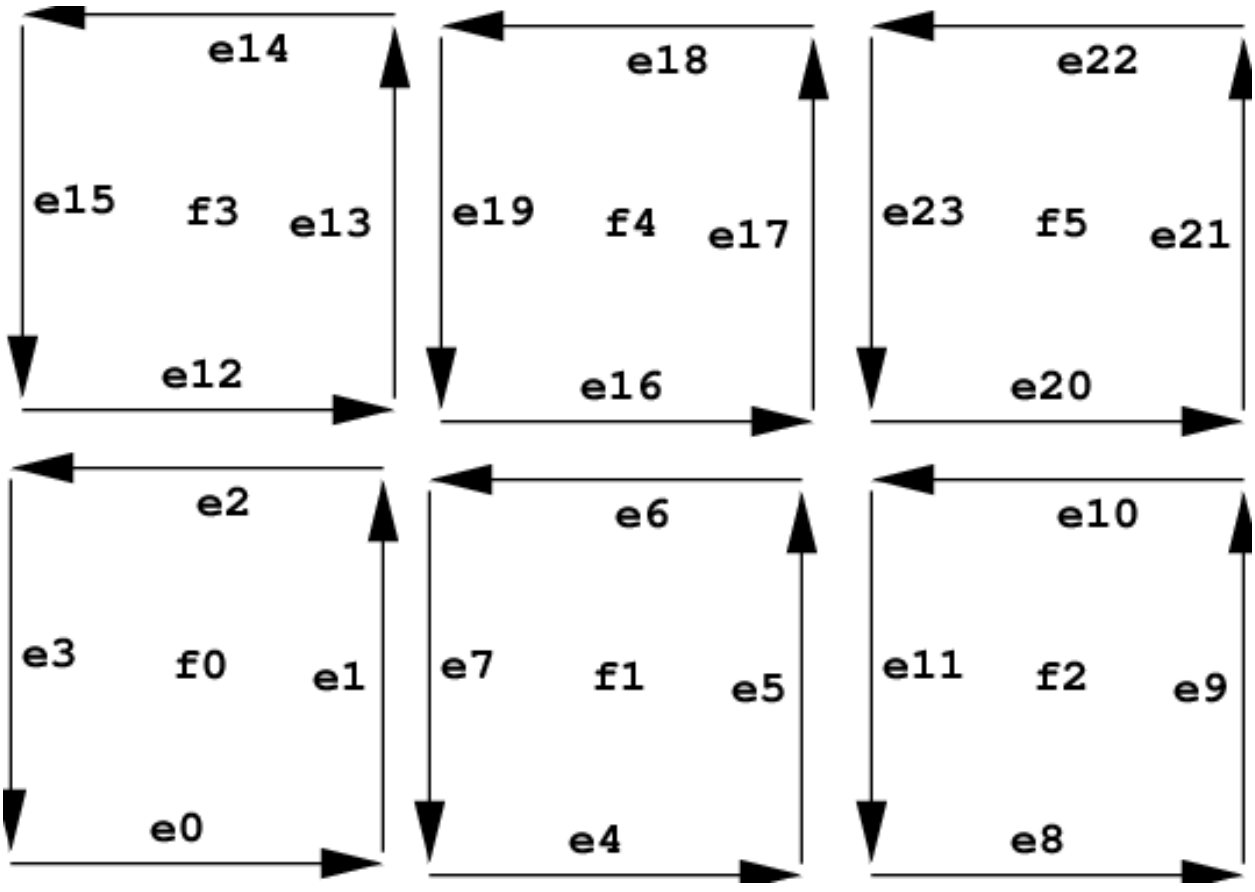
SYNOPSIS

```
#include <embree3/rtcore.h>

unsigned int rtcGetGeometryNextHalfEdge(
  RTCGeometry geometry,
  unsigned int edgeID
);
```

DESCRIPTION

The `rtcGetGeometryNextHalfEdge` function returns the ID of the next half edge of the specified half edge (`edgeID` argument). For instance in the following example the next half edge of `e10` is `e11`.



This function can only be used for subdivision geometries. As all topologies of a subdivision geometry share the same face buffer the function does not depend on the topology ID.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, *rtcGetGeometryFace*, *rtcGetGeometryOppositeHalfEdge*, *rtcGetGeometryNextHalfEdge*, *rtcGetGeometryPreviousHalfEdge*

rtcGetGeometryPreviousHalfEdge

NAME

```
rtcGetGeometryPreviousHalfEdge - returns the previous half edge
```

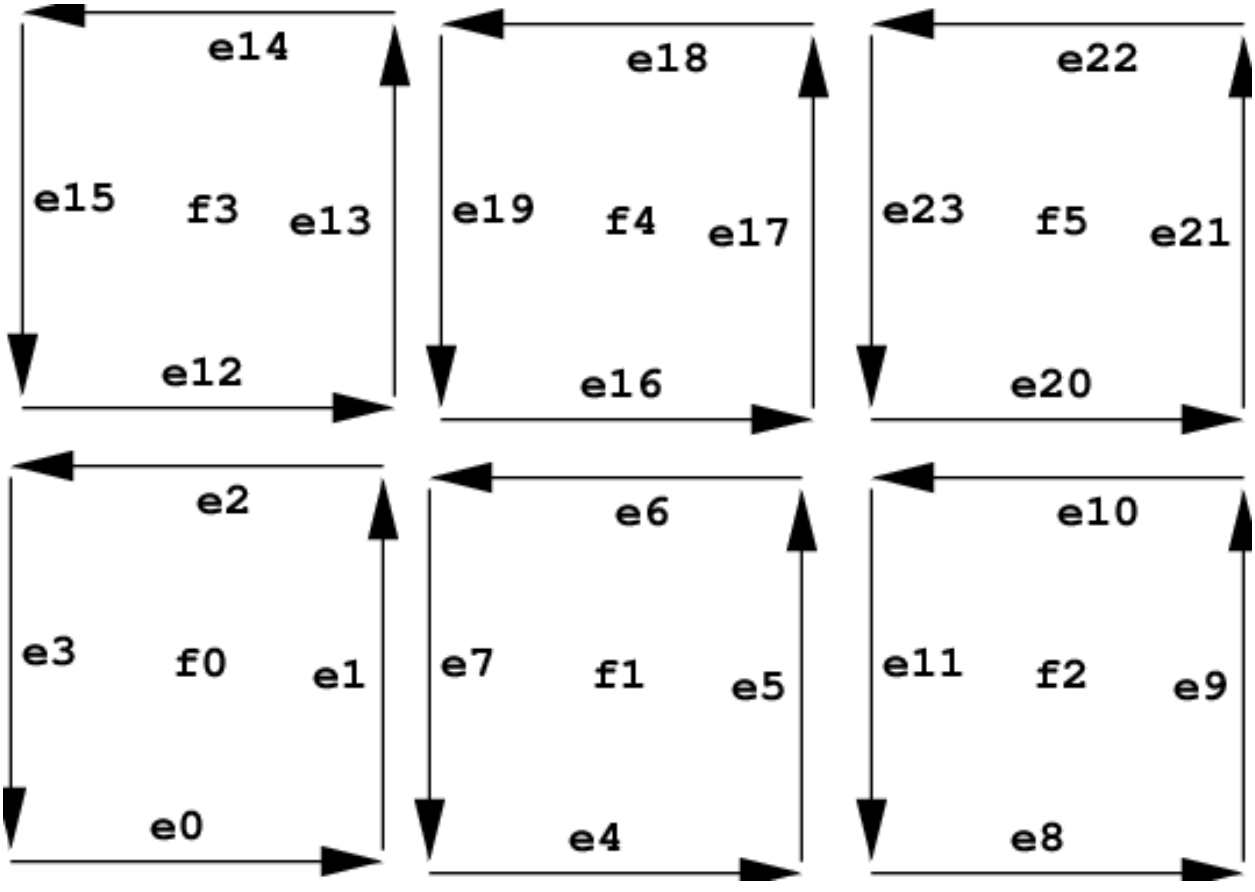
SYNOPSIS

```
#include <embree3/rtcore.h>

unsigned int rtcGetGeometryPreviousHalfEdge(
  RTCGeometry geometry,
  unsigned int edgeID
);
```

DESCRIPTION

The `rtcGetGeometryPreviousHalfEdge` function returns the ID of the previous half edge of the specified half edge (`edgeID` argument). For instance in the following example the previous half edge of `e6` is `e5`.



This function can only be used for subdivision geometries. As all topologies of a subdivision geometry share the same face buffer the function does not depend on the topology ID.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, *rtcGetGeometryFace*, *rtcGetGeometryOppositeHalfEdge*, *rtcGetGeometryNextHalfEdge*, *rtcGetGeometryPreviousHalfEdge*

rtcGetGeometryOppositeHalfEdge

NAME

```
rtcGetGeometryOppositeHalfEdge - returns the opposite half edge
```

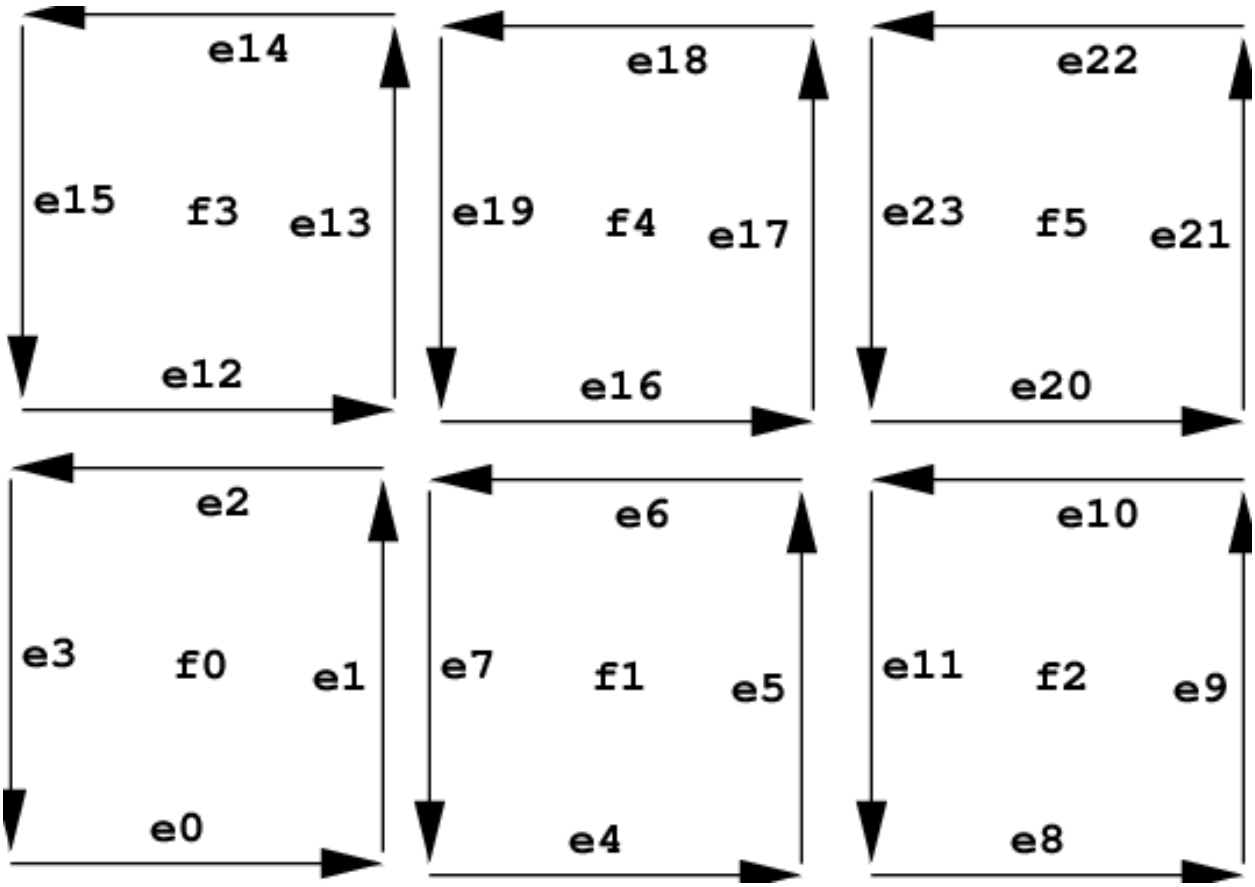
SYNOPSIS

```
#include <embree3/rtcore.h>

unsigned int rtcGetGeometryOppositeHalfEdge(
  RTCGeometry geometry,
  unsigned int topologyID,
  unsigned int edgeID
);
```

DESCRIPTION

The `rtcGetGeometryOppositeHalfEdge` function returns the ID of the opposite half edge of the specified half edge (`edgeID` argument) in the specified topology (`topologyID` argument). For instance in the following example the opposite half edge of `e6` is `e16`.



An opposite half edge does not exist if the specified half edge has either no neighboring face, or more than 2 neighboring faces. In these cases the function just returns the same edge `edgeID` again.

This function can only be used for subdivision geometries. The function depends on the topology as the topologies of a subdivision geometry have different index buffers assigned.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcGetGeometryFirstHalfEdge, *rtcGetGeometryFace*, *rtcGetGeometryOppositeHalfEdge*, *rtcGetGeometryNextHalfEdge*, *rtcGetGeometryPreviousHalfEdge*

rtcInterpolate

NAME

```
rtcInterpolate - interpolates vertex attributes
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCInterpolateArguments
{
  RTCGeometry geometry;
  unsigned int primID;
  float u;
  float v;
  enum RTCBufferType bufferType;
  unsigned int bufferSlot;
  float* P;
  float* dPdu;
  float* dPdv;
  float* ddPdudu;
  float* ddPdvdv;
  float* ddPdudv;
  unsigned int valueCount;
};

void rtcInterpolate(
  const struct RTCInterpolateArguments* args
);
```

DESCRIPTION

The `rtcInterpolate` function smoothly interpolates per-vertex data over the geometry. This interpolation is supported for triangle meshes, quad meshes, curve geometries, and subdivision geometries. Apart from interpolating the vertex attribute itself, it is also possible to get the first and second order derivatives of that value. This interpolation ignores displacements of subdivision surfaces and always interpolates the underlying base surface.

The `rtcInterpolate` call gets passed a number of arguments inside a structure of type `RTCInterpolateArguments`. For some geometry (`geometry` parameter) this function smoothly interpolates the per-vertex data stored inside the specified geometry buffer (`bufferType` and `bufferSlot` parameters) to the u/v location (`u` and `v` parameters) of the primitive (`primID` parameter). The number of floating point values to interpolate and store to the destination arrays can be specified using the `valueCount` parameter. As interpolation buffer, one can specify vertex buffers (`RTC_BUFFER_TYPE_VERTEX`) and vertex attribute buffers (`RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`) as well.

The `rtcInterpolate` call stores `valueCount` number of interpolated floating point values to the memory location pointed to by `P`. One can avoid storing the interpolated value by setting `P` to `NULL`.

The first order derivative of the interpolation by u and v are stored at the `dPdu` and `dPdv` memory locations. One can avoid storing first order derivatives by setting both `dPdu` and `dPdv` to `NULL`.

The second order derivatives are stored at the `ddPdudu`, `ddPdvdv`, and `ddPdudv` memory locations. One can avoid storing second order derivatives by setting these three pointers to `NULL`.

To use `rtcInterpolate` for a geometry, all changes to that geometry must be properly committed using `rtcCommitGeometry`.

All input buffers and output arrays must be padded to 16 bytes, as the implementation uses 16-byte SSE instructions to read and write into these buffers.

See tutorial Interpolation for an example of using the `rtcInterpolate` function.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcInterpolateN

rtcInterpolateN

NAME

```
rtcInterpolateN - performs N interpolations of vertex attribute data
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCInterpolateNArguments
{
  RTCGeometry geometry;
  const void* valid;
  const unsigned int* primIDs;
  const float* u;
  const float* v;
  unsigned int N;
  enum RTCBufferType bufferType;
  unsigned int bufferSlot;
  float* P;
  float* dPdu;
  float* dPdv;
  float* ddPdudu;
  float* ddPdvdv;
  float* ddPdudv;
  unsigned int valueCount;
};

void rtcInterpolateN(
  const struct RTCInterpolateNArguments* args
);
```

DESCRIPTION

The `rtcInterpolateN` is similar to `rtcInterpolate`, but performs `N` many interpolations at once. It additionally gets an array of u/v coordinates and a valid mask (`valid` parameter) that specifies which of these coordinates are valid. The valid mask points to `N` integers, and a value of -1 denotes valid and 0 invalid. If the valid pointer is `NULL` all elements are considers valid. The destination arrays are filled in structure of array (SOA) layout. The value `N` must be divisible by 4.

To use `rtcInterpolateN` for a geometry, all changes to that geometry must be properly committed using `rtcCommitGeometry`.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcInterpolate

rtcNewBuffer

NAME

```
rtcNewBuffer - creates a new data buffer
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCBuffer rtcNewBuffer(
  RTCDevice device,
  size_t byteSize
);
```

DESCRIPTION

The `rtcNewBuffer` function creates a new data buffer object of specified size in bytes (`byteSize` argument) that is bound to the specified device (`device` argument). The buffer object is reference counted with an initial reference count of 1. The returned buffer object can be released using the `rtcReleaseBuffer` API call. The specified number of bytes are allocated at buffer construction time and deallocated when the buffer is destroyed.

When the buffer will be used as a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` and `RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`), the last buffer element must be readable using 16-byte SSE load instructions, thus padding the last element is required for certain layouts. E.g. a standard `float3` vertex buffer layout should add storage for at least one more float to the end of the buffer.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcRetainBuffer, *rtcReleaseBuffer*

rtcNewSharedBuffer

NAME

```
rtcNewSharedBuffer - creates a new shared data buffer
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCBuffer rtcNewSharedBuffer(
  RTCDevice device,
  void* ptr,
  size_t byteSize
);
```

DESCRIPTION

The `rtcNewSharedBuffer` function creates a new shared data buffer object bound to the specified device (`device` argument). The buffer object is reference counted with an initial reference count of 1. The buffer can be released using the `rtcReleaseBuffer` function.

At construction time, the pointer to the user-managed buffer data (`ptr` argument) including its size in bytes (`byteSize` argument) is provided to create the buffer. At buffer construction time no buffer data is allocated, but the buffer data provided by the application is used. The buffer data must remain valid for as long as the buffer may be used, and the user is responsible to free the buffer data when no longer required.

When the buffer will be used as a vertex buffer (`RTC_BUFFER_TYPE_VERTEX` and `RTC_BUFFER_TYPE_VERTEX_ATTRIBUTE`), the last buffer element must be readable using 16-byte SSE load instructions, thus padding the last element is required for certain layouts. E.g. a standard `float3` vertex buffer layout should add storage for at least one more float to the end of the buffer.

The data pointer (`ptr` argument) must be aligned to 4 bytes; otherwise the `rtcNewSharedBuffer` function will fail.

EXIT STATUS

On failure `NULL` is returned and an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcRetainBuffer, *rtcReleaseBuffer*

rtcRetainBuffer

NAME

```
rtcRetainBuffer - increments the buffer reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcRetainBuffer(RTCBuffer buffer);
```

DESCRIPTION

Buffer objects are reference counted. The `rtcRetainBuffer` function increments the reference count of the passed buffer object (`buffer` argument). This function together with `rtcReleaseBuffer` allows to use the internal reference counting in a C++ wrapper class to handle the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBuffer, *rtcReleaseBuffer*

rtcReleaseBuffer

NAME

```
rtcReleaseBuffer - decrements the buffer reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcReleaseBuffer(RTCBuffer buffer);
```

DESCRIPTION

Buffer objects are reference counted. The `rtcReleaseBuffer` function decrements the reference count of the passed buffer object (`buffer` argument). When the reference count falls to 0, the buffer gets destroyed.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBuffer, *rtcRetainBuffer*

rtcGetBufferData

NAME

```
rtcGetBufferData - gets a pointer to the buffer data
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void* rtcGetBufferData(RTCBuffer buffer);
```

DESCRIPTION

The `rtcGetBufferData` function returns a pointer to the buffer data of the specified buffer object (`buffer` argument).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBuffer

RTCRay

NAME

```
RTCRay - single ray structure
```

SYNOPSIS

```
#include <embree3/rtcore_ray.h>

struct RTC_ALIGN(16) RTCRay
{
  float org_x;        // x coordinate of ray origin
  float org_y;        // y coordinate of ray origin
  float org_z;        // z coordinate of ray origin
  float tnear;        // start of ray segment

  float dir_x;        // x coordinate of ray direction
  float dir_y;        // y coordinate of ray direction
  float dir_z;        // z coordinate of ray direction
  float time;         // time of this ray for motion blur

  float tfar;         // end of ray segment (set to hit distance)
  unsigned int mask;  // ray mask
  unsigned int id;    // ray ID
  unsigned int flags; // ray flags
};
```

DESCRIPTION

The `RTCRay` structure defines the ray layout for a single ray. The ray contains the origin (`org_x`, `org_y`, `org_z` members), direction vector (`dir_x`, `dir_y`, `dir_z` members), and ray segment (`tnear` and `tfar` members). The ray direction does not have to be normalized, and only the parameter range specified by the `tnear`/`tfar` interval is considered valid.

The ray segment must be in the range $[0, \infty]$, thus ranges that start behind the ray origin are not allowed, but ranges can reach to infinity. For rays inside a ray stream, `tfar` < `tnear` identifies an inactive ray.

The ray further contains a motion blur time in the range $[0, 1]$ (`time` member), a ray mask (`mask` member), a ray ID (`id` member), and ray flags (`flags` member). The ray mask can be used to mask out some geometries for some rays (see `rtcSetGeometryMask` for more details). The ray ID can be used to identify a ray inside a callback function, even if the order of rays inside a ray packet or stream has changed. The ray flags are reserved.

The `embree3/rtcore_ray.h` header additionally defines the same ray structure in structure of array (SOA) layout for API functions accepting ray packets of size 4 (`RTCRay4` type), size 8 (`RTCRay8` type), and size 16 (`RTCRay16` type). The header additionally defines an `RTCRayNt` template for ray packets of an arbitrary compile-time size.

EXIT STATUS

SEE ALSO

RTCHit

RTCHit

NAME

```
RTCHit - single hit structure
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCHit
{
  float Ng_x;                                       // x coordinate of geometry␣
↪normal
  float Ng_y;                                       // y coordinate of geometry␣
↪normal
  float Ng_z;                                       // z coordinate of geometry␣
↪normal

  float u;                                          // barycentric u coordinate of␣
↪hit
  float v;                                          // barycentric v coordinate of␣
↪hit

  unsigned int primID;                              // geometry ID
  unsigned int geomID;                              // primitive ID
  unsigned int instID[RTC_MAX_INSTANCE_LEVEL_COUNT]; // instance ID
};
```

DESCRIPTION

The `RTCHit` type defines the type of a ray/primitive intersection result. The hit contains the unnormalized geometric normal in object space at the hit location (`Ng_x`, `Ng_y`, `Ng_z` members), the barycentric u/v coordinates of the hit (`u` and `v` members), as well as the primitive ID (`primID` member), geometry ID (`geomID` member), and instance ID stack (`instID` member) of the hit. The parametric intersection distance is not stored inside the hit, but stored inside the `tfar` member of the ray.

The `embree3/rtcore_ray.h` header additionally defines the same hit structure in structure of array (SOA) layout for hit packets of size 4 (`RTCHit4` type), size 8 (`RTCHit8` type), and size 16 (`RTCHit16` type). The header additionally defines an `RTCHitNt` template for hit packets of an arbitrary compile-time size.

EXIT STATUS

SEE ALSO

RTCRay, [Multi-Level Instancing]

RTCRayHit

NAME

```
RTCRayHit - combined single ray/hit structure
```

SYNOPSIS

```
#include <embree3/rtcore_ray.h>

struct RTCORE_ALIGN(16) RTCRayHit
{
  struct RTCRay ray;
  struct RTCHit hit;
};
```

DESCRIPTION

The `RTCRayHit` structure is used as input for the `rtcIntersect`-type functions and stores the ray to intersect and some hit fields that hold the intersection result afterwards.

The `embree3/rtcore_ray.h` header additionally defines the same ray/hit structure in structure of array (SOA) layout for API functions accepting ray packets of size 4 (`RTCRayHit4` type), size 8 (`RTCRayHit8` type), and size 16 (`RTCRayHit16` type). The header additionally defines an `RTCRayHitNt` template to generate ray/hit packets of an arbitrary compile-time size.

EXIT STATUS

SEE ALSO

RTCRay, *RTCHit*

RTCRayN

NAME

```
RTCRayN - ray packet of runtime size
```

SYNOPSIS

```
#include <embree3/rtcore_ray.h>

struct RTCRayN;

float& RTCRayN_org_x(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_org_y(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_org_z(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_tnear(RTCRayN* ray, unsigned int N, unsigned int i);

float& RTCRayN_dir_x(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_dir_y(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_dir_z(RTCRayN* ray, unsigned int N, unsigned int i);
float& RTCRayN_time (RTCRayN* ray, unsigned int N, unsigned int i);

float&        RTCRayN_tfar (RTCRayN* ray, unsigned int N, unsigned int i);
unsigned int& RTCRayN_mask (RTCRayN* ray, unsigned int N, unsigned int i);
unsigned int& RTCRayN_id   (RTCRayN* ray, unsigned int N, unsigned int i);
unsigned int& RTCRayN_flags(RTCRayN* ray, unsigned int N, unsigned int i);
```

DESCRIPTION

When the ray packet size is not known at compile time (e.g. when Embree returns a ray packet in the `RTCFilterFuncN` callback function), Embree uses the `RTCRayN` type for ray packets. These ray packets can only have sizes of 1, 4, 8, or 16. No other packet size will be used.

You can either implement different special code paths for each of these possible packet sizes and cast the ray to the appropriate ray packet type, or implement one general code path that uses the `RTCRayN_XXX` helper functions to access the ray packet components.

These helper functions get a pointer to the ray packet (`ray` argument), the packet size (`N` argument), and returns a reference to a component (e.g. x-component of origin) of the the i-th ray of the packet (`i` argument).

EXIT STATUS

SEE ALSO

RTCHitN

RTCHitN

NAME

```
RTCHitN - hit packet of runtime size
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct HitN;

float& RTCHitN_Ng_x(RTCHitN* hit, unsigned int N, unsigned int i);
float& RTCHitN_Ng_y(RTCHitN* hit, unsigned int N, unsigned int i);
float& RTCHitN_Ng_z(RTCHitN* hit, unsigned int N, unsigned int i);

float& RTCHitN_u(RTCHitN* hit, unsigned int N, unsigned int i);
float& RTCHitN_v(RTCHitN* hit, unsigned int N, unsigned int i);

unsigned& RTCHitN_primID(RTCHitN* hit, unsigned int N, unsigned int i);
unsigned& RTCHitN_geomID(RTCHitN* hit, unsigned int N, unsigned int i);
unsigned& RTCHitN_instID(RTCHitN* hit, unsigned int N, unsigned int i, unsigned int␣
↪level);
```

DESCRIPTION

When the hit packet size is not known at compile time (e.g. when Embree returns a hit packet in the `RTCFilterFuncN` callback function), Embree uses the `RTCHitN` type for hit packets. These hit packets can only have sizes of 1, 4, 8, or 16. No other packet size will be used.

You can either implement different special code paths for each of these possible packet sizes and cast the hit to the appropriate hit packet type, or implement one general code path that uses the `RTCHitN_XXX` helper functions to access hit packet components.

These helper functions get a pointer to the hit packet (`hit` argument), the packet size (`N` argument), and returns a reference to a component (e.g. x component of `Ng`) of the the i-th hit of the packet (`i` argument).

EXIT STATUS

SEE ALSO

RTCRayN

RTCRayHitN

NAME

```
RTCRayHitN - combined ray/hit packet of runtime size
```

SYNOPSIS

```
#include <embree3/rtcore_ray.h>

struct RTCRayHitN;

struct RTCRayN* RTCRayHitN_RayN(struct RTCRayHitN* rayhit, unsigned int N);
struct RTCHitN* RTCRayHitN_HitN(struct RTCRayHitN* rayhit, unsigned int N);
```

DESCRIPTION

When the packet size of a ray/hit structure is not known at compile time (e.g. when Embree returns a ray/hit packet in the `RTCIntersectFunctionN` callback function), Embree uses the `RTCRayHitN` type for ray packets. These ray/hit packets can only have sizes of 1, 4, 8, or 16. No other packet size will be used.

You can either implement different special code paths for each of these possible packet sizes and cast the ray/hit to the appropriate ray/hit packet type, or extract the `RTCRayN` and `RTCHitN` components using the `rtcGetRayN` and `rtcGetHitN` helper functions and use the `RTCRayN_XXX` and `RTCHitN_XXX` functions to access the ray and hit parts of the structure.

EXIT STATUS

SEE ALSO

RTCHitN

rtcInitIntersectContext

NAME

```
rtcInitIntersectContext - initializes the intersection context
```

SYNOPSIS

```
#include <embree3/rtcore.h>

enum RTCIntersectContextFlags
{
  RTC_INTERSECT_CONTEXT_FLAG_NONE,
  RTC_INTERSECT_CONTEXT_FLAG_INCOHERENT,
  RTC_INTERSECT_CONTEXT_FLAG_COHERENT,
};

struct RTCIntersectContext
{
  enum RTCIntersectContextFlags flags;
  RTCFilterFunctionN filter;

  #if RTC_MAX_INSTANCE_LEVEL_COUNT > 1
    unsigned int instStackSize;
  #endif

  unsigned int instID[RTC_MAX_INSTANCE_LEVEL_COUNT];

  #if RTC_MIN_WIDTH
    float minWidthDistanceFactor;
  #endif
};

void rtcInitIntersectContext(
  struct RTCIntersectContext* context
);
```

DESCRIPTION

A per ray-query intersection context (`RTCIntersectContext` type) is supported that can be used to configure intersection flags (`flags` member), specify a filter callback function (`filter` member), specify the chain of IDs of the current instance (`instID` and `instStackSize` members), and to attach arbitrary data to the query (e.g. per ray data).

The `rtcInitIntersectContext` function initializes the context to default values and should be called to initialize every intersection context. This function gets inlined, which minimizes overhead and allows for compiler optimizations.

The intersection context flag can be used to tune the behavior of the traversal algorithm. Using the `RTC_INTERSECT_CONTEXT_FLAG_INCOHERENT` flags uses an optimized traversal algorithm for incoherent rays (default), while `RTC_INTERSECT_CONTEXT_FLAG_COHERENT` uses an optimized traversal algorithm for coherent rays (e.g. primary camera rays).

Best primary ray performance can be obtained by using the ray stream API and setting the intersect context flag to `RTC_INTERSECT_CONTEXT_FLAG_COHERENT`. For secondary rays, it is typically better to use the `RTC_INTERSECT_CONTEXT_FLAG_INCOHERENT` flag, unless the rays are known to be very coherent too (e.g. for primary transparency rays).

A filter function can be specified inside the context. This filter function is invoked as a second filter stage after the per-geometry intersect or occluded filter function is invoked. Only rays that passed the first filter stage are valid in this second filter stage. Having such a per ray-query filter function can be useful to implement modifications of the behavior of the query, such as collecting all hits or accumulating transparencies. The support for the context filter function must be enabled for a scene by using the `RTC_SCENE_FLAG_CONTEXT_FILTER_FUNCTION` scene flag. In case of instancing this feature has to get enabled also for each instantiated scene.

The minWidthDistanceFactor value controls the target size of the curve radii when the min-width feature is enabled. Please see the [rtcSetGeometryMaxRadiusScale] function for more details on the min-width feature.

It is guaranteed that the pointer to the intersection context passed to a ray query is directly passed to the registered callback functions. This way it is possible to attach arbitrary data to the end of the intersection context, such as a per-ray payload.

Please note that the ray pointer is not guaranteed to be passed to the callback functions, thus reading additional data from the ray pointer passed to callbacks is not possible.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcIntersect1, *rtcOccluded1*

rtcIntersect1

NAME

```
rtcIntersect1 - finds the closest hit for a single ray
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcIntersect1(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayHit* rayhit
);
```

DESCRIPTION

The `rtcIntersect1` function finds the closest hit of a single ray with the scene (`scene` argument). The provided ray/hit structure (`rayhit` argument) contains the ray to intersect and some hit output fields that are filled when a hit is found.

The user has to initialize the ray origin (`org` ray member), ray direction (`dir` ray member), ray segment (`tnear`, `tfar` ray members), and set the ray flags to `0` (`flags` ray member). If the scene contains motion blur geometries, also the ray time (`time` ray member) must be initialized to a value in the range $[0, 1]$. If ray masks are enabled at compile time, the ray mask (`mask` ray member) must be initialized as well. The ray segment has to be in the range $[0, \infty]$, thus ranges that start behind the ray origin are not valid, but ranges can reach to infinity. See Section *RTCRay* for the ray layout description.

The geometry ID (`geomID` hit member) of the hit data must be initialized to `RTC_INVALID_GEOMETRY_ID` (-1).

Further, an intersection context for the ray query function must be created and initialized (see `rtcInitIntersectContext`).

When no intersection is found, the ray/hit data is not updated. When an intersection is found, the hit distance is written into the `tfar` member of the ray and all hit data is set, such as unnormalized geometry normal in object space (`Ng` hit member), local hit coordinates (`u`, `v` hit member), instance ID stack (`instID` hit member), geometry ID (`geomID` hit member), and primitive ID (`primID` hit member). See Section *RTCHit* for the hit layout description.

If the instance ID stack has a prefix of values not equal to `RTC_INVALID_GEOMETRY_ID`, the instance ID on each level corresponds to the geometry ID of the hit instance of the higher-level scene, the geometry ID corresponds to the hit geometry inside the hit instanced scene, and the primitive ID corresponds to the n-th primitive of that geometry.

If level 0 of the instance ID stack is equal to `RTC_INVALID_GEOMETRY_ID`, the geometry ID corresponds to the hit geometry inside the top-level scene, and the primitive ID corresponds to the n-th primitive of that geometry.

The implementation makes no guarantees that primitives whose hit distance is exactly at (or very close to) `tnear` or `tfar` are hit or missed. If you want to exclude intersections at `tnear` just pass a slightly enlarged `tnear`, and if you want to include intersections at `tfar` pass a slightly enlarged `tfar`.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The ray pointer passed to callback functions is not guaranteed to be identical to the original ray provided. To extend the ray with additional data to be accessed in callback functions, use the intersection context.

The ray/hit structure must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded1, *RTCRayHit*, *RTCRay*, *RTCHit*

rtcOccluded1

NAME

```
rtcOccluded1 - finds any hit for a single ray
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcOccluded1(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRay* ray
);
```

DESCRIPTION

The `rtcOccluded1` function checks for a single ray (`ray` argument) whether there is any hit with the scene (`scene` argument).

The user must initialize the ray origin (`org` ray member), ray direction (`dir` ray member), ray segment (`tnear`, `tfar` ray members), and must set the ray flags to `0` (`flags` ray member). If the scene contains motion blur geometries, also the ray time (`time` ray member) must be initialized to a value in the range $[0, 1]$. If ray masks are enabled at compile time, the ray mask (`mask` ray member) must be initialized as well. The ray segment must be in the range $[0, \infty]$, thus ranges that start behind the ray origin are not valid, but ranges can reach to infinity. See Section *RTCRay* for the ray layout description.

When no intersection is found, the ray data is not updated. In case a hit was found, the `tfar` component of the ray is set to `-inf`.

The implementation makes no guarantees that primitives whose hit distance is exactly at (or very close to) `tnear` or `tfar` are hit or missed. If you want to exclude intersections at `tnear` just pass a slightly enlarged `tnear`, and if you want to include intersections at `tfar` pass a slightly enlarged `tfar`.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The ray pointer passed to callback functions is not guaranteed to be identical to the original ray provided. To extend the ray with additional data to be accessed in callback functions, use the intersection context.

The ray must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded1, *RTCRay*

rtcIntersect4/8/16

NAME

```
rtcIntersect4/8/16 - finds the closest hits for a ray packet
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcIntersect4(
  const int* valid,
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayHit4* rayhit
);

void rtcIntersect8(
  const int* valid,
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayHit8* rayhit
);

void rtcIntersect16(
  const int* valid,
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayHit16* rayhit
);
```

DESCRIPTION

The `rtcIntersect4/8/16` functions finds the closest hits for a ray packet of size 4, 8, or 16 (`rayhit` argument) with the scene (`scene` argument). The ray/hit input contains a ray packet and hit packet. See Section *rtcIntersect1* for a description of how to set up and trace rays.

A ray valid mask must be provided (`valid` argument) which stores one 32-bit integer (`-1` means valid and `0` invalid) per ray in the packet. Only active rays are processed, and hit data of inactive rays is not changed.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The ray pointer passed to callback functions is not guaranteed to be identical to the original ray provided. To extend the ray with additional data to be accessed in callback functions, use the intersection context.

The implementation of these functions is guaranteed to invoke callback functions always with the same ray packet size and ordering of rays as specified initially.

For `rtcIntersect4` the ray packet must be aligned to 16 bytes, for `rtcIntersect8` the alignment must be 32 bytes, and for `rtcIntersect16` the alignment must be 64 bytes.

The `rtcIntersect4`, `rtcIntersect8` and `rtcIntersect16` functions may change the ray packet size and ray order when calling back into intersect filter functions or user geometry callbacks. Under some conditions the application can assume packets to stay intakt, which can determined by querying the `RTC_DEVICE_PROPERTY_NATIVE_RAY4_SUPPORTED`, `RTC_DEVICE_PROPERTY_NATIVE_RAY8_SUPPORTED`, `RTC_DEVICE_PROPERTY_NATIVE_RAY16_SUPPORTED` properties through the `rtcGetDeviceProperty` function. See *rtcGetDeviceProperty* for more information.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded4/8/16

rtcOccluded4/8/16

NAME

```
rtcOccluded4/8/16 - finds any hits for a ray packet
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcOccluded4(
  const int* valid,
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRay4* ray
);

void rtcOccluded8(
  const int* valid,
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRay8* ray
);

void rtcOccluded16(
  const int* valid,
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRay16* ray
);
```

DESCRIPTION

The `rtcOccluded4/8/16` functions checks for each active ray of the ray packet of size 4, 8, or 16 (`ray` argument) whether there is any hit with the scene (`scene` argument). See Section *rtcOccluded1* for a description of how to set up and trace occlusion rays.

A ray valid mask must be provided (`valid` argument) which stores one 32-bit integer (`-1` means valid and `0` invalid) per ray in the packet. Only active rays are processed, and hit data of inactive rays is not changed.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The ray pointer passed to callback functions is not guaranteed to be identical to the original ray provided. To extend the ray with additional data to be accessed in callback functions, use the intersection context.

The implementation of these functions is guaranteed to invoke callback functions always with the same ray packet size and ordering of rays as specified initially.

For `rtcOccluded4` the ray packet must be aligned to 16 bytes, for `rtcOccluded8` the alignment must be 32 bytes, and for `rtcOccluded16` the alignment must be 64 bytes.

The `rtcOccluded4`, `rtcOccluded8` and `rtcOccluded16` functions may change the ray packet size and ray order when calling back into intersect filter functions or user geometry callbacks. Under some conditions the application can assume packets to stay intakt, which can determined by querying the `RTC_DEVICE_PROPERTY_NATIVE_RAY4_SUPPORTED`, `RTC_DEVICE_PROPERTY_NATIVE_RAY8_SUPPORTED`, `RTC_DEVICE_PROPERTY_NATIVE_RAY16_SUPPORTED` properties through the `rtcGetDeviceProperty` function. See *rtcGetDeviceProperty* for more information.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded4/8/16

rtcIntersect1M

NAME

```
rtcIntersect1M - finds the closest hits for a stream of M single
  rays
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcIntersect1M(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayHit* rayhit,
  unsigned int M,
  size_t byteStride
);
```

DESCRIPTION

The `rtcIntersect1M` function finds the closest hits for a stream of `M` single rays (`rayhit` argument) with the scene (`scene` argument). The `rayhit` argument points to an array of ray and hit data with specified byte stride (`byteStride` argument) between the ray/hit structures. See Section *rtcIntersect1* for a description of how to set up and trace rays.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The implementation of the stream ray query functions may re-order rays arbitrarily and re-pack rays into ray packets of different size. For this reason, callback functions may be invoked with an arbitrary packet size (of size 1, 4, 8, or 16) and different ordering as specified initially. For this reason, one may have to use the `rayID` component of the ray to identify the original ray, e.g. to access a per-ray payload.

A ray in a ray stream is considered inactive if its `tnear` value is larger than its `tfar` value.

The stream size `M` can be an arbitrary positive integer including 0. Each ray must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded1M

rtcOccluded1M

NAME

```
rtcOccluded1M - finds any hits for a stream of M single rays
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcOccluded1M(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRay* ray,
  unsigned int M,
  size_t byteStride
);
```

DESCRIPTION

The `rtcOccluded1M` function checks whether there are any hits for a stream of `M` single rays (`ray` argument) with the scene (`scene` argument). The `ray` argument points to an array of rays with specified byte stride (`byteStride` argument) between the rays. See Section *rtcOccluded1* for a description of how to set up and trace occlusion rays.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The implementation of the stream ray query functions may re-order rays arbitrarily and re-pack rays into ray packets of different size. For this reason, callback functions may be invoked with an arbitrary packet size (of size 1, 4, 8, or 16) and different ordering as specified initially. For this reason, one may have to use the `rayID` component of the ray to identify the original ray, e.g. to access a per-ray payload.

A ray in a ray stream is considered inactive if its `tnear` value is larger than its `tfar` value.

The stream size `M` can be an arbitrary positive integer including 0. Each ray must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcIntersect1M

rtcIntersect1Mp

NAME

```
rtcIntersect1Mp - finds the closest hits for a stream of M pointers
  to single rays
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcIntersect1Mp(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayHit** rayhit,
  unsigned int M
);
```

DESCRIPTION

The `rtcIntersect1Mp` function finds the closest hits for a stream of `M` single rays (`rayhit` argument) with the scene (`scene` argument). The `rayhit` argument points to an array of pointers to the individual ray/hit structures. See Section *rtcIntersect1* for a description of how to set up and trace a ray.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The implementation of the stream ray query functions may re-order rays arbitrarily and re-pack rays into ray packets of different size. For this reason, callback functions may be invoked with an arbitrary packet size (of size 1, 4, 8, or 16) and different ordering as specified initially. For this reason, one may have to use the `rayID` component of the ray to identify the original ray, e.g. to access a per-ray payload.

A ray in a ray stream is considered inactive if its `tnear` value is larger than its `tfar` value.

The stream size `M` can be an arbitrary positive integer including 0. Each ray must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccluded1Mp

rtcOccluded1Mp

NAME

```
rtcOccluded1Mp - find any hits for a stream of M pointers to
  single rays
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcOccluded1M(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRay** ray,
  unsigned int M
);
```

DESCRIPTION

The `rtcOccluded1Mp` function checks whether there are any hits for a stream of `M` single rays (`ray` argument) with the scene (`scene` argument). The `ray` argument points to an array of pointers to rays. Section *rtcOccluded1* for a description of how to set up and trace a occlusion rays.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The implementation of the stream ray query functions may re-order rays arbitrarily and re-pack rays into ray packets of different size. For this reason, callback functions may be invoked with an arbitrary packet size (of size 1, 4, 8, or 16) and different ordering as specified initially. For this reason, one may have to use the `rayID` component of the ray to identify the original ray, e.g. to access a per-ray payload.

A ray in a ray stream is considered inactive if its `tnear` value is larger than its `tfar` value.

The stream size `M` can be an arbitrary positive integer including 0. Each ray must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcIntersect1Mp

rtcIntersectNM

NAME

```
rtcIntersectNM - finds the closest hits for a stream of M
  ray packets of size N
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcIntersectNM(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayHitN* rayhit,
  unsigned int N,
  unsigned int M,
  size_t byteStride
);
```

DESCRIPTION

The `rtcIntersectNM` function finds the closest hits for a stream of `M` ray packets (`rayhit` argument) of size `N` with the scene (`scene` argument). The `rays` argument points to an array of ray and hit packets with specified byte stride (`byteStride` argument) between the ray/hit packets. See Section *rtcIntersect1* for a description of how to set up and trace rays.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The implementation of the stream ray query functions may re-order rays arbitrarily and re-pack rays into ray packets of different size. For this reason, callback functions may be invoked with an arbitrary packet size (of size 1, 4, 8, or 16) and different ordering as specified initially. For this reason, one may have to use the `rayID` component of the ray to identify the original ray, e.g. to access a per-ray payload.

A ray in a ray stream is considered inactive if its `tnear` value is larger than its `tfar` value.

The packet size `N` must be larger than 0, and the stream size `M` can be an arbitrary positive integer including 0. Each ray must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccludedNM

rtcOccludedNM

NAME

```
rtcOccludedNM - finds any hits for a stream of M ray packets of
  size N
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcOccludedNM(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayN* ray,
  unsigned int N,
  unsigned int M,
  size_t byteStride
);
```

DESCRIPTION

The `rtcOccludedNM` function checks whether there are any hits for a stream of `M` ray packets (`ray` argument) of size `N` with the scene (`scene` argument). The `ray` argument points to an array of ray packets with specified byte stride (`byteStride` argument) between the ray packets. See Section *rtcOccluded1* for a description of how to set up and trace occlusion rays.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The implementation of the stream ray query functions may re-order rays arbitrarily and re-pack rays into ray packets of different size. For this reason, callback functions may be invoked with an arbitrary packet size (of size 1, 4, 8, or 16) and different ordering as specified initially. For this reason, one may have to use the `rayID` component of the ray to identify the original ray, e.g. to access a per-ray payload.

A ray in a ray stream is considered inactive if its `tnear` value is larger than its `tfar` value.

The packet size `N` must be larger than 0, and the stream size `M` can be an arbitrary positive integer including 0. Each ray must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcIntersectNM

rtcIntersectNp

NAME

```
rtcIntersectNp - finds the closest hits for a SOA ray stream of
  size N
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcIntersectNp(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayHitNp* rayhit,
  unsigned int N
);
```

DESCRIPTION

The `rtcIntersectNp` function finds the closest hits for a SOA ray stream (`rays` argument) of size `N` (basically a large ray packet) with the scene (`scene` argument). The `rayhit` argument points to two structures of pointers with one pointer for each ray and hit component. Each of these pointers points to an array with the ray or hit component data for each ray or hit. This way the individual components of the SOA ray stream do not need to be stored sequentially in memory, which makes it possible to have large varying size ray packets in SOA layout. See Section *rtcIntersect1* for a description of how to set up and trace rays.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The implementation of the stream ray query functions may re-order rays arbitrarily and re-pack rays into ray packets of different size. For this reason, callback functions may be invoked with an arbitrary packet size (of size 1, 4, 8, or 16) and different ordering as specified initially. For this reason, one may have to use the `rayID` component of the ray to identify the original ray, e.g. to access a per-ray payload.

A ray in a ray stream is considered inactive if its `tnear` value is larger than its `tfar` value.

The stream size `N` can be an arbitrary positive integer including 0. Each ray component array must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcOccludedNp

rtcOccludedNp

NAME

```
rtcOccludedNp - finds any hits for a SOA ray stream of size N
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcOccludedNp(
  RTCScene scene,
  struct RTCIntersectContext* context,
  struct RTCRayNp* ray,
  unsigned int N
);
```

DESCRIPTION

The `rtcOccludedNp` function checks whether there are any hits for a SOA ray stream (`ray` argument) of size `N` (basically a large ray packet) with the scene (`scene` argument). The `ray` argument points to a structure of pointers with one pointer for each ray component. Each of these pointers points to an array with the ray component data for each ray. This way the individual components of the SOA ray stream do not need to be stored sequentially in memory, which makes it possible to have large varying size ray packets in SOA layout. See Section *rtcOccluded1* for a description of how to set up and trace occlusion rays.

The intersection context (`context` argument) can specify flags to optimize traversal and a filter callback function to be invoked for every intersection. Further, the pointer to the intersection context is propagated to callback functions invoked during traversal and can thus be used to extend the ray with additional data. See Section `RTCIntersectContext` for more information.

The implementation of the stream ray query functions may re-order rays arbitrarily and re-pack rays into ray packets of different size. For this reason, callback functions may be invoked with an arbitrary packet size (of size 1, 4, 8, or 16) and different ordering as specified initially. For this reason, one may have to use the `rayID` component of the ray to identify the original ray, e.g. to access a per-ray payload.

A ray in a ray stream is considered inactive if its `tnear` value is larger than its `tfar` value.

The stream size `N` can be an arbitrary positive integer including 0. Each ray component array must be aligned to 16 bytes.

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcIntersectNp

rtcInitPointQueryContext

NAME

```
rtcInitPointQueryContext - initializes the context information (e.g.
  stack of (multilevel-)instance transformations) for point queries
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTC_ALIGN(16) RTCPointQueryContext
{
  // accumulated 4x4 column major matrices from world to instance space.
  float world2inst[RTC_MAX_INSTANCE_LEVEL_COUNT][16];

  // accumulated 4x4 column major matrices from instance to world space.
  float inst2world[RTC_MAX_INSTANCE_LEVEL_COUNT][16];

  // instance ids.
  unsigned int instID[RTC_MAX_INSTANCE_LEVEL_COUNT];

  // number of instances currently on the stack.
  unsigned int instStackSize;
};

void rtcInitPointQueryContext(
  struct RTCPointQueryContext* context
);
```

DESCRIPTION

A stack (`RTCPointQueryContext` type) which stores the IDs and instance transformations during a BVH traversal for a point query. The transformations are assumed to be affine transformations (3×3 matrix plus translation) and therefore the last column is ignored (see *RTC_GEOMETRY_TYPE_INSTANCE* for details).

The `rtcInitPointContext` function initializes the context to default values and should be called for initialization.

The context will be passed as an argument to the point query callback function (see *rtcSetGeometryPointQueryFunction*) and should be used to pass instance information down the instancing chain for user defined instancing (see tutorial [ClosestPoint] for a reference implementation of point queries with user defined instancing).

The context is an necessary argument to *rtcPointQuery* and Embree internally uses the topmost instance tranformation of the stack to transform the point query into instance space.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcPointQuery, *rtcSetGeometryPointQueryFunction*

rtcPointQuery

NAME

```
rtcPointQuery - traverses the BVH with a point query object
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTC_ALIGN(16) RTCPointQuery
{
  // location of the query
  float x;
  float y;
  float z;

  // radius and time of the query
  float radius;
  float time;
};

void rtcPointQuery(
  RTCScene scene,
  struct RTCPointQuery* query,
  struct RTCPointQueryContext* context,
  struct RTCPointQueryFunction* queryFunc,
  void* userPtr
);
```

DESCRIPTION

The `rtcPointQuery` function traverses the BVH using a `RTCPointQuery` object (`query` argument) and calls a user defined callback function (e.g `queryFunc` argument) for each primitive of the scene (`scene` argument) that intersects the query domain.

The user has to initialize the query location (`x`, `y` and `z` member) and query radius in the range $[0, \infty]$. If the scene contains motion blur geometries, also the query time (`time` member) must be initialized to a value in the range $[0, 1]$.

Further, a `RTCPointQueryContext` (`context` argument) must be created and initialized. It contains ID and transformation information of the instancing hierarchy if (multilevel-)instancing is used. See *rtcInitPointQueryContext* for further information.

For every primitive that intersects the query domain, the callback function (`queryFunc` argument) is called, in which distance computations to the primitive can be implemented. The user will be provided with the primID and geomID of the according primitive, however, the geometry information (e.g. triangle index and vertex data) has to be determined manually. The `userPtr` argument can be used to input geometry data of the scene or output results of the point query (e.g. closest point currently found on surface geometry (see tutorial [ClosestPoint])).

The parameter `queryFunc` is optional and can be NULL, in which case the callback function is not invoked. However, a callback function can still get attached to a specific `RTCGeometry` object using *rtcSetGeometryPointQueryFunction*. If a callback function is attached to a geometry and (a potentially different) callback function is passed as an argument to `rtcPointQuery`, both functions are called for the primitives of the according geometries.

The query radius can be decreased inside the callback function, which allows to efficiently cull parts of the scene during BVH traversal. Increasing the query radius and modifying time or location of the query will result in undefined behaviour.

The callback function will be called for all primitives in a leaf node of the BVH even if the primitive is outside the query domain, since Embree does not gather geometry information of primitives internally.

Point queries can be used with (multilevel)-instancing. However, care has to be taken when the instance transformation contains anisotropic scaling or sheering. In these cases distance computations have to be performed in world space to ensure correctness and the ellipsoidal query domain (in instance space) will be approximated with its axis aligned bounding box interally. Therefore, the callback function might be invoked even for primitives in inner BVH nodes that do not intersect the query domain. See *rtcSetGeometryPointQueryFunction* for details.

The point query structure must be aligned to 16 bytes.

SUPPORTED PRIMITIVES

Currenly, all primitive types are supported by the point query API except of points (see *RTC_GEOMETRY_TYPE_POINT*), curves (see *RTC_GEOMETRY_TYPE_CURVE*) and sudivision surfaces (see [RTC_GEOMETRY_SUBDIVISION]).

EXIT STATUS

For performance reasons this function does not do any error checks, thus will not set any error flags on failure.

SEE ALSO

rtcSetGeometryPointQueryFunction, *rtcInitPointQueryContext*

rtcCollide

NAME

```
rtcCollide - intersects one BVH with another
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTCCollision {
  unsigned int geomID0, primID0;
  unsigned int geomID1, primID1;
};

typedef void (*RTCCollideFunc) (
  void* userPtr,
  RTCCollision* collisions,
  size_t num_collisions);

void rtcCollide (
    RTCScene hscene0,
    RTCScene hscene1,
    RTCCollideFunc callback,
    void* userPtr
);
```

DESCRIPTION

The `rtcCollide` function intersects the BVH of `hscene0` with the BVH of scene `hscene1` and calls a user defined callback function (e.g `callback` argument) for each pair of intersecting primitives between the two scenes. A user defined data pointer (`userPtr` argument) can also be passed in.

For every pair of primitives that may intersect each other, the callback function (`callback` argument) is called. The user will be provided with the primID's and geomID's of multiple potentially intersecting primitive pairs. Currently, only scene entirely composed of user geometries are supported, thus the user is expected to implement a primitive/primitive intersection to filter out false positives in the callback function. The `userPtr` argument can be used to input geometry data of the scene or output results of the intersection query.

SUPPORTED PRIMITIVES

Currently, the only supported type is the user geometry type (see *RTC_GEOMETRY_TYPE_USER*).

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBVH

NAME

```
rtcNewBVH - creates a new BVH object
```

SYNOPSIS

```
#include <embree3/rtcore.h>

RTCBVH rtcNewBVH(RTCDevice device);
```

DESCRIPTION

This function creates a new BVH object and returns a handle to this BVH. The BVH object is reference counted with an initial reference count of 1. The handle can be released using the `rtcReleaseBVH` API call.

The BVH object can be used to build a BVH in a user-specified format over user-specified primitives. See the documentation of the `rtcBuildBVH` call for more details.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcRetainBVH, *rtcReleaseBVH*, *rtcBuildBVH*

rtcRetainBVH

NAME

```
rtcRetainBVH - increments the BVH reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcRetainBVH(RTCBVH bvh);
```

DESCRIPTION

BVH objects are reference counted. The `rtcRetainBVH` function increments the reference count of the passed BVH object (`bvh` argument). This function together with `rtcReleaseBVH` allows to use the internal reference counting in a C++ wrapper class to handle the ownership of the object.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBVH, *rtcReleaseBVH*

rtcReleaseBVH

NAME

```
rtcReleaseBVH - decrements the BVH reference count
```

SYNOPSIS

```
#include <embree3/rtcore.h>

void rtcReleaseBVH(RTCBVH bvh);
```

DESCRIPTION

BVH objects are reference counted. The `rtcReleaseBVH` function decrements the reference count of the passed BVH object (`bvh` argument). When the reference count falls to 0, the BVH gets destroyed.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBVH, *rtcRetainBVH*

rtcBuildBVH

NAME

```
rtcBuildBVH - builds a BVH
```

SYNOPSIS

```
#include <embree3/rtcore.h>

struct RTC_ALIGN(32) RTCBuildPrimitive
{
  float lower_x, lower_y, lower_z;
  unsigned int geomID;
  float upper_x, upper_y, upper_z;
  unsigned int primID;
};

typedef void* (*RTCCreateNodeFunction) (
  RTCThreadLocalAllocator allocator,
  unsigned int childCount,
  void* userPtr
);

typedef void (*RTCSetNodeChildrenFunction) (
  void* nodePtr,
  void** children,
  unsigned int childCount,
  void* userPtr
);

typedef void (*RTCSetNodeBoundsFunction) (
  void* nodePtr,
  const struct RTCBounds** bounds,
  unsigned int childCount,
  void* userPtr
);

typedef void* (*RTCCreateLeafFunction) (
  RTCThreadLocalAllocator allocator,
  const struct RTCBuildPrimitive* primitives,
  size_t primitiveCount,
  void* userPtr
);

typedef void (*RTCSplitPrimitiveFunction) (
  const struct RTCBuildPrimitive* primitive,
  unsigned int dimension,
  float position,
  struct RTCBounds* leftBounds,
  struct RTCBounds* rightBounds,
  void* userPtr
);
```

(continues on next page)

(continued from previous page)

```
typedef bool (*RTCProgressMonitorFunction)(
  void* userPtr, double n
);

enum RTCBuildFlags
{
  RTC_BUILD_FLAG_NONE,
  RTC_BUILD_FLAG_DYNAMIC
};

struct RTCBuildArguments
{
  size_t byteSize;

  enum RTCBuildQuality buildQuality;
  enum RTCBuildFlags buildFlags;
  unsigned int maxBranchingFactor;
  unsigned int maxDepth;
  unsigned int sahBlockSize;
  unsigned int minLeafSize;
  unsigned int maxLeafSize;
  float traversalCost;
  float intersectionCost;

  RTCBVH bvh;
  struct RTCBuildPrimitive* primitives;
  size_t primitiveCount;
  size_t primitiveArrayCapacity;

  RTCCreateNodeFunction createNode;
  RTCSetNodeChildrenFunction setNodeChildren;
  RTCSetNodeBoundsFunction setNodeBounds;
  RTCCreateLeafFunction createLeaf;
  RTCSplitPrimitiveFunction splitPrimitive;
  RTCProgressMonitorFunction buildProgress;
  void* userPtr;
};

struct RTCBuildArguments rtcDefaultBuildArguments();

void* rtcBuildBVH(
  const struct RTCBuildArguments* args
);
```

DESCRIPTION

The `rtcBuildBVH` function can be used to build a BVH in a user-defined format over arbitrary primitives. All arguments to the function are provided through the `RTCBuildArguments` structure. The first member of that structure must be set to the size of the structure in bytes (`bytesSize` member) which allows future extensions of the structure. It is recommended to initialize the build arguments structure using the `rtcDefaultBuildArguments` function.

The `rtcBuildBVH` function gets passed the BVH to build (`bvh` member), the array of primitives (`primitives` member), the capacity of that array (`primitiveArrayCapacity` member), the number of primitives stored inside the array (`primitiveCount` member), callback function pointers, and a user-defined pointer (`userPtr` member)

that is passed to all callback functions when invoked. The `primitives` array can be freed by the application after the BVH is built. All callback functions are typically called from multiple threads, thus their implementation must be thread-safe.

Four callback functions must be registered, which are invoked during build to create BVH nodes (`createNode` member), to set the pointers to all children (`setNodeChildren` member), to set the bounding boxes of all children (`setNodeBounds` member), and to create a leaf node (`createLeaf` member).

The function pointer to the primitive split function (`splitPrimitive` member) may be `NULL`, however, then no spatial splitting in high quality mode is possible. The function pointer used to report the build progress (`buildProgress` member) is optional and may also be `NULL`.

Further, some build settings are passed to configure the BVH build. Using the build quality settings (`buildQuality` member), one can select between a faster, low quality build which is good for dynamic scenes, and a standard quality build for static scenes. One can also specify the desired maximum branching factor of the BVH (`maxBranchingFactor` member), the maximum depth the BVH should have (`maxDepth` member), the block size for the SAH heuristic (`sahBlockSize` member), the minimum and maximum leaf size (`minLeafSize` and `maxLeafSize` member), and the estimated costs of one traversal step and one primitive intersection (`traversalCost` and `intersectionCost` members). When enabling the `RTC_BUILD_FLAG_DYNAMIC` build flags (`buildFlags` member), re-build performance for dynamic scenes is improved at the cost of higher memory requirements.

To spatially split primitives in high quality mode, the builder needs extra space at the end of the build primitive array to store splitted primitives. The total capacity of the build primitive array is passed using the `primitiveArrayCapacity` member, and should be about twice the number of primitives when using spatial splits.

The `RTCCreateNodeFunc` and `RTCCreateLeafFunc` callbacks are passed a thread local allocator object that should be used for fast allocation of nodes using the `rtcThreadLocalAlloc` function. We strongly recommend using this allocation mechanism, as alternative approaches like standard `malloc` can be over 10× slower. The allocator object passed to the create callbacks may be used only inside the current thread. Memory allocated using `rtcThreadLocalAlloc` is automatically freed when the `RTCBVH` object is deleted. If you use your own memory allocation scheme you have to free the memory yourself when the `RTCBVH` object is no longer used.

The `RTCCreateNodeFunc` callback additionally gets the number of children for this node in the range from 2 to `maxBranchingFactor` (`childCount` argument).

The `RTCSetNodeChildFunc` callback function gets a pointer to the node as input (`nodePtr` argument), an array of pointers to the children (`childPtrs` argument), and the size of this array (`childCount` argument).

The `RTCSetNodeBoundsFunc` callback function gets a pointer to the node as input (`nodePtr` argument), an array of pointers to the bounding boxes of the children (`bounds` argument), and the size of this array (`childCount` argument).

The `RTCCreateLeafFunc` callback additionally gets an array of primitives as input (`primitives` argument), and the size of this array (`primitiveCount` argument). The callback should read the `geomID` and `primID` members from the passed primitives to construct the leaf.

The `RTCSplitPrimitiveFunc` callback is invoked in high quality mode to split a primitive (`primitive` argument) at the specified position (`position` argument) and dimension (`dimension` argument). The callback should return bounds of the clipped left and right parts of the primitive (`leftBounds` and `rightBounds` arguments).

The `RTCProgressMonitorFunction` callback function is called with the estimated completion rate `n` in the range $[0, 1]$. Returning `true` from the callback lets the build continue; returning `false` cancels the build.

EXIT STATUS

On failure an error code is set that can be queried using `rtcGetDeviceError`.

SEE ALSO

rtcNewBVH

RTCQuaternionDecomposition

NAME

```
RTCQuaternionDecomposition - structure that represents a quaternion
  decomposition of an affine transformation
```

SYNOPSIS

```
struct RTCQuaternionDecomposition
{
  float scale_x, scale_y, scale_z;
  float skew_xy, skew_xz, skew_yz;
  float shift_x, shift_y, shift_z;
  float quaternion_r, quaternion_i, quaternion_j, quaternion_k;
  float translation_x, translation_y, translation_z;
};
```

DESCRIPTION

The struct `RTCQuaternionDecomposition` represents an affine transformation decomposed into three parts. An upper triangular scaling/skew/shift matrix

$$S = \begin{pmatrix} scale_x & skew_{xy} & skew_{xz} & shift_x \\ 0 & scale_y & skew_{yz} & shift_y \\ 0 & 0 & scale_z & shift_z \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

a translation matrix

$$T = \begin{pmatrix} 1 & 0 & 0 & translation_x \\ 0 & 1 & 0 & translation_y \\ 0 & 0 & 1 & translation_z \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

and a rotation matrix R, represented as a quaternion

$quaternion_r + quaternion_i\ \mathbf{i} + quaternion_j\ \mathbf{i} + quaternion_k\ \mathbf{k}$

where **i**, **j** **k** are the imaginary quaternion units. The passed quaternion will be normalized internally.

The affine transformation matrix corresponding to a `RTCQuaternionDecomposition` is TRS and a point $p = (p_x, p_y, p_z, 1)^T$ will be transformed as

$$p' = T\ R\ S\ p.$$

The functions `rtcInitQuaternionDecomposition`, `rtcQuaternionDecompositionSetQuaternion`, `rtcQuaternionDecompositionSetScale`, `rtcQuaternionDecompositionSetSkew`, `rtcQuaternionDecompositionSetShift`, and `rtcQuaternionDecompositionSetTranslation` allow to set the fields of the structure more conveniently.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcSetGeometryTransformQuaternion, *rtcInitQuaternionDecomposition*

rtcInitQuaternionDecomposition

NAME

```
rtcInitQuaternionDecomposition - initializes quaternion decomposition
```

SYNOPSIS

```
void rtcInitQuaternionDecomposition(
  struct RTCQuaternionDecomposition* qd
);
```

DESCRIPTION

The `rtcInitQuaternionDecomposition` function initializes a `RTCQuaternionDecomposition` structure to represent an identity transformation.

EXIT STATUS

No error code is set by this function.

SEE ALSO

rtcSetGeometryTransformQuaternion, *RTCQuaternionDecomposition*

Open VKL

Open Volume Kernel Library (Open VKL) is a collection of high-performance volume computation kernels.

Introduction

Open Volume Kernel Library (Open VKL) is a collection of high-performance volume computation kernels. The target users of Open VKL are graphics application engineers who want to improve the performance of their volume rendering applications by leveraging Open VKL's performance-optimized kernels, which include volume traversal and sampling functionality for a variety of volumetric data formats.

Open VKL provides a C API, and also supports applications written with the Intel® SPMD Program Compiler (ISPC) by also providing an ISPC interface to the core volume algorithms. This makes it possible to write a renderer in ISPC that automatically vectorizes and leverages SSE, AVX, AVX2, and AVX-512 instructions. ISPC also supports runtime code selection, thus ISPC will select the best code path for your application.

Open VKL API

To access the Open VKL API you first need to include the Open VKL header. For C99 or C++:

```
#include <openvkl/openvkl.h>
```

For the Intel SPMD Program Compiler (ISPC):

```
#include <openvkl/openvkl.isph>
```

This documentation will discuss the C99/C++ API. The ISPC version has the same functionality and flavor. Looking at the headers, the `vklTutorialISPC` example, and this documentation should be enough to figure it out.

Initialization and shutdown

To use the API, one of the implemented backends must be loaded. Currently the only one that exists is the ISPC driver. ISPC in the name here just refers to the implementation language – it can also be used from the C99/C++ APIs. To load the module that implements the ISPC driver:

```
vklLoadModule("ispc_driver");
```

The driver then needs to be instantiated:

```
VKLDriver driver = vklNewDriver("ispc");
```

By default, the ISPC driver selects the maximum supported SIMD width (and associated ISA) for the system. Optionally, a specific width may be requested using the `ispc_4`, `ispc_8`, or `ispc_16` driver names. Note that the system must support the given width (SSE4.1 for 4-wide, AVX for 8-wide, and AVX512 for 16-wide).

Once a driver is created, you can call

```
void vklDriverSetInt(VKLDriver, const char *name, int val);
void vklDriverSetString(VKLDriver, const char *name, const char *val);
```

to set parameters on the driver. The following parameters are understood by all drivers:

Table 1: Parameters shared by all drivers.

Type	Name	Description
int	logLevel	logging level; valid values are `VKL_LOG_DEBUG`, `VKL_LOG_INFO`, `VKL_LOG_WARNING`, `VKL_LOG_ERROR` and `VKL_LOG_NONE`
string	logOutput	convenience for setting where log messages go; valid values are `cout`, `cerr` and `none`
string	errorOutput	convenience for setting where error messages go; valid values are `cout`, `cerr` and `none`
int	numThreads	number of threads which Open VKL can use
int	flushDenormals	sets the `Flush to Zero` and `Denormals are Zero` mode of the MXCSR control and status register (default: 1); see Performance Recommendations section for details

Once parameters are set, the driver must be committed with

```
vklCommitDriver(driver);
```

Finally, to use the newly committed driver, you must call

```
vklSetCurrentDriver(driver);
```

Users can change parameters on a driver after initialization. In this case the driver would need to be re-committed. If changes are made to the driver that is already set as the current driver, it does not need to be set as current again. The currently set driver can be retrieved at any time by calling

```
VKLDriver driver = vklGetCurrentDriver();
```

Open VKL provides vector-wide versions for several APIs. To determine the native vector width for the given driver, call:

```
int width = vklGetNativeSIMDWidth();
```

When the application is finished with Open VKL or shutting down, call the shutdown function:

```
vklShutdown();
```

Environment variables

The generic driver parameters can be overridden via environment variables for easy changes to Open VKL's behavior without needing to change the application (variables are prefixed by convention with "`OPENVKL_`"):

Table 2: Environment variables understood by all drivers.

Variable	Description
OPENVKL_LOG_LEVEL	logging level; valid values are `debug`, `info`, `warning`, `error` and `none`
OPENVKL_LOG_OUTPUT	convenience for setting where log messages go; valid values are `cout`, `cerr` and `none`
OPENVKL_ERROR_OUTPUT	convenience for setting where error messages go; valid values are `cout`, `cerr` and `none`
OPENVKL_THREADS	number of threads which Open VKL can use
OPENVKL_FLUSH_DENORMALS	sets the `Flush to Zero` and `Denormals are Zero` mode of the MXCSR control and status register (default: 1); see Performance Recommendations section for details

Note that these environment variables take precedence over values set through the `vklDriverSet*()` functions.

Error handling and log messages

The following errors are currently used by Open VKL:

Table 3: Possible error codes, i.e., valid named constants of type `VKLError`.

Name	Description
VKL_NO_ERROR	no error occurred
VKL_UNKNOWN_ERROR	an unknown error occurred
VKL_INVALID_ARGUMENT	an invalid argument was specified
VKL_INVALID_OPERATION	the operation is not allowed for the specified object
VKL_OUT_OF_MEMORY	there is not enough memory to execute the command
VKL_UNSUPPORTED_CPU	the CPU is not supported (minimum ISA is SSE4.1)

These error codes are either directly returned by some API functions, or are recorded to be later queried by the application via

```
VKLError vklDriverGetLastErrorCode(VKLDriver);
```

A more descriptive error message can be queried by calling

```
const char* vklDriverGetLastErrorMsg(VKLDriver);
```

Alternatively, the application can also register a callback function of type

```
typedef void (*VKLErrorCallback)(void *, VKLError, const char* message);
```

via

```
void vklDriverSetErrorCallback(VKLDriver, VKLErrorFunc, void *);
```

to get notified when errors occur. Applications may be interested in messages which Open VKL emits, whether for debugging or logging events. Applications can register a callback function of type

```
typedef void (*VKLLogCallback)(void *, const char* message);
```

via

```
void vklDriverSetLogCallback(VKLDriver, VKLLogCallback, void *);
```

which Open VKL will use to emit log messages. Applications can clear either callback by passing `nullptr` instead of an actual function pointer. By default, Open VKL uses `cout` and `cerr` to emit log and error messages, respectively. The last parameter to `vklDriverSetErrorCallback` and `vklDriverSetLogCallback` is a user data pointer. Open VKL passes this pointer to the callback functions as the first parameter. Note that in addition to setting the above callbacks, this behavior can be changed via the driver parameters and environment variables described previously.

Basic data types

Open VKL defines 3-component vectors of integer and vector types:

```
typedef struct
{
  int x, y, z;
} vkl_vec3i;

typedef struct
{
  float x, y, z;
} vkl_vec3f;
```

Vector versions of these are also defined in structure-of-array format for 4, 8, and 16 wide types.

```
typedef struct
{
  float x[WIDTH];
  float y[WIDTH];
  float z[WIDTH];
} vkl_vvec3f##WIDTH;

typedef struct
{
  float lower[WIDTH], upper[WIDTH];
} vkl_vrange1f##WIDTH;
```

1-D range and 3-D ranges are defined as ranges and boxes, with no vector versions:

```
typedef struct
{
  float lower, upper;
} vkl_range1f;

typedef struct
{
  vkl_vec3f lower, upper;
} vkl_box3f;
```

Object model

Objects in Open VKL are exposed to the APIs as handles with internal reference counting for lifetime determination. Objects are created with particular type's `vklNew...` API entry point. For example, `vklNewData` and `vklNewVolume`.

In general, modifiable parameters to objects are modified using `vklSet...` functions based on the type of the parameter being set. The parameter name is passed as a string. Below are all variants of `vklSet...`.

```
void vklSetBool(VKLObject object, const char *name, int b);
void vklSetFloat(VKLObject object, const char *name, float x);
void vklSetVec3f(VKLObject object, const char *name, float x, float y, float z);
void vklSetInt(VKLObject object, const char *name, int x);
void vklSetVec3i(VKLObject object, const char *name, int x, int y, int z);
void vklSetData(VKLObject object, const char *name, VKLData data);
```

(continues on next page)

(continued from previous page)

```
void vklSetString(VKLObject object, const char *name, const char *s);
void vklSetVoidPtr(VKLObject object, const char *name, void *v);
```

The exception to this rule is the `VKLValueSelector` object (described in the iterators section below), which has object-specific set methods. The reason for this is to align the C99/C++ API with the ISPC API, which can't use a parameter method due to language limitations.

After parameters have been set, `vklCommit` must be called on the object to make them take effect.

Open VKL uses reference counting to manage the lifetime of all objects. Therefore one cannot explicitly "delete" any object. Instead, one can indicate the application does not need or will not access the given object anymore by calling

```
void vklRelease(VKLObject);
```

This decreases the object's reference count. If the count reaches `0` the object will automatically be deleted.

Managed data

Large data is passed to Open VKL via a `VKLData` handle created with `vklNewData`:

```
VKLData vklNewData(size_t numItems,
                   VKLDataType dataType,
                   const void *source,
                   VKLDataCreationFlags dataCreationFlags,
                   size_t byteStride);
```

Types accepted are listed in `VKLDataType.h`; basic types (`UCHAR`, `INT`, `UINT`, `LONG`, `ULONG`, `FLOAT`) exist as both scalar and chunked formats. The types accepted vary per volume at the moment; read the volume section below for specifics.

Data objects can be created as Open VKL owned (`dataCreationFlags = VKL_DATA_DEFAULT`), in which the library will make a copy of the data for its use, or shared (`dataCreationFlags = VKL_DATA_SHARED_BUFFER`), which will try to use the passed pointer for usage. The library is allowed to copy data when a volume is committed.

The distance between consecutive elements in `source` is given in bytes with `byteStride`. If the provided `byteStride` is zero, then it will be determined automatically as `sizeof(type)`. Open VKL owned data will be compacted into a naturally-strided array on copy, regardless of the original `byteStride`.

As with other object types, when data objects are no longer needed they should be released via `vklRelease`.

Observers

Volumes and samplers in Open VKL may provide observers to communicate data back to the application. Observers may be created with

```
VKLObserver vklNewSamplerObserver(VKLSampler sampler,
                                  const char *type);

VKLObserver vklNewVolumeObserver(VKLVolume volume,
                                 const char *type);
```

The object passed to `vklNew*Observer` must already be committed. Valid observer type strings are defined by volume implementations (see section 'Volume types' below).

`vklNew*Observer` returns `NULL` on failure.

To access the underlying data, an observer must first be mapped using

```
const void * vklMapObserver(VKLObserver observer);
```

If this fails, the function returns `NULL`. `vklMapObserver` may fail on observers that are already mapped. On success, the application may query the underlying type and the number of elements in the buffer using

```
VKLDataType vklGetObserverElementType(VKLObserver observer);
size_t vklGetObserverNumElements(VKLObserver observer);
```

On failure, these functions return `VKL_UNKNOWN` and `0`, respectively. Possible data types are defined by the volume that provides the observer , as are the semantics of the observation. See section 'Volume types' for details.

The pointer returned by `vklMapObserver` may be cast to the type corresponding to the value returned by `vklGetObserverElementType` to access the observation. For example, if `vklGetObserverElementType` returns `VKL_FLOAT`, then the pointer returned by `vklMapObserver` may be cast to `const float *` to access up to `vklGetObserverNumElements` consecutive values of type `float`.

Once the application has finished processing the observation, it should unmap the observer using

```
void vklUnmapObserver(VKLObserver observer);
```

so that the observer may be mapped again.

When an observer is no longer needed, it should be released using `vklRelease`.

The observer API is not thread safe, and these functions should not be called concurrently on the same object.

Volume types

Open VKL currently supports structured volumes on regular and spherical grids; unstructured volumes with tetrahedral, wedge, pyramid, and hexaderal primitive types; adaptive mesh refinement (AMR) volumes; sparse VDB volumes; and particle volumes. These volumes are created with `vklNewVolume` with the appropriate type string.

In addition to the usual `vklSet...()` and `vklCommit()` APIs, the volume bounding box can be queried:

```
vkl_box3f vklGetBoundingBox(VKLVolume volume);
```

The number of attributes in a volume can also be queried:

```
unsigned int vklGetNumAttributes(VKLVolume volume);
```

Finally, the value range of the volume (first attribute only) can be queried:

```
vkl_range1f vklGetValueRange(VKLVolume volume);
```

Structured Volumes

Structured volumes only need to store the values of the samples, because their addresses in memory can be easily computed from a 3D position. The dimensions for all structured volume types are in units of vertices, not cells. For example, a volume with dimensions (x, y, z) will have $(x - 1, y - 1, z - 1)$ cells in each dimension. Voxel data provided is assumed vertex-centered, so $x * y * z$ values must be provided.

Structured Regular Volumes

A common type of structured volumes are regular grids, which are created by passing a type string of `"structuredRegular"` to `vklNewVolume`. The parameters understood by structured regular volumes are summarized in the table below.

Table 4: Configuration parameters for structured regular (`"structuredRegular"`) volumes.

Type	Name	Default	Description
vec3i	dimensions		number of voxels in each dimension (x, y, z)
VKLData VKLData[]	data		VKLData object(s) of voxel data, supported types are:
			`VKL_UCHAR`
			`VKL_SHORT`
			`VKL_USHORT`
			`VKL_FLOAT`
			`VKL_DOUBLE`
			Multiple attributes are supported through passing an array of VKLData objects.
vec3f	gridOrigin	$(0, 0, 0)$	origin of the grid in object space
vec3f	gridSpacing	$(1, 1, 1)$	size of the grid cells in object space
int	temporallyStructured-NumTimesteps		for temporally structured variation, number of timesteps per voxel
uint32[] uint64[]	temporallyUnstruc-turedIndices		for temporally unstructured variation, indices to `data` time series beginning per voxel
float[]	temporallyUnstructured-Times		for temporally unstructured variation, time values corresponding to values in `data`

Structured regular volumes support two forms of temporal variation: temporally structured and temporally unstructured. When one of these modes is enabled, the volume can be sampled at different times. In both modes, time is assumed to vary between zero and one. This can be useful for implementing renderers with motion blur, for example.

Temporally structured variation is configured through the `temporallyStructuredNumTimesteps` parameter. This specifies how many time steps (at least two) are provided for all voxels. Therefore, for a volume with dimensions (x, y, z), each attribute must have $x * y * z * temporallyStructuredNumTimesteps$ values provided in its `data` array. The values are assumed evenly spaced over times $[0, 1]$.

Temporally unstructured variation is configured through the `temporallyUnstructuredIndices` and `temporallyUnstructuredTimes` parameters, and supports differing time step counts and sample times per voxel. `temporallyUnstructuredIndices` specifies the index ranges for each voxel's values in `data`, such that values for the the ith voxel can be found in the indices $[temporallyUnstructuredIndices[i], temporallyUnstructuredIndices[i + 1])$. Therefore `temporallyUnstructuredIndices` must have $x * y * z + 1$ values. `temporallyUnstructuredTimes` specifies the times corresponding to the sample values in each attribute's `data` array; the time values for each voxel

must be between zero and one and strictly increasing: $t0 < t1 < ... < tN$. To return a value at sample time t, $t0 <= t <= tN$, Open VKL will interpolate linearly from the two nearest time steps. Time values outside this range are clamped $[t0, tN]$.

The following additional parameters can be set both on `"structuredRegular"` volumes and their sampler objects. Sampler object parameters default to volume parameters.

Table 5: Configuration parameters for structured regular (`"structuredRegular"`) volumes and their sampler objects.

Type	Name	Default	Description
int	filter	`VKL_FILTER_TRILINEAR`	The filter used for reconstructing the field. Use `VKLFilter` for named constants.
int	gradientFilter	`filter`	The filter used for reconstructing the field during gradient computations. Use `VKLFilter` for named constants.

Structured Spherical Volumes

Structured spherical volumes are also supported, which are created by passing a type string of `"structuredSpherical"` to `vklNewVolume`. The grid dimensions and parameters are defined in terms of radial distance (r), inclination angle (θ), and azimuthal angle (ϕ), conforming with the ISO convention for spherical coordinate systems. The coordinate system and parameters understood by structured spherical volumes are summarized below.

[Structured spherical volume coordinate system: radial distance (r), inclination angle (θ), and azimuthal angle (ϕ).][imgStructuredSphericalCoords]

Table 6: Configuration parameters for structured spherical (`"structuredSpherical"`) volumes.

Type	Name	Default	Description
vec3i	dimensions		number of voxels in each dimension (r, θ, ϕ)
VKLData VKLData[]	data		VKLData object(s) of voxel data, supported types are:
			`VKL_UCHAR`
			`VKL_SHORT`
			`VKL_USHORT`
			`VKL_FLOAT`
			`VKL_DOUBLE`
			Multiple attributes are supported through passing an array of VKLData objects.
vec3f	gridOrigin	$(0, 0, 0)$	origin of the grid in units of (r, θ, ϕ); angles in degrees
vec3f	gridSpacing	$(1, 1, 1)$	size of the grid cells in units of (r, θ, ϕ); angles in degrees

These grid parameters support flexible specification of spheres, hemispheres, spherical shells, spherical wedges, and so forth. The grid extents (computed as $[gridOrigin, gridOrigin + (dimensions - 1) * gridSpacing]$) however must be constrained such that:

- $r \geq 0$
- $0 \leq \theta \leq 180$

- $0 \leq \phi \leq 360$

The following additional parameters can be set both on `"structuredSpherical"` volumes and their sampler objects. Sampler object parameters default to volume parameters.

Table 7: Configuration parameters for structured spherical (`"structuredSpherical"`) volumes and their sampler objects.

Type	Name	Default	Description
int	filter	`VKL_FILTER_TRILINEAR`	The filter used for reconstructing the field. Use `VKLFilter` for named constants.
int	gradientFilter	`filter`	The filter used for reconstructing the field during gradient computations. Use `VKLFilter` for named constants.

Adaptive Mesh Refinement (AMR) Volumes

Open VKL currently supports block-structured (Berger-Colella) AMR volumes. Volumes are specified as a list of blocks, which exist at levels of refinement in potentially overlapping regions. Blocks exist in a tree structure, with coarser refinement level blocks containing finer blocks. The cell width is equal for all blocks at the same refinement level, though blocks at a coarser level have a larger cell width than finer levels.

There can be any number of refinement levels and any number of blocks at any level of refinement.

Blocks are defined by three parameters: their bounds, the refinement level in which they reside, and the scalar data contained within each block.

Note that cell widths are defined *per refinement level*, not per block.

AMR volumes are created by passing the type string `"amr"` to `vklNewVolume`, and have the following parameters:

Table 8: Configuration parameters for AMR (`"amr"`) volumes.

Type	Name	Default	Description
float[]	cellWidth		[data] array of each level's cell width
box3i[]	block.bounds		[data] array of each block's bounds (in voxels)
int[]	block.level		[data] array of each block's refinement level
VKLData[]	block.data		[data] array of each block's VKLData object containing the actual scalar voxel data. Currently only `VKL_FLOAT` data is supported.
vec3f	gridOrigin	$(0, 0, 0)$	origin of the grid in object space
vec3f	gridSpacing	$(1, 1, 1)$	size of the grid cells in object space

Note that the `gridOrigin` and `gridSpacing` parameters act just like the structured volume equivalent, but they only modify the root (coarsest level) of refinement.

The following additional parameters can be set both on `"amr"` volumes and their sampler objects. Sampler object parameters default to volume parameters.

Table 9: Configuration parameters for AMR ("AMR") volumes and their sampler objects.

Type	Name	Default	Description
VKLAMRMethod	method	VKL_AMR_CURRENT	VKLAMRMethod sampling method. Supported methods are:
			VKL_AMR_CURRENT
			VKL_AMR_FINEST
			VKL_AMR_OCTANT

Open VKL's AMR implementation was designed to cover Berger-Colella [1] and Chombo [2] AMR data. The method parameter above determines the interpolation method used when sampling the volume.

- VKL_AMR_CURRENT finds the finest refinement level at that cell and interpolates through this "current" level
- VKL_AMR_FINEST will interpolate at the closest existing cell in the volume-wide finest refinement level regardless of the sample cell's level
- VKL_AMR_OCTANT interpolates through all available refinement levels at that cell. This method avoids discontinuities at refinement level boundaries at the cost of performance

Details and more information can be found in the publication for the implementation [3].

1. M. J. Berger, and P. Colella. "Local adaptive mesh refinement for shock hydrodynamics." Journal of Computational Physics 82.1 (1989): 64-84. DOI: 10.1016/0021-9991(89)90035-1

2. M. Adams, P. Colella, D. T. Graves, J.N. Johnson, N.D. Keen, T. J. Ligocki. D. F. Martin. P.W. McCorquodale, D. Modiano. P.O. Schwartz, T.D. Sternberg and B. Van Straalen, Chombo Software Package for AMR Applications - Design Document, Lawrence Berkeley National Laboratory Technical Report LBNL-6616E.

3. I. Wald, C. Brownlee, W. Usher, and A. Knoll. CPU volume rendering of adaptive mesh refinement data. SIGGRAPH Asia 2017 Symposium on Visualization on - SA '17, 18(8), 1–8. DOI: 10.1145/3139295.3139305

Unstructured Volumes

Unstructured volumes can have their topology and geometry freely defined. Geometry can be composed of tetrahedral, hexahedral, wedge or pyramid cell types. The data format used is compatible with VTK and consists of multiple arrays: vertex positions and values, vertex indices, cell start indices, cell types, and cell values.

Sampled cell values can be specified either per-vertex (vertex.data) or per-cell (cell.data). If both arrays are set, cell.data takes precedence.

Similar to a mesh, each cell is formed by a group of indices into the vertices. For each vertex, the corresponding (by array index) data value will be used for sampling when rendering, if specified. The index order for a tetrahedron is the same as VTK_TETRA: bottom triangle counterclockwise, then the top vertex.

For hexahedral cells, each hexahedron is formed by a group of eight indices into the vertices and data values. Vertex ordering is the same as VTK_HEXAHEDRON: four bottom vertices counterclockwise, then top four counterclockwise.

For wedge cells, each wedge is formed by a group of six indices into the vertices and data values. Vertex ordering is the same as VTK_WEDGE: three bottom vertices counterclockwise, then top three counterclockwise.

For pyramid cells, each cell is formed by a group of five indices into the vertices and data values. Vertex ordering is the same as VTK_PYRAMID: four bottom vertices counterclockwise, then the top vertex.

To maintain VTK data compatibility, the index array may be specified with cell sizes interleaved with vertex indices in the following format: $n, id_1, ..., id_n, m, id_1, ..., id_m$. This alternative index array layout can be enabled through the indexPrefixed flag (in which case, the cell.type parameter should be omitted).

A binary bounding volume hierarchy (BVH) is used internally to accelerate interval iteration. Intervals are found by intersecting BVH nodes up to a maximum level of the tree, configurable by the `maxIteratorDepth` parameter. Larger values of `maxIteratorDepth` lead to smaller individual intervals (up to leaf node intersections), yielding potentially more efficient space-skipping behavior and tighter bounds on returned interval metadata.

Unstructured volumes are created by passing the type string `"unstructured"` to `vklNewVolume`, and have the following parameters:

Table 10: Configuration parameters for unstructured (`"unstructured"`) volumes.

Type	Name	Default	Description
vec3f[]	vertex.position		[data] array of vertex positions
float[]	vertex.data		[data] array of vertex data values to be sampled
uint32[] / uint64[]	index		[data] array of indices (into the vertex array(s)) that form cells
bool	indexPrefixed	false	indicates that the `index` array is provided in a VTK-compatible format, where the indices of each cell are prefixed with the number of vertices
uint32[] / uint64[]	cell.index		[data] array of locations (into the index array), specifying the first index of each cell
float[]	cell.data		[data] array of cell data values to be sampled
uint8[]	cell.type		[data] array of cell types (VTK compatible). Supported types are:
			`VKL_TETRAHEDRON`
			`VKL_HEXAHEDRON`
			`VKL_WEDGE`
			`VKL_PYRAMID`
bool	hexIterative	false	hexahedron interpolation method, defaults to fast non-iterative version which could have rendering inaccuracies may appear if hex is not parallelepiped
bool	precomputedNormals	false	whether to accelerate by precomputing, at a cost of 12 bytes/face

The following additional parameters can be set both on `unstructured` volumes and their sampler objects (sampler object parameters default to volume parameters).

Table 11: Configuration parameters for unstructured (`"unstructured"`) volumes and their sampler objects.

Type	Name	Default	Description
int	maxIteratorDepth	6	Do not descend further than to this BVH depth during interval iteration.

VDB Volumes

VDB volumes implement a data structure that is very similar to the data structure outlined in Museth [1].

The data structure is a hierarchical regular grid at its core: Nodes are regular grids, and each grid cell may either store a constant value (this is called a tile), or child pointers.

Nodes in VDB trees are wide: Nodes on the first level have a resolution of 32^3 voxels by default, on the next level 16^3, and on the leaf level 8^3 voxels. All nodes on a given level have the same resolution. This makes it easy to find the node containing a coordinate using shift operations (cp. [1]).

VDB leaf nodes are implicit in Open VKL: they are stored as pointers to user-provided data.

[Structure of "vdb" volumes in the default configuration][imgVdbStructure]

VDB volumes interpret input data as constant cells (which are then potentially filtered). This is in contrast to structuredRegular volumes, which have a vertex-centered interpretation.

The VDB implementation in Open VKL follows the following goals:

- Efficient data structure traversal on vector architectures.
- Enable the use of industry-standard .vdb files created through the OpenVDB library.
- Compatibility with OpenVDB on a leaf data level, so that .vdb files may be loaded with minimal overhead.

VDB volumes are created by passing the type string "vdb" to vklNewVolume, and have the following parameters:

Table 12: Configuration parameters for VDB ("vdb") volumes.

Type	Name	Default	Description
float[]	indexToObject	1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0	An array of 12 values of type float that define the transformation from index space to object space. In index space, the grid is an axis-aligned regular grid, and leaf voxels have size (1,1,1). The first 9 values are interpreted as a row-major linear transformation matrix. The last 3 values are the translation of the grid origin.
uint32[]	node.format		For each input node, the data format. Currently supported are VKL_FORMAT_TILE for tiles, and VKL_FORMAT_CONSTANT_ZYX for nodes that are dense regular grids.
uint32[]	node.level		For each input node, the level on which this node exists. Tiles may exist on levels [1, VKL_VDB_NUM_LEVELS-1], all other nodes may only exist on level VKL_VDB_NUM_LEVELS-1.
vec3i[]	node.origin		For each input node, the node origin index.
VKLData[]	node.data		For each input node, the attribute data. Single-attribute volumes may have one array provided per node, while multi-attribute volumes require an array per attribute for each node. Nodes with format VKL_FORMAT_TILE are expected to have single-entry arrays per attribute. Nodes with format VKL_FORMAT_CONSTANT_ZYX are expected to have arrays with vklVdbLevelNumVoxels(level[i]) entries per attribute. Only VKL_FLOAT data is currently supported.

The level, origin, format, and data parameters must have the same size, and there must be at least one valid node or commit() will fail.

The following additional parameters can be set both on vdb volumes and their sampler objects (sampler object parameters default to volume parameters).

Table 13: Configuration parameters for VDB ("vdb") volumes and their sampler objects.

Type	Name	Default	Description
int	filter	VKL_FILTER_TRILINEAR	The filter used for reconstructing the field. Use VKLFilter for named constants.
int	gradientFilter	filter	The filter used for reconstructing the field during gradient computations. Use VKLFilter for named constants.
int	maxSamplingDepth	VKL_VDB_NUM_LEVELS-1	Do not descend further than to this depth during sampling.
int	maxIteratorDepth	VKL_VDB_NUM_LEVELS-2	Do not descend further than to this depth during iteration.

VDB sampler objects support the following observers:

Table 14: Observers supported by sampler objects created on VDB ("vdb") volumes.

Name	Buffer Type	Description
LeafNodeAccess	uint32[]	This observer returns an array with as many entries as input nodes were passed. If the input node i was accessed during traversal, then the ith entry in this array has a nonzero value. This can be used for on-demand loading of leaf nodes.

Major differences to OpenVDB

- Open VKL implements sampling in ISPC, and can exploit wide SIMD architectures.
- VDB volumes in Open VKL are read-only once committed, and designed for rendering only. Authoring or manipulating datasets is not in the scope of this implementation.
- The only supported field type is VKL_FLOAT at this point. Other field types may be supported in the future.
- The root level in Open VKL has a single node with resolution 64^3 (cp. [1]. OpenVDB uses a hash map, instead).
- Open VKL supports four-level vdb volumes. The resolution of each level can be configured at compile time using CMake variables.
  - VKL_VDB_LOG_RESOLUTION_0 sets the base 2 logarithm of the root level resolution. This variable defaults to 6, which means that the root level has a resolution of $(2^6)^3 = 64^3$.
  - VKL_VDB_LOG_RESOLUTION_1 and VKL_VDB_LOG_RESOLUTION_2 default to 5 and 4, respectively. This matches the default Open VDB resolution for inner levels.
  - VKL_VDB_LOG_RESOLUTION_3 set the base 2 logarithm of the leaf level resolution, and defaults to 3. Therefore, leaf nodes have a resolution of 8^3 voxels. Again, this matches the Open VDB default. The default settings lead to a domain resolution of $2^1 8^3 = 262144^3$ voxels.

Loading OpenVDB .vdb files

Files generated with OpenVDB can be loaded easily since Open VKL vdb volumes implement the same leaf data layout. This means that OpenVDB leaf data pointers can be passed to Open VKL using shared data buffers, avoiding copy operations.

An example of this can be found in vdb_util/include/openvkl/OpenVdbGrid.h, where the class OpenVdbFloatGrid encapsulates the necessary operations. This class is also accessible through the vklExamples application using the -file and -field command line arguments.

To use this example feature, compile Open VKL with OpenVDB_ROOT pointing to the OpenVDB prefix.

1. Museth, K. VDB: High-Resolution Sparse Volumes with Dynamic Topology. ACM Transactions on Graphics 32(3), 2013. DOI: 10.1145/2487228.2487235

Particle Volumes

Particle volumes consist of a set of points in space. Each point has a position, a radius, and a weight typically associated with an attribute. A radial basis function defines the contribution of that particle. Currently, we use the Gaussian radial basis function,

phi(P) = w * exp(-0.5 * ((P - p) / r)^2)

where P is the particle position, p is the sample position, r is the radius and w is the weight.

At each sample, the scalar field value is then computed as the sum of each radial basis function phi, for each particle that overlaps it.

The Open VKL implementation is similar to direct evaluation of samples in Reda et al.[2]. It uses an Embree-built BVH with a custom traversal, similar to the method in [1].

Similar to unstructured volumes, a binary BVH is used internally to accelerate interval iteration. Intervals are found by intersecting BVH nodes up to a maximum level of the tree, configurable by the `maxIteratorDepth` parameter.

Particle volumes are created by passing the type string `"particle"` to `vklNewVolume`, and have the following parameters:

Table 15: Configuration parameters for particle (`"particle"`) volumes.

Type	Name	Default	Description
vec3f[]	particle.position		[data] array of particle positions
float[]	particle.radius		[data] array of particle radii
float[]	particle.weight	null	[data] (optional) array of particle weights, specifying the height of the kernel.
float	radiusSupportFactor	3.0	The multipler of the particle radius required for support. Larger radii ensure smooth results at the cost of performance. In the Gaussian kernel, the the radius is one standard deviation (sigma), so a `radiusSupportFactor` of 3 corresponds to 3*sigma.
float	clampMaxCumulativeValue	0	The maximum cumulative value possible, set by user. All cumulative values will be clamped to this, and further traversal (RBF summation) of particle contributions will halt when this value is reached. A value of zero or less turns this off.
bool	estimateValueRanges	true	Enable heuristic estimation of value ranges which are used in internal acceleration structures for interval and hit iterators, as well as for determining the volume's overall value range. When set to `false`, the user *must* specify `clampMaxCumulativeValue`, and all value ranges will be assumed [0, `clampMaxCumulativeValue`]. Disabling this may improve volume commit time, but will make interval and hit iteration less efficient.

The following additional parameters can be set both on `particle` volumes and their sampler objects (sampler object parameters default to volume parameters).

Table 16: Configuration parameters for particle ("particle") volumes and their sampler objects.

Type	Name	Default	Description
int	maxIteratorDepth	6	Do not descend further than to this BVH depth during interval iteration.

1. Knoll, A., Wald, I., Navratil, P., Bowen, A., Reda, K., Papka, M.E. and Gaither, K. (2014), RBF Volume Ray Casting on Multicore and Manycore CPUs. Computer Graphics Forum, 33: 71-80. doi:10.1111/cgf.12363

2. K. Reda, A. Knoll, K. Nomura, M. E. Papka, A. E. Johnson and J. Leigh, "Visualizing large-scale atomistic simulations in ultra-resolution immersive environments," 2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), Atlanta, GA, 2013, pp. 59-65.

Sampler Objects

Computing the value of a volume at an object space coordinate is done using the sampling API, and sampler objects. Sampler objects can be created using

```
VKLSampler vklNewSampler(VKLVolume volume);
```

Sampler objects may then be parametrized with traversal parameters. Available parameters are defined by volumes, and are a subset of the volume parameters. As an example, `filter` can be set on both `vdb` volumes and their sampler objects. The volume parameter is used as the default for sampler objects. The sampler object parameter provides an override per ray. More detail on parameters can be found in the sections on volumes. Use `vklCommit()` to commit parameters to the sampler object.

Sampling

The scalar API takes a volume and coordinate, and returns a float value. NaN is returned for probe points outside the volume. The attribute index selects the scalar attribute of interest; not all volumes support multiple attributes. The time value, which must be between 0 and 1, specifies the sampling time. For temporally constant volumes, this value has no effect.

```
float vklComputeSample(VKLSampler sampler,
                       const vkl_vec3f *objectCoordinates,
                       unsigned int attributeIndex,
                       float time);
```

Vector versions allow sampling at 4, 8, or 16 positions at once. Depending on the machine type and Open VKL driver implementation, these can give greater performance. An active lane mask `valid` is passed in as an array of integers; set 0 for lanes to be ignored, -1 for active lanes. An array of time values corresponding to each object coordinate may be provided; a `NULL` value indicates all times are zero.

```
void vklComputeSample4(const int *valid,
                       VKLSampler sampler,
                       const vkl_vvec3f4 *objectCoordinates,
                       float *samples,
                       unsigned int attributeIndex,
                       const float *times);

void vklComputeSample8(const int *valid,
                       VKLSampler sampler,
                       const vkl_vvec3f8 *objectCoordinates,
                       float *samples,
```

(continues on next page)

(continued from previous page)

```
                        unsigned int attributeIndex,
                        const float *times);

void vklComputeSample16(const int *valid,
                        VKLSampler sampler,
                        const vkl_vvec3f16 *objectCoordinates,
                        float *samples,
                        unsigned int attributeIndex,
                        const float *times);
```

A stream version allows sampling an arbitrary number of positions at once. While the vector version requires coordinates to be provided in a structure-of-arrays layout, the stream version allows coordinates to be provided in an array-of-structures layout. Thus, the stream API can be used to avoid reformatting of data by the application. As with the vector versions, the stream API can give greater performance than the scalar API.

```
void vklComputeSampleN(VKLSampler sampler,
                       unsigned int N,
                       const vkl_vec3f *objectCoordinates,
                       float *samples,
                       unsigned int attributeIndex,
                       const float *times);
```

All of the above sampling APIs can be used, regardless of the driver's native SIMD width.

Sampling Multiple Attributes

Open VKL provides additional APIs for sampling multiple scalar attributes in a single call through the `vklComputeSampleM*()` interfaces. Beyond convenience, these can give improved performance relative to the single attribute sampling APIs. As with the single attribute APIs, sampling time values may be specified; note that these are provided per object coordinate only (rather than separately per attribute).

A scalar API supports sampling `M` attributes specified by `attributeIndices` on a single object space coordinate:

```
void vklComputeSampleM(VKLSampler sampler,
                       const vkl_vec3f *objectCoordinates,
                       float *samples,
                       unsigned int M,
                       const unsigned int *attributeIndices,
                       float time);
```

Vector versions allow sampling at 4, 8, or 16 positions at once across the `M` attributes:

```
void vklComputeSampleM4(const int *valid,
                        VKLSampler sampler,
                        const vkl_vvec3f4 *objectCoordinates,
                        float *samples,
                        unsigned int M,
                        const unsigned int *attributeIndices,
                        const float *times);

void vklComputeSampleM8(const int *valid,
                        VKLSampler sampler,
                        const vkl_vvec3f8 *objectCoordinates,
                        float *samples,
                        unsigned int M,
```

(continues on next page)

(continued from previous page)

```
                          const unsigned int *attributeIndices,
                          const float *times);

void vklComputeSampleM16(const int *valid,
                         VKLSampler sampler,
                         const vkl_vvec3f16 *objectCoordinates,
                         float *samples,
                         unsigned int M,
                         const unsigned int *attributeIndices,
                         const float *times);
```

The `[4, 8, 16] * M` sampled values are populated in the `samples` array in a structure-of-arrays layout, with all values for each attribute provided in sequence. That is, sample values `s_m,n` for the `m`th attribute and `n`th object coordinate will be populated as

```
samples = [s_0,0,   s_0,1,   ..., s_0,N-1,
           s_1,0,   s_1,1,   ..., s_1,N-1,
           ...,
           s_M-1,0, s_M-1,1, ..., s_M-1,N-1]
```

A stream version allows sampling an arbitrary number of positions at once across the `M` attributes. As with single attribute stream sampling, the `N` coordinates are provided in an array-of-structures layout.

```
void vklComputeSampleMN(VKLSampler sampler,
                        unsigned int N,
                        const vkl_vec3f *objectCoordinates,
                        float *samples,
                        unsigned int M,
                        const unsigned int *attributeIndices,
                        const float *times);
```

The `M * N` sampled values are populated in the `samples` array in an array-of-structures layout, with all attribute values for each coordinate provided in sequence as

```
samples = [s_0,0,   s_1,0,   ..., s_M-1,0,
           s_0,1,   s_1,1,   ..., s_M-1,1,
           ...,
           s_0,N-1, s_1,N-1, ..., s_M-1,N-1]
```

All of the above sampling APIs can be used, regardless of the driver's native SIMD width.

Gradients

In a very similar API to `vklComputeSample`, `vklComputeGradient` queries the value gradient at an object space coordinate. Again, a scalar API, now returning a vec3f instead of a float. NaN values are returned for points outside the volume. The time value, which must be between 0 and 1, specifies the sampling time. For temporally constant volumes, this value has no effect.

```
vkl_vec3f vklComputeGradient(VKLSampler sampler,
                             const vkl_vec3f *objectCoordinates,
                             unsigned int attributeIndex,
                             float time);
```

Vector versions are also provided:

```
void vklComputeGradient4(const int *valid,
                         VKLSampler sampler,
                         const vkl_vvec3f4 *objectCoordinates,
                         vkl_vvec3f4 *gradients,
                         unsigned int attributeIndex,
                         const float *times);

void vklComputeGradient8(const int *valid,
                         VKLSampler sampler,
                         const vkl_vvec3f8 *objectCoordinates,
                         vkl_vvec3f8 *gradients,
                         unsigned int attributeIndex,
                         const float *times);

void vklComputeGradient16(const int *valid,
                          VKLSampler sampler,
                          const vkl_vvec3f16 *objectCoordinates,
                          vkl_vvec3f16 *gradients,
                          unsigned int attributeIndex,
                          const float *times);
```

Finally, a stream version is provided:

```
void vklComputeGradientN(VKLSampler sampler,
                         unsigned int N,
                         const vkl_vec3f *objectCoordinates,
                         vkl_vec3f *gradients,
                         unsigned int attributeIndex,
                         const float *times);
```

All of the above gradient APIs can be used, regardless of the driver's native SIMD width.

Iterators

Open VKL has APIs to search for particular volume values along a ray. Queries can be for ranges of volume values (`vklIterateInterval`) or for particular values (`vklIterateHit`). Only the first volume attribute is currently considered in the iterator APIs.

The desired values are set in a `VKLValueSelector`, which needs to be created, filled in with values, and then committed.

```
VKLValueSelector vklNewValueSelector(VKLVolume volume);

void vklValueSelectorSetRanges(VKLValueSelector valueSelector,
                               size_t numRanges,
                               const vkl_range1f *ranges);

void vklValueSelectorSetValues(VKLValueSelector valueSelector,
                               size_t numValues,
                               const float *values);
```

To query an interval, a `VKLIntervalIterator` of scalar or vector width must be initialized with `vklInitIntervalIterator`.

```
VKLIntervalIterator vklInitIntervalIterator(VKLSampler sampler,
                                            const vkl_vec3f *origin,
```

(continues on next page)

(continued from previous page)

```
                                                const vkl_vec3f *direction,
                                                const vkl_range1f *tRange,
                                                VKLValueSelector valueSelector,
                                                void *buffer);

VKLIntervalIterator4 vklInitIntervalIterator4(const int *valid,
                                              VKLSampler sampler,
                                              const vkl_vvec3f4 *origin,
                                              const vkl_vvec3f4 *direction,
                                              const vkl_vrange1f4 *tRange,
                                              VKLValueSelector valueSelector,
                                              void *buffer);

VKLIntervalIterator8 vklInitIntervalIterator8(const int *valid,
                                              VKLSampler sampler,
                                              const vkl_vvec3f8 *origin,
                                              const vkl_vvec3f8 *direction,
                                              const vkl_vrange1f8 *tRange,
                                              VKLValueSelector valueSelector,
                                              void *buffer);

VKLIntervalIterator16 vklInitIntervalIterator16(const int *valid,
                                                VKLSampler sampler,
                                                const vkl_vvec3f16 *origin,
                                                const vkl_vvec3f16 *direction,
                                                const vkl_vrange1f16 *tRange,
                                                VKLValueSelector valueSelector,
                                                void *buffer);
```

Open VKL places the iterator struct into a user-provided buffer, and the returned handle is essentially a pointer into this buffer. This means that the iterator handle must not be used after the buffer ceases to exist. Copying iterator buffers is currently not supported.

The required size, in bytes, of the buffer can be queried with

```
size_t vklGetIntervalIteratorSize(VKLSampler sampler);

size_t vklGetIntervalIteratorSize4(VKLSampler sampler);

size_t vklGetIntervalIteratorSize8(VKLSampler sampler);

size_t vklGetIntervalIteratorSize16(VKLSampler sampler);
```

The values these functions return may change depending on the parameters set on `sampler`.

Open VKL also provides a conservative maximum size over all volume types as a preprocessor definition (`VKL_MAX_INTERVAL_ITERATOR_SIZE`). This is particularly useful for stack-based allocation in ISPC. Open VKL will attempt to detect the native vector width using `TARGET_WIDTH`, which is defined in recent versions of ISPC.

Intervals can then be processed by calling `vklIterateInterval` as long as the returned lane masks indicates that the iterator is still within the volume:

```
int vklIterateInterval(VKLIntervalIterator iterator,
                       VKLInterval *interval);

void vklIterateInterval4(const int *valid,
```

(continues on next page)

(continued from previous page)

```
                          VKLIntervalIterator4 iterator,
                          VKLInterval4 *interval,
                          int *result);

void vklIterateInterval8(const int *valid,
                          VKLIntervalIterator8 iterator,
                          VKLInterval8 *interval,
                          int *result);

void vklIterateInterval16(const int *valid,
                           VKLIntervalIterator16 iterator,
                           VKLInterval16 *interval,
                           int *result);
```

The intervals returned have a t-value range, a value range, and a `nominalDeltaT` which is approximately the step size (in units of ray direction) that should be used to walk through the interval, if desired. The number and length of intervals returned is volume type implementation dependent. There is currently no way of requesting a particular splitting.

```
typedef struct
{
  vkl_range1f tRange;
  vkl_range1f valueRange;
  float nominalDeltaT;
} VKLInterval;

typedef struct
{
  vkl_vrange1f4 tRange;
  vkl_vrange1f4 valueRange;
  float nominalDeltaT[4];
} VKLInterval4;

typedef struct
{
  vkl_vrange1f8 tRange;
  vkl_vrange1f8 valueRange;
  float nominalDeltaT[8];
} VKLInterval8;

typedef struct
{
  vkl_vrange1f16 tRange;
  vkl_vrange1f16 valueRange;
  float nominalDeltaT[16];
} VKLInterval16;
```

Querying for particular values is done using a `VKLHitIterator` in much the same fashion. This API could be used, for example, to find isosurfaces. In contrast to interval iterators, time value(s) may be provided to specify the sampling time. These values must be between 0 and 1; for the vector versions, a `NULL` value indicates all times are zero. For temporally constant volumes, the time values have no effect. Again, a user allocated buffer must be provided, and a `VKLHitIterator` of the desired width must be initialized:

```
VKLHitIterator vklInitHitIterator(VKLSampler sampler,
                                  const vkl_vec3f *origin,
                                  const vkl_vec3f *direction,
```

(continues on next page)

(continued from previous page)

```
                                    const vkl_range1f *tRange,
                                    float time,
                                    VKLValueSelector valueSelector,
                                    void *buffer);

VKLHitIterator4 vklInitHitIterator4(const int *valid,
                            VKLSampler sampler,
                            const vkl_vvec3f4 *origin,
                            const vkl_vvec3f4 *direction,
                            const vkl_vrange1f4 *tRange,
                            const float *times,
                            VKLValueSelector valueSelector,
                            void *buffer);

VKLHitIterator8 vklInitHitIterator8(const int *valid,
                            VKLSampler sampler,
                            const vkl_vvec3f8 *origin,
                            const vkl_vvec3f8 *direction,
                            const vkl_vrange1f8 *tRange,
                            const float *times,
                            VKLValueSelector valueSelector,
                            void *buffer);

VKLHitIterator16 vklInitHitIterator16(const int *valid,
                             VKLSampler sampler,
                             const vkl_vvec3f16 *origin,
                             const vkl_vvec3f16 *direction,
                             const vkl_vrange1f16 *tRange,
                             const float *times,
                             VKLValueSelector valueSelector,
                             void *buffer);
```

Buffer size can be queried with

```
size_t vklGetHitIteratorSize(VKLSampler sampler);

size_t vklGetHitIteratorSize4(VKLSampler sampler);

size_t vklGetHitIteratorSize8(VKLSampler sampler);

size_t vklGetHitIteratorSize16(VKLSampler sampler);
```

Open VKL also provides the macro `VKL_MAX_HIT_ITERATOR_SIZE` as a conservative estimate.

Hits are then queried by looping a call to `vklIterateHit` as long as the returned lane mask indicates that the iterator is still within the volume.

```
int vklIterateHit(VKLHitIterator iterator, VKLHit *hit);

void vklIterateHit4(const int *valid,
                    VKLHitIterator4 iterator,
                    VKLHit4 *hit,
                    int *result);

void vklIterateHit8(const int *valid,
                    VKLHitIterator8 iterator,
                    VKLHit8 *hit,
```

(continues on next page)

(continued from previous page)

```
                     int *result);

void vklIterateHit16(const int *valid,
                     VKLHitIterator16 iterator,
                     VKLHit16 *hit,
                     int *result);
```

Returned hits consist of a t-value, a volume value (equal to one of the requested values specified in the value selector), and an (object space) epsilon value estimating the error of the intersection:

```
typedef struct
{
  float t;
  float sample;
  float epsilon;
} VKLHit;

typedef struct
{
  float t[4];
  float sample[4];
  float epsilon[4];
} VKLHit4;

typedef struct
{
  float t[8];
  float sample[8];
  float epsilon[8];
} VKLHit8;

typedef struct
{
  float t[16];
  float sample[16];
  float epsilon[16];
} VKLHit16;
```

For both interval and hit iterators, only the vector-wide API for the native SIMD width (determined via `vklGetNativeSIMDWidth` can be called. The scalar versions are always valid. This restriction will likely be lifted in the future.

Performance Recommendations

MXCSR control and status register

It is strongly recommended to have the `Flush to Zero` and `Denormals are Zero` mode of the MXCSR control and status register enabled for each thread before calling the sampling, gradient, or interval API functions. Otherwise, under some circumstances special handling of denormalized floating point numbers can significantly reduce application and Open VKL performance. The driver parameter `flushDenormals` or environment variable `OPENVKL_FLUSH_DENORMALS` can be used to toggle this mode; by default it is enabled. Alternatively, when using Open VKL together with the Intel® Threading Building Blocks, it is sufficient to execute the following code at the beginning of the application main thread (before the creation of the `tbb::task_scheduler_init` object):

```
#include <xmmintrin.h>
#include <pmmintrin.h>
...
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
```

If using a different tasking system, make sure each thread calling into Open VKL has the proper mode set.

Iterator Allocation

`vklInitIntervalIterator` and `vklInitHitIterator` expect a user allocated buffer. While this buffer can be allocated by any means, we expect iterators to be used in inner loops and advise against heap allocation in that case. Applications may provide high performance memory pools, but as a preferred alternative we recommend stack allocated buffers.

In C99, variable length arrays provide an easy way to achieve this:

```
const size_t bufferSize = vklGetIntervalIteratorSize(sampler);
char buffer[bufferSize];
```

Note that the call to `vklGetIntervalIteratorSize` or `vklGetHitIteratorSize` should not appear in an inner loop as it is relatively costly. The return value depends on the volume type, target architecture, and parameters to `sampler`.

In C++, variable length arrays are not part of the standard. Here, users may rely on `alloca` and similar functions:

```
#include <alloca.h>
const size_t bufferSize = vklGetIntervalIteratorSize(sampler);
char *buffer = alloca(bufferSize);
```

Users should understand the implications of `alloca`. In particular, `alloca` does check available stack space and may result in stack overflow. `buffer` also becomes invalid at the end of the scope. As one consequence, it cannot be returned from a function. On Windows, `_malloca` is a safer option that performs additional error checking, but requires the use of `_freea`.

In ISPC, variable length or `alloca` do not exist. Applications may instead rely on the `VKL_MAX_INTERVAL_ITERATOR_SIZE` and `VKL_MAX_HIT_ITERATOR_SIZE` macros:

```
uniform unsigned int8 buffer[VKL_MAX_INTERVAL_ITERATOR_SIZE];
```

These values are majorants over all drivers and volume types. Note that Open VKL attempts to detect the target SIMD width using `TARGET_WIDTH`, returning smaller buffer sizes for narrow architectures. However, Open VKL may fall back to the largest buffer size over all targets.

Multi-attribute Volume Data Layout

Open VKL provides flexible managed data APIs that allow applications to specify input data in various formats and layouts. When shared buffers are used (`dataCreationFlags = VKL_DATA_SHARED_BUFFER`), Open VKL will use the application-owned memory directly, respecting the input data layout. Shared buffers therefore allow applications to strategically select the best layout for multi-attribute volume data and expected sampling behavior.

For volume attributes that are sampled individually (e.g. using `vklComputeSample[4,8,16,N]()`), it is recommended to use a structure-of-arrays layout. That is, each attribute's data should be compact in contiguous memory. This can be accomplished by simply using Open VKL owned data objects (`dataCreationFlags = VKL_DATA_DEFAULT`), or by using a natural `byteStride` for shared buffers.

For volume attributes that are sampled simultaneously (e.g. using `vklComputeSampleM[4,8,16,N]()`), it is recommended to use an array-of-structures layout. That is, data for these attributes should be provided per voxel in a contiguous layout. This is accomplished using shared buffers for each attribute with appropriate byte strides. For example, for a three attribute structured volume representing a velocity field, the data can be provided as:

```
// used in Open VKL shared buffers, so must not be freed by application
std::vector<vkl_vec3f> velocities(numVoxels);

for (auto &v : velocities) {
  v.x = ...;
  v.y = ...;
  v.z = ...;
}

std::vector<VKLData> attributes;

attributes.push_back(vklNewData(velocities.size(),
                                VKL_FLOAT,
                                &velocities[0].x,
                                VKL_DATA_SHARED_BUFFER,
                                sizeof(vkl_vec3f)));

attributes.push_back(vklNewData(velocities.size(),
                                VKL_FLOAT,
                                &velocities[0].y,
                                VKL_DATA_SHARED_BUFFER,
                                sizeof(vkl_vec3f)));

attributes.push_back(vklNewData(velocities.size(),
                                VKL_FLOAT,
                                &velocities[0].z,
                                VKL_DATA_SHARED_BUFFER,
                                sizeof(vkl_vec3f)));

VKLData attributesData =
    vklNewData(attributes.size(), VKL_DATA, attributes.data());

for (auto &attribute : attributes)
  vklRelease(attribute);

VKLVolume volume = vklNewVolume("structuredRegular");

vklSetData(volume, "data", attributesData);
vklRelease(attributesData);

// set other volume parameters...

vklCommit(volume);
```

These are general recommendations for common scenarios; it is still recommended to evaluate performance of different volume data layouts for your application's particular use case.

Open Image Denoise

The purpose of Open Image Denoise is to provide an open, high-quality, efficient, and easy-to-use denoising library that allows one to significantly reduce rendering times in ray tracing based rendering applications.

Introduction

The purpose of Open Image Denoise is to provide an open, high-quality, efficient, and easy-to-use denoising library that allows one to significantly reduce rendering times in ray tracing based rendering applications. It filters out the Monte Carlo noise inherent to stochastic ray tracing methods like path tracing, reducing the amount of necessary samples per pixel by even multiple orders of magnitude (depending on the desired closeness to the ground truth). A simple but flexible C/C++ API ensures that the library can be easily integrated into most existing or new rendering solutions.

At the heart of the Open Image Denoise library is a collection of efficient deep learning based denoising filters, which were trained to handle a wide range of samples per pixel (spp), from 1 spp to almost fully converged. Thus it is suitable for both preview and final-frame rendering. The filters can denoise images either using only the noisy color (beauty) buffer, or, to preserve as much detail as possible, can optionally utilize auxiliary feature buffers as well (e.g. albedo, normal). Such buffers are supported by most renderers as arbitrary output variables (AOVs) or can be usually implemented with little effort.

Although the library ships with a set of pre-trained filter models, it is not mandatory to use these. To optimize a filter for a specific renderer, sample count, content type, scene, etc., it is possible to train the model using the included training toolkit and user-provided image datasets.

Intel Open Image Denoise API

Intel Open Image Denoise provides a C99 API (also compatible with C++) and a C++11 wrapper API as well. For simplicity, this document mostly refers to the C99 version of the API.

The API is designed in an object-oriented manner, e.g. it contains device objects (`OIDNDevice` type), buffer objects (`OIDNBuffer` type), and filter objects (`OIDNFilter` type). All objects are reference-counted, and handles can be released by calling the appropriate release function (e.g. `oidnReleaseDevice`) or retained by incrementing the reference count (e.g. `oidnRetainDevice`).

An important aspect of objects is that setting their parameters do not have an immediate effect (with a few exceptions). Instead, objects with updated parameters are in an unusable state until the parameters get explicitly committed to a given object. The commit semantic allows for batching up multiple small changes, and specifies exactly when changes to objects will occur.

All API calls are thread-safe, but operations that use the same device will be serialized, so the amount of API calls from different threads should be minimized.

To have a quick overview of the C99 and C++11 APIs, see the following simple example code snippets.

C99 API Example

```
#include <OpenImageDenoise/oidn.h>
...
// Create an Intel Open Image Denoise device
OIDNDevice device = oidnNewDevice(OIDN_DEVICE_TYPE_DEFAULT);
oidnCommitDevice(device);

// Create a denoising filter
OIDNFilter filter = oidnNewFilter(device, "RT"); // generic ray tracing filter
oidnSetSharedFilterImage(filter, "color",  colorPtr,
```

(continues on next page)

(continued from previous page)

```
                          OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0);
oidnSetSharedFilterImage(filter, "albedo", albedoPtr,
                          OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0); // optional
oidnSetSharedFilterImage(filter, "normal", normalPtr,
                          OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0); // optional
oidnSetSharedFilterImage(filter, "output", outputPtr,
                          OIDN_FORMAT_FLOAT3, width, height, 0, 0, 0);
oidnSetFilter1b(filter, "hdr", true); // image is HDR
oidnCommitFilter(filter);

// Filter the image
oidnExecuteFilter(filter);

// Check for errors
const char* errorMessage;
if (oidnGetDeviceError(device, &errorMessage) != OIDN_ERROR_NONE)
  printf("Error: %s\n", errorMessage);

// Cleanup
oidnReleaseFilter(filter);
oidnReleaseDevice(device);
```

C++11 API Example

```
#include <OpenImageDenoise/oidn.hpp>
...
// Create an Intel Open Image Denoise device
oidn::DeviceRef device = oidn::newDevice();
device.commit();

// Create a denoising filter
oidn::FilterRef filter = device.newFilter("RT"); // generic ray tracing filter
filter.setImage("color",  colorPtr,  oidn::Format::Float3, width, height);
filter.setImage("albedo", albedoPtr, oidn::Format::Float3, width, height); // optional
filter.setImage("normal", normalPtr, oidn::Format::Float3, width, height); // optional
filter.setImage("output", outputPtr, oidn::Format::Float3, width, height);
filter.set("hdr", true); // image is HDR
filter.commit();

// Filter the image
filter.execute();

// Check for errors
const char* errorMessage;
if (device.getError(errorMessage) != oidn::Error::None)
  std::cout << "Error: " << errorMessage << std::endl;
```

Device

Intel Open Image Denoise supports a device concept, which allows different components of the application to use the Open Image Denoise API without interfering with each other. An application first needs to create a device with

```
OIDNDevice oidnNewDevice(OIDNDeviceType type);
```

where the `type` enumeration maps to a specific device implementation, which can be one of the following:

Table 17: Supported device types, i.e., valid constants of type `OIDNDeviceType`.

Name	Description
OIDN_DEVICE_TYPE_DEFAULT	select the approximately fastest device
OIDN_DEVICE_TYPE_CPU	CPU device (requires SSE4.1 support)

Once a device is created, you can call

```
void oidnSetDevice1b(OIDNDevice device, const char* name, bool value);
void oidnSetDevice1i(OIDNDevice device, const char* name, int  value);
bool oidnGetDevice1b(OIDNDevice device, const char* name);
int  oidnGetDevice1i(OIDNDevice device, const char* name);
```

to set and get parameter values on the device. Note that some parameters are constants, thus trying to set them is an error. See the tables below for the parameters supported by devices.

Table 18: Parameters supported by all devices.

Type	Name	Default	Description
const int	version		combined version number (major.minor.patch) with two decimal digits per component
const int	versionMajor		major version number
const int	versionMinor		minor version number
const int	versionPatch		patch version number
int	verbose	0	verbosity level of the console output between 0–4; when set to 0, no output is printed, when set to a higher level more output is printed

Table 19: Additional parameters supported only by CPU devices.

Type	Name	Default	Description
int	numThreads	0	maximum number of threads which the library should use; 0 will set it automatically to get the best performance
bool	setAffinity	true	bind software threads to hardware threads if set to true (improves performance); false disables binding

Note that the CPU device heavily relies on setting the thread affinities to achieve optimal performance, so it is highly recommended to leave this option enabled. However, this may interfere with the application if that also sets the thread affinities, potentially causing performance degradation. In such cases, the recommended solution is to either disable setting the affinities in the application or in Intel Open Image Denoise, or to always set/reset

the affinities before/after each parallel region in the application (e.g., if using TBB, with `tbb::task_arena` and `tbb::task_scheduler_observer`).

Once parameters are set on the created device, the device must be committed with

```
void oidnCommitDevice(OIDNDevice device);
```

This device can then be used to construct further objects, such as buffers and filters. Note that a device can be committed only once during its lifetime. Before the application exits, it should release all devices by invoking

```
void oidnReleaseDevice(OIDNDevice device);
```

Note that Intel Open Image Denoise uses reference counting for all object types, so this function decreases the reference count of the device, and if the count reaches 0 the device will automatically get deleted. It is also possible to increase the reference count by calling

```
void oidnRetainDevice(OIDNDevice device);
```

An application typically creates only a single device. If required differently, it should only use a small number of devices at any given time.

Error Handling

Each user thread has its own error code per device. If an error occurs when calling an API function, this error code is set to the occurred error if it stores no previous error. The currently stored error can be queried by the application via

```
OIDNError oidnGetDeviceError(OIDNDevice device, const char** outMessage);
```

where `outMessage` can be a pointer to a C string which will be set to a more descriptive error message, or it can be `NULL`. This function also clears the error code, which assures that the returned error code is always the first error occurred since the last invocation of `oidnGetDeviceError` on the current thread. Note that the optionally returned error message string is valid only until the next invocation of the function.

Alternatively, the application can also register a callback function of type

```
typedef void (*OIDNErrorFunction)(void* userPtr, OIDNError code, const char* message);
```

via

```
void oidnSetDeviceErrorFunction(OIDNDevice device, OIDNErrorFunction func, void*␣
↪userPtr);
```

to get notified when errors occur. Only a single callback function can be registered per device, and further invocations overwrite the previously set callback function, which do *not* require also calling the `oidnCommitDevice` function. Passing `NULL` as function pointer disables the registered callback function. When the registered callback function is invoked, it gets passed the user-defined payload (`userPtr` argument as specified at registration time), the error code (`code` argument) of the occurred error, as well as a string (`message` argument) that further describes the error. The error code is always set even if an error callback function is registered. It is recommended to always set a error callback function, to detect all errors.

When the device construction fails, `oidnNewDevice` returns `NULL` as device. To detect the error code of a such failed device construction, pass `NULL` as device to the `oidnGetDeviceError` function. For all other invocations of `oidnGetDeviceError`, a proper device handle must be specified.

The following errors are currently used by Intel Open Image Denoise:

Table 20: Possible error codes, i.e., valid constants of type `OIDNError`.

Name	Description
OIDN_ERROR_NONE	no error occurred
OIDN_ERROR_UNKNOWN	an unknown error occurred
OIDN_ERROR_INVALID_ARGUMENT	an invalid argument was specified
OIDN_ERROR_INVALID_OPERATION	the operation is not allowed
OIDN_ERROR_OUT_OF_MEMORY	not enough memory to execute the operation
OIDN_ERROR_UNSUPPORTED_HARDWARE	the hardware (e.g., CPU) is not supported
OIDN_ERROR_CANCELLED	the operation was cancelled by the user

Buffer

Large data like images can be passed to Intel Open Image Denoise either via pointers to memory allocated and managed by the user (this is the recommended, often easier and more efficient approach, if supported by the device) or by creating buffer objects (supported by all devices). To create a new data buffer with memory allocated and owned by the device, holding `byteSize` number of bytes, use

```
OIDNBuffer oidnNewBuffer(OIDNDevice device, size_t byteSize);
```

The created buffer is bound to the specified device (`device` argument). The specified number of bytes are allocated at buffer construction time and deallocated when the buffer is destroyed.

It is also possible to create a "shared" data buffer with memory allocated and managed by the user with

```
OIDNBuffer oidnNewSharedBuffer(OIDNDevice device, void* ptr, size_t byteSize);
```

where `ptr` points to the user-managed memory and `byteSize` is its size in bytes. At buffer construction time no buffer data is allocated, but the buffer data provided by the user is used. The buffer data must remain valid for as long as the buffer may be used, and the user is responsible to free the buffer data when no longer required.

Similar to device objects, buffer objects are also reference-counted and can be retained and released by calling the following functions:

```
void oidnRetainBuffer(OIDNBuffer buffer);
void oidnReleaseBuffer(OIDNBuffer buffer);
```

Accessing the data stored in a buffer object is possible by mapping it into the address space of the application using

```
void* oidnMapBuffer(OIDNBuffer buffer, OIDNAccess access, size_t byteOffset, size_t␣
↪byteSize)
```

where `access` is the desired access mode of the mapped memory, `byteOffset` is the offset to the beginning of the mapped memory region in bytes, and `byteSize` is the number of bytes to map. The function returns a pointer to the mapped buffer data. If the specified `byteSize` is 0, the maximum available amount of memory will be mapped. The `access` argument must be one of the access modes in the following table:

Table 21: Access modes for memory regions mapped with `oidnMapBuffer`, i.e., valid constants of type `OIDNAccess`.

Name	Description
OIDN_ACCESS_READ	read-only access
OIDN_ACCESS_WRITE	write-only access
OIDN_ACCESS_READ_WRITE	read and write access
OIDN_ACCESS_WRITE_DISCARD	write-only access but the previous contents will be discarded

After accessing the mapped data in the buffer, the memory region must be unmapped with

```
void oidnUnmapBuffer(OIDNBuffer buffer, void* mappedPtr);
```

where `mappedPtr` must be a pointer returned by a call to `oidnMapBuffer` for the specified buffer. Any change to the mapped data is guaranteed to take effect only after unmapping the memory region.

Data Format

Buffers store opaque data and thus have no information about the type and format of the data. Other objects, e.g. filters, typically require specifying the format of the data stored in buffers or shared via pointers. This can be done using the `OIDNFormat` enumeration type:

Table 22: Supported data formats, i.e., valid constants of type `OIDNFormat`.

Name	Description
OIDN_FORMAT_UNDEFINED	undefined format
OIDN_FORMAT_FLOAT	32-bit single-precision floating point scalar
OIDN_FORMAT_FLOAT[234]	... and [234]-element vector

Filter

Filters are the main objects in Intel Open Image Denoise that are responsible for the actual denoising. The library ships with a collection of filters which are optimized for different types of images and use cases. To create a filter object, call

```
OIDNFilter oidnNewFilter(OIDNDevice device, const char* type);
```

where `type` is the name of the filter type to create. The supported filter types are documented later in this section. Once created, filter objects can be retained and released with

```
void oidnRetainFilter(OIDNFilter filter);
void oidnReleaseFilter(OIDNFilter filter);
```

After creating a filter, it needs to be set up by specifying the input and output images, and potentially setting other parameter values as well.

To bind images to the filter, you can use one of the following functions:

```
void oidnSetFilterImage(OIDNFilter filter, const char* name,
                        OIDNBuffer buffer, OIDNFormat format,
                        size_t width, size_t height,
                        size_t byteOffset,
                        size_t bytePixelStride, size_t byteRowStride);

void oidnSetSharedFilterImage(OIDNFilter filter, const char* name,
                              void* ptr, OIDNFormat format,
                              size_t width, size_t height,
                              size_t byteOffset,
                              size_t bytePixelStride, size_t byteRowStride);
```

It is possible to specify either a data buffer object (`buffer` argument) with the `oidnSetFilterImage` function, or directly a pointer to shared user-managed data (`ptr` argument) with the `oidnSetSharedFilterImage` function.

In both cases, you must also specify the name of the image parameter to set (`name` argument, e.g. `"color"`, `"output"`), the pixel format (`format` argument), the width and height of the image in number of pixels (`width` and `height` arguments), the starting offset of the image data (`byteOffset` argument), the pixel stride (`bytePixelStride` argument) and the row stride (`byteRowStride` argument), in number of bytes. Note that the row stride must be an integer multiple of the pixel stride.

If the pixels and/or rows are stored contiguously (tightly packed without any gaps), you can set `bytePixelStride` and/or `byteRowStride` to 0 to let the library compute the actual strides automatically, as a convenience.

Some special data used by filters are opaque/untyped (e.g. trained model weights blobs), which can be specified with the `oidnSetSharedFilterData` function:

```
void oidnSetSharedFilterData(OIDNFilter filter, const char* name,
                             void* ptr, size_t byteSize);
```

Filters may have parameters other than buffers as well, which you can set and get using the following functions:

```
void  oidnSetFilter1b(OIDNFilter filter, const char* name, bool  value);
void  oidnSetFilter1i(OIDNFilter filter, const char* name, int   value);
void  oidnSetFilter1f(OIDNFilter filter, const char* name, float value);
bool  oidnGetFilter1b(OIDNFilter filter, const char* name);
int   oidnGetFilter1i(OIDNFilter filter, const char* name);
float oidnGetFilter1f(OIDNFilter filter, const char* name);
```

Filters support a progress monitor callback mechanism that can be used to report progress of filter operations and to cancel them as well. Calling `oidnSetFilterProgressMonitorFunction` registers a progress monitor callback function (`func` argument) with payload (`userPtr` argument) for the specified filter (`filter` argument):

```
typedef bool (*OIDNProgressMonitorFunction)(void* userPtr, double n);

void oidnSetFilterProgressMonitorFunction(OIDNFilter filter,
                                          OIDNProgressMonitorFunction func,
                                          void* userPtr);
```

Only a single callback function can be registered per filter, and further invocations overwrite the previously set callback function. Passing `NULL` as function pointer disables the registered callback function. Once registered, Intel Open Image Denoise will invoke the callback function multiple times during filter operations, by passing the payload as set at registration time (`userPtr` argument), and a `double` in the range [0, 1] which estimates the progress of the operation (`n` argument). When returning `true` from the callback function, Intel Open Image Denoise will continue the filter operation normally. When returning `false`, the library will cancel the filter operation with the `OIDN_ERROR_CANCELLED` error code.

After setting all necessary parameters for the filter, the changes must be commmitted by calling

```
void oidnCommitFilter(OIDNFilter filter);
```

The parameters can be updated after committing the filter, but it must be re-committed for the changes to take effect.

Finally, an image can be filtered by executing the filter with

```
void oidnExecuteFilter(OIDNFilter filter);
```

which will read the input image data from the specified buffers and produce the denoised output image.

In the following we describe the different filters that are currently implemented in Intel Open Image Denoise.

RT

The `RT` (**r**ay **t**racing) filter is a generic ray tracing denoising filter which is suitable for denoising images rendered with Monte Carlo ray tracing methods like unidirectional and bidirectional path tracing. It supports depth of field and motion blur as well, but it is *not* temporally stable. The filter is based on a convolutional neural network (CNN), and it aims to provide a good balance between denoising performance and quality. The filter comes with a set of pre-trained CNN models that work well with a wide range of ray tracing based renderers and noise levels.

It accepts either a low dynamic range (LDR) or high dynamic range (HDR) color image as input. Optionally, it also accepts auxiliary *feature* images, e.g. albedo and normal, which improve the denoising quality, preserving more details in the image.

The `RT` filter has certain limitations regarding the supported input images. Most notably, it cannot denoise images that were not rendered with ray tracing. Another important limitation is related to anti-aliasing filters. Most renderers use a high-quality pixel reconstruction filter instead of a trivial box filter to minimize aliasing artifacts (e.g. Gaussian, Blackman-Harris). The `RT` filter does support such pixel filters but only if implemented with importance sampling. Weighted pixel sampling (sometimes called *splatting*) introduces correlation between neighboring pixels, which causes the denoising to fail (the noise will not be filtered), thus it is not supported.

The filter can be created by passing `"RT"` to the `oidnNewFilter` function as the filter type. The filter supports the parameters listed in the table below. All specified images must have the same dimensions. The output image can be one of the input images (i.e. in-place denoising is supported).

Table 23: Parameters supported by the `RT` filter.

Type	Format	Name	Default	Description
Image	float3	color		input color image (LDR values in [0, 1] or HDR values in [0, $+\infty$), 3 channels)
Image	float3	albedo		input feature image containing the albedo (values in [0, 1], 3 channels) of the first hit per pixel; *optional*
Image	float3	normal		input feature image containing the shading normal (world-space or view-space, arbitrary length, values in ($-\infty$, $+\infty$), 3 channels) of the first hit per pixel; *optional*, requires setting the albedo image too
Image	float3	output		output color image (3 channels); it can be one of the input images
Data		weights		trained model weights blob; *optional*
bool		hdr	false	whether the color is HDR
float		hdrScale	NaN	HDR color values are interpreted such that, multiplied by this scale, a value of 1 corresponds to a luminance level of 100 cd/m^2 (this affects the quality of the output but the output color values will *not* be scaled); if set to NaN, the scale is computed automatically (*default*)
bool		srgb	false	whether the color is encoded with the sRGB (or 2.2 gamma) curve (LDR only) or is linear; the output will be encoded with the same curve
int		maxMemoryMB	6000	approximate maximum amount of scratch memory to use in megabytes (actual memory usage may be higher); limiting memory usage may cause slower denoising due to internally splitting the image into overlapping tiles, but cannot cause the denoising to fail
const int		alignment		when manually denoising the image in tiles, the tile size and offsets should be multiples of this amount of pixels to avoid artifacts; note that manual tiled denoising of HDR images is supported *only* when hdrScale is set by the user
const int		overlap		when manually denoising the image in tiles, the tiles should overlap by this amount of pixels

[Example noisy color image rendered using unidirectional path tracing (64 spp). *Scene by Evermo-*

tion.][imgMazdaColor]

[Example output image denoised using color and auxiliary feature images (albedo and normal).][imgMazdaDenoised]

Using auxiliary feature images like albedo and normal helps preserving fine details and textures in the image thus can significantly improve denoising quality. These images should typically contain feature values for the first hit (i.e. the surface which is directly visible) per pixel. This works well for most surfaces but does not provide any benefits for reflections and objects visible through transparent surfaces (compared to just using the color as input). However, in certain cases this issue can be fixed by storing feature values for a subsequent hit (i.e. the reflection and/or refraction) instead of the first hit. For example, it usually works well to follow perfect specular (*delta*) paths and store features for the first diffuse or glossy surface hit instead (e.g. for perfect specular dielectrics and mirrors). This can greatly improve the quality of reflections and transmission. We will describe this approach in more detail in the following subsections.

The auxiliary feature images should be as noise-free as possible. It is not a strict requirement but too much noise in the feature images may cause residual noise in the output. Also, all feature images should use the same pixel reconstruction filter as the color image. Using a properly anti-aliased color image but aliased albedo or normal images will likely introduce artifacts around edges.

Albedo

The albedo image is the feature image that usually provides the biggest quality improvement. It should contain the approximate color of the surfaces independent of illumination and viewing angle.

For simple matte surfaces this means using the diffuse color/texture as the albedo. For other, more complex surfaces it is not always obvious what is the best way to compute the albedo, but the denoising filter is flexibile to a certain extent and works well with differently computed albedos. Thus it is not necessary to compute the strict, exact albedo values but must be always between 0 and 1.

[Example albedo image obtained using the first hit. Note that the albedos of all transparent surfaces are 1.][imgMazdaAlbedoFirstHit]

[Example albedo image obtained using the first diffuse or glossy (non-delta) hit. Note that the albedos of perfect specular (delta) transparent surfaces are computed as the Fresnel blend of the reflected and transmitted albedos.][imgMazdaAlbedoNonDeltaHit]

For metallic surfaces the albedo should be either the reflectivity at normal incidence (e.g. from the artist friendly metallic Fresnel model) or the average reflectivity; or if these are constant (not textured) or unknown, the albedo can be simply 1 as well.

The albedo for dielectric surfaces (e.g. glass) should be either 1 or, if the surface is perfect specular (i.e. has a delta BSDF), the Fresnel blend of the reflected and transmitted albedos (as previously discussed). The latter usually works better but *only* if it does not introduce too much additional noise due to random sampling. Thus we recommend to split the path into a reflected and a transmitted path at the first hit, and perhaps fall back to an albedo of 1 for subsequent dielectric hits, to avoid noise. The reflected albedo in itself can be used for mirror-like surfaces as well.

The albedo for layered surfaces can be computed as the weighted sum of the albedos of the individual layers. Non-absorbing clear coat layers can be simply ignored (or the albedo of the perfect specular reflection can be used as well) but absorption should be taken into account.

Normal

The normal image should contain the shading normals of the surfaces either in world-space or view-space. It is recommended to include normal maps to preserve as much detail as possible.

Just like any other input image, the normal image should be anti-aliased (i.e. by accumulating the normalized normals per pixel). The final accumulated normals do not have to be normalized but must be in a range symmetric about 0 (i.e. normals mapped to [0, 1] are *not* acceptable and must be remapped to e.g. $[-1, 1]$).

Similar to the albedo, the normal can be stored for either the first or a subsequent hit (if the first hit has a perfect specular/delta BSDF).

[Example normal image obtained using the first hit (the values are actually in $[-1, 1]$ but were mapped to [0, 1] for illustration purposes).][imgMazdaNormalFirstHit]

[Example normal image obtained using the first diffuse or glossy (non-delta) hit. Note that the normals of perfect specular (delta) transparent surfaces are computed as the Fresnel blend of the reflected and transmitted normals.][imgMazdaNormalNonDeltaHit]

Weights

Instead of using the built-in trained models for filtering, it is also possible to specify user-trained models at runtime. This can be achieved by passing the model *weights* blob corresponding to the specified set of features and other filter parameters, produced by the included training tool. See Section [Training] for details.

RTLightmap

The `RTLightmap` filter is a variant of the `RT` filter optimized for denoising HDR lightmaps. It does not support LDR images.

The filter can be created by passing `"RTLightmap"` to the `oidnNewFilter` function as the filter type. The filter supports the following parameters:

Table 24: Parameters supported by the `RTLightmap` filter.

Type	Format	Name	Default	Description
Image	float3	color		input color image (HDR values in $[0, +\infty)$, 3 channels)
Image	float3	output		output color image (3 channels); it can be one of the input images
Data		weights		trained model weights blob; *optional*
float		hdrScale	NaN	HDR color values are interpreted such that, multiplied by this scale, a value of 1 corresponds to a luminance level of 100 cd/m^2; if set to NaN, the scale is computed automatically (*default*)
int		maxMemoryMB	-6000	approximate maximum amount of scratch memory to use in megabytes (actual memory usage may be higher)
const int		alignment		when manually denoising the image in tiles, the tile size and offsets should be multiples of this amount of pixels
const int		overlap		when manually denoising the image in tiles, the tiles should overlap by this amount of pixels

OSPRay

OSPRay is a scalable, and portable ray tracing engine for high-performance, high-fidelity visualization.

Introduction

OSPRay is a scalable, and portable ray tracing engine for high-performance, high-fidelity visualization.

The purpose of OSPRay is to provide an open, powerful, and easy-to-use rendering library that allows one to easily build applications that use ray tracing based rendering for interactive applications (including both surface- and volume-based visualizations).

OSPRay API

To access the OSPRay API you first need to include the OSPRay header

```
#include "ospray/ospray.h"
```

where the API is compatible with C99 and C++.

Initialization and Shutdown

To use the API, OSPRay must be initialized with a "device". A device is the object which implements the API. Creating and initializing a device can be done in either of two ways: command line arguments using `ospInit` or manually instantiating a device and setting parameters on it.

Command Line Arguments

The first is to do so by giving OSPRay the command line from `main()` by calling

```
OSPError ospInit(int *argc, const char **argv);
```

OSPRay parses (and removes) its known command line parameters from your application's `main` function. For an example see the [tutorial]. For possible error codes see section *Error Handling and Status Messages*. It is important to note that the arguments passed to `ospInit()` are processed in order they are listed. The following parameters (which are prefixed by convention with "`--osp:`") are understood:

Table 25: Command line parameters accepted by OSPRay's `ospInit`.

Parameter	Description
`--osp:debug`	enables various extra checks and debug output, and disables multi-threading
`--osp:num-threads=<n>`	use `n` threads instead of per default using all detected hardware threads
`--osp:log-level=<str>`	set logging level; valid values (in order of severity) are `none`, `error`, `warning`, `info`, and `debug`
`--osp:warn-as-error`	send `warning` and `error` messages through the error callback, otherwise send `warning` messages through the message callback; must have sufficient `logLevel` to enable warnings
`--osp:verbose`	shortcut for `--osp:log-level=info` and enable debug output on `cout`, error output on `cerr`
`--osp:vv`	shortcut for `--osp:log-level=debug` and enable debug output on `cout`, error output on `cerr`
`--osp:load-modules=<name>[,...]`	load one or more modules during initialization; equivalent to calling `ospLoadModule(name)`
`--osp:log-output=<dst>`	convenience for setting where status messages go; valid values for `dst` are `cerr` and `cout`
`--osp:error-output=<dst>`	convenience for setting where error messages go; valid values for `dst` are `cerr` and `cout`
`--osp:device=<name>`	use `name` as the type of device for OSPRay to create; e.g., `--osp:device=cpu` gives you the default `cpu` device; Note if the device to be used is defined in a module, remember to pass `--osp:load-modules=<name>` first
`--osp:set-affinity=<n>`	if `1` bind software threads to hardware threads; `0` disables binding; default is `1` on KNL and `0` otherwise
`--osp:device-param:<param>:<value>[,...]`	set one or more other device parameters; equivalent to calling `ospDeviceSet*(param, value)`

Manual Device Instantiation

The second method of initialization is to explicitly create the device and possibly set parameters. This method looks almost identical to how other *objects* are created and used by OSPRay (described in later sections). The first step is to create the device with

```
OSPDevice ospNewDevice(const char *type);
```

where the `type` string maps to a specific device implementation. OSPRay always provides the "`cpu`" device, which maps to a fast, local CPU implementation. Other devices can also be added through additional modules, such as distributed MPI device implementations.

Once a device is created, you can call

```
void ospDeviceSetParam(OSPObject, const char *id, OSPDataType type, const void *mem);
```

to set parameters on the device. The semantics of setting parameters is exactly the same as `ospSetParam`, which is documented below in the *parameters* section. The following parameters can be set on all devices:

Table 26: Parameters shared by all devices.

Type	Name	Description
int	numThreads	number of threads which OSPRay should use
int	logLevel	logging level; valid values (in order of severity) are `OSP_LOG_NONE`, `OSP_LOG_ERROR`, `OSP_LOG_WARNING`, `OSP_LOG_INFO`, and `OSP_LOG_DEBUG`
string	logOutput	convenience for setting where status messages go; valid values are `cerr` and `cout`
string	errorOutput	convenience for setting where error messages go; valid values are `cerr` and `cout`
bool	debug	set debug mode; equivalent to `logLevel=debug` and `numThreads=1`
bool	warnAsError	send `warning` and `error` messages through the error callback, otherwise send `warning` messages through the message callback; must have sufficient `logLevel` to enable warnings
bool	setAffinity	bind software threads to hardware threads if set to 1; 0 disables binding omitting the parameter will let OSPRay choose

Once parameters are set on the created device, the device must be committed with

```
void ospDeviceCommit(OSPDevice);
```

To use the newly committed device, you must call

```
void ospSetCurrentDevice(OSPDevice);
```

This then sets the given device as the object which will respond to all other OSPRay API calls.

Device handle lifetimes are managed with two calls, the first which increments the internal reference count to the given `OSPDevice`

```
void ospDeviceRetain(OSPDevice)
```

and the second which decrements the reference count

```
void ospDeviceRelease(OSPDevice)
```

Users can change parameters on the device after initialization (from either method above), by calling

```
OSPDevice ospGetCurrentDevice();
```

This function returns the handle to the device currently used to respond to OSPRay API calls, where users can set/change parameters and recommit the device. If changes are made to the device that is already set as the current device, it does not need to be set as current again. Note this API call will increment the ref count of the returned device handle, so applications must use `ospDeviceRelease` when finished using the handle to avoid leaking the underlying device object. If there is no current device set, this will return an invalid `NULL` handle.

When a device is created, its reference count is initially `1`. When a device is set as the current device, it internally has its reference count incremented. Note that `ospDeviceRetain` and `ospDeviceRelease` should only be used with reference counts that the application tracks: removing reference held by the current set device should be handled by `ospShutdown`. Thus, `ospDeviceRelease` should only decrement the reference counts that come from `ospNewDevice`, `ospGetCurrentDevice`, and the number of explicit calls to `ospDeviceRetain`.

OSPRay allows applications to query runtime properties of a device in order to do enhanced validation of what device was loaded at runtime. The following function can be used to get these device-specific properties (attributes about the device, not parameter values)

```
int64_t ospDeviceGetProperty(OSPDevice, OSPDeviceProperty);
```

It returns an integer value of the queried property and the following properties can be provided as parameter:

```
OSP_DEVICE_VERSION
OSP_DEVICE_VERSION_MAJOR
OSP_DEVICE_VERSION_MINOR
OSP_DEVICE_VERSION_PATCH
OSP_DEVICE_SO_VERSION
```

Environment Variables

OSPRay's generic device parameters can be overridden via environment variables for easy changes to OSPRay's behavior without needing to change the application (variables are prefixed by convention with "`OSPRAY_`"):

Table 27: Environment variables interpreted by OSPRay.

Variable	Description
OSPRAY_NUM_THREADS	equivalent to `--osp:num-threads`
OSPRAY_LOG_LEVEL	equivalent to `--osp:log-level`
OSPRAY_LOG_OUTPUT	equivalent to `--osp:log-output`
OSPRAY_ERROR_OUTPUT	equivalent to `--osp:error-output`
OSPRAY_DEBUG	equivalent to `--osp:debug`
OSPRAY_WARN_AS_ERROR	equivalent to `--osp:warn-as-error`
OSPRAY_SET_AFFINITY	equivalent to `--osp:set-affinity`
OSPRAY_LOAD_MODULES	equivalent to `--osp:load-modules`, can be a comma separated list of modules which will be loaded in order
OSPRAY_DEVICE	equivalent to `--osp:device:`

Note that these environment variables take precedence over values specified through `ospInit` or manually set device parameters.

Error Handling and Status Messages

The following errors are currently used by OSPRay:

Table 28: Possible error codes, i.e., valid named constants of type `OSPError`.

Name	Description
OSP_NO_ERROR	no error occurred
OSP_UNKNOWN_ERROR	an unknown error occurred
OSP_INVALID_ARGUMENT	an invalid argument was specified
OSP_INVALID_OPERATION	the operation is not allowed for the specified object
OSP_OUT_OF_MEMORY	there is not enough memory to execute the command
OSP_UNSUPPORTED_CPU	the CPU is not supported (minimum ISA is SSE4.1)
OSP_VERSION_MISMATCH	a module could not be loaded due to mismatching version

These error codes are either directly return by some API functions, or are recorded to be later queried by the application via

```
OSPError ospDeviceGetLastErrorCode(OSPDevice);
```

A more descriptive error message can be queried by calling

```
const char* ospDeviceGetLastErrorMsg(OSPDevice);
```

Alternatively, the application can also register a callback function of type

```
typedef void (*OSPErrorCallback)(void *userData, OSPError, const char* errorDetails);
```

via

```
void ospDeviceSetErrorCallback(OSPDevice, OSPErrorCallback, void *userData);
```

to get notified when errors occur.

Applications may be interested in messages which OSPRay emits, whether for debugging or logging events. Applications can call

```
void ospDeviceSetStatusCallback(OSPDevice, OSPStatusCallback, void *userData);
```

in order to register a callback function of type

```
typedef void (*OSPStatusCallback)(void *userData, const char* messageText);
```

which OSPRay will use to emit status messages. By default, OSPRay uses a callback which does nothing, so any output desired by an application will require that a callback is provided. Note that callbacks for C++ `std::cout` and `std::cerr` can be alternatively set through `ospInit()` or the `OSPRAY_LOG_OUTPUT` environment variable.

Applications can clear either callback by passing `NULL` instead of an actual function pointer.

Loading OSPRay Extensions at Runtime

OSPRay's functionality can be extended via plugins (which we call "modules"), which are implemented in shared libraries. To load module `name` from `libospray_module_<name>.so` (on Linux and Mac OS X) or `ospray_module_<name>.dll` (on Windows) use

```
OSPError ospLoadModule(const char *name);
```

Modules are searched in OS-dependent paths. `ospLoadModule` returns `OSP_NO_ERROR` if the plugin could be successfully loaded.

Shutting Down OSPRay

When the application is finished using OSPRay (typically on application exit), the OSPRay API should be finalized with

```
void ospShutdown();
```

This API call ensures that the current device is cleaned up appropriately. Due to static object allocation having non-deterministic ordering, it is recommended that applications call `ospShutdown()` before the calling application process terminates.

Objects

All entities of OSPRay (the [renderer], *volumes*, *geometries*, *lights*, *cameras*, ...) are a logical specialization of `OSPObject` and share common mechanism to deal with parameters and lifetime.

An important aspect of object parameters is that parameters do not get passed to objects immediately. Instead, parameters are not visible at all to objects until they get explicitly committed to a given object via a call to

```
void ospCommit(OSPObject);
```

at which time all previously additions or changes to parameters are visible at the same time. If a user wants to change the state of an existing object (e.g., to change the origin of an already existing camera) it is perfectly valid to do so, as long as the changed parameters are recommitted.

The commit semantic allow for batching up multiple small changes, and specifies exactly when changes to objects will occur. This can impact performance and consistency for devices crossing a PCI bus or across a network.

Note that OSPRay uses reference counting to manage the lifetime of all objects, so one cannot explicitly "delete" any object. Instead, to indicate that the application does not need and does not access the given object anymore, call

```
void ospRelease(OSPObject);
```

This decreases its reference count and if the count reaches `0` the object will automatically get deleted. Passing `NULL` is not an error. Note that every handle returned via the API needs to be released when the object is no longer needed, to avoid memory leaks.

Sometimes applications may want to have more than one reference to an object, where it is desirable for the application to increment the reference count of an object. This is done with

```
void ospRetain(OSPObject);
```

It is important to note that this is only necessary if the application wants to call `ospRelease` on an object more than once: objects which contain other objects as parameters internally increment/decrement ref counts and should not be explicitly done by the application.

Parameters

Parameters allow to configure the behavior of and to pass data to objects. However, objects do *not* have an explicit interface for reasons of high flexibility and a more stable compile-time API. Instead, parameters are passed separately to objects in an arbitrary order, and unknown parameters will simply be ignored (though a warning message will be posted). The following function allows adding various types of parameters with name `id` to a given object:

```
void ospSetParam(OSPObject, const char *id, OSPDataType type, const void *mem);
```

The valid parameter names for all `OSPObject`s and what types are valid are discussed in future sections.

Note that `mem` must always be a pointer *to* the object, otherwise accidental type casting can occur. This is especially true for pointer types (`OSP_VOID_PTR` and `OSPObject` handles), as they will implicitly cast to `void\ *`, but be incorrectly interpreted. To help with some of these issues, there also exist variants of `ospSetParam` for specific types, such as `ospSetInt` and `ospSetVec3f` in the OSPRay utility library (found in `ospray_util.h`).

Users can also remove parameters that have been explicitly set from `ospSetParam`. Any parameters which have been removed will go back to their default value during the next commit unless a new parameter was set after the parameter was removed. To remove a parameter, use

```
void ospRemoveParam(OSPObject, const char *id);
```

Data

OSPRay consumes data arrays from the application using a specific object type, `OSPData`. There are several components to describing a data array: element type, 1/2/3 dimensional striding, and whether the array is shared with the application or copied into opaque, OSPRay-owned memory.

Shared data arrays require that the application's array memory outlives the lifetime of the created `OSPData`, as OSPRay is referring to application memory. Where this is not preferable, applications use opaque arrays to allow the `OSPData` to own the lifetime of the array memory. However, opaque arrays dictate the cost of copying data into it, which should be kept in mind.

Thus, the most efficient way to specify a data array from the application is to created a shared data array, which is done with

```
OSPData ospNewSharedData(const void *sharedData,
    OSPDataType,
    uint64_t numItems1,
    int64_t byteStride1 = 0,
    uint64_t numItems2 = 1,
    int64_t byteStride2 = 0,
    uint64_t numItems3 = 1,
    int64_t byteStride3 = 0);
```

The call returns an `OSPData` handle to the created array. The calling program guarantees that the `sharedData` pointer will remain valid for the duration that this data array is being used. The number of elements `numItems` must be positive (there cannot be an empty data object). The data is arranged in three dimensions, with specializations to two or one dimension (if some `numItems` are 1). The distance between consecutive elements (per dimension) is given in bytes with `byteStride` and can also be negative. If `byteStride` is zero it will be determined automatically (e.g., as `sizeof(type)`). Strides do not need to be ordered, i.e., `byteStride2` can be smaller than `byteStride1`,

which is equivalent to a transpose. However, if the stride should be calculated, then an ordering in dimensions is assumed to disambiguate, i.e., `byteStride1 < byteStride2 < byteStride3`.

The enum type `OSPDataType` describes the different element types that can be represented in OSPRay; valid constants are listed in the table below.

Table 29: Valid named constants for `OSPDataType`.

Type/Name	Description
OSP_DEVICE	API device object reference
OSP_DATA	data reference
OSP_OBJECT	generic object reference
OSP_CAMERA	camera object reference
OSP_FRAMEBUFFER	framebuffer object reference
OSP_LIGHT	light object reference
OSP_MATERIAL	material object reference
OSP_TEXTURE	texture object reference
OSP_RENDERER	renderer object reference
OSP_WORLD	world object reference
OSP_GEOMETRY	geometry object reference
OSP_VOLUME	volume object reference
OSP_TRANSFER_FUNCTION	transfer function object reference
OSP_IMAGE_OPERATION	image operation object reference
OSP_STRING	C-style zero-terminated character string
OSP_CHAR, OSP_VEC[234]C	8 bit signed character scalar and [234]-element vector
OSP_UCHAR, OSP_VEC[234]UC	8 bit unsigned character scalar and [234]-element vector
OSP_SHORT, OSP_VEC[234]S	16 bit unsigned integer scalar and [234]-element vector
OSP_USHORT, OSP_VEC[234]US	16 bit unsigned integer scalar and [234]-element vector
OSP_INT, OSP_VEC[234]I	32 bit signed integer scalar and [234]-element vector
OSP_UINT, OSP_VEC[234]UI	32 bit unsigned integer scalar and [234]-element vector
OSP_LONG, OSP_VEC[234]L	64 bit signed integer scalar and [234]-element vector
OSP_ULONG, OSP_VEC[234]UL	64 bit unsigned integer scalar and [234]-element vector
OSP_FLOAT, OSP_VEC[234]F	32 bit single precision floating-point scalar and [234]-element vector
OSP_DOUBLE, OSP_VEC[234]D	64 bit double precision floating-point scalar and [234]-element vector
OSP_BOX[1234]I	32 bit integer box (lower + upper bounds)
OSP_BOX[1234]F	32 bit single precision floating-point box (lower + upper bounds)
OSP_LINEAR[23]F	32 bit single precision floating-point linear transform ([23] vectors)
OSP_AFFINE[23]F	32 bit single precision floating-point affine transform (linear transform plus translation)
OSP_VOID_PTR	raw memory address (only found in module extensions)

If the elements of the array are handles to objects, then their reference counter is incremented.

An opaque `OSPData` with memory allocated by OSPRay is created with

```
OSPData ospNewData(OSPDataType,
    uint64_t numItems1,
    uint64_t numItems2 = 1,
    uint64_t numItems3 = 1);
```

To allow for (partial) copies or updates of data arrays use

```
void ospCopyData(const OSPData source,
    OSPData destination,
    uint64_t destinationIndex1 = 0,
```

(continues on next page)

(continued from previous page)

```
    uint64_t destinationIndex2 = 0,
    uint64_t destinationIndex3 = 0);
```

which will copy the whole[1] content of the `source` array into `destination` at the given location `destinationIndex`. The `OSPDataType`s of the data objects must match. The region to be copied must be valid inside the destination, i.e., in all dimensions, `destinationIndex + sourceSize <= destinationSize`. The affected region `[destinationIndex, destinationIndex + sourceSize)` is marked as dirty, which may be used by OSPRay to only process or update that sub-region (e.g., updating an acceleration structure). If the destination array is shared with OSPData by the application (created with `ospNewSharedData`), then

- the source array must be shared as well (thus `ospCopyData` cannot be used to read opaque data)
- if source and destination memory overlaps (aliasing), then behavior is undefined
- except if source and destination regions are identical (including matching strides), which can be used by application to mark that region as dirty (instead of the whole `OSPData`)

To add a data array as parameter named `id` to another object call also use

```
void ospSetObject(OSPObject, const char *id, OSPData);
```

Volumes

Volumes are volumetric data sets with discretely sampled values in 3D space, typically a 3D scalar field. To create a new volume object of given type `type` use

```
OSPVolume ospNewVolume(const char *type);
```

Note that OSPRay's implementation forwards `type` directly to Open VKL, allowing new Open VKL volume types to be usable within OSPRay without the need to change (or even recompile) OSPRay.

Structured Regular Volume

Structured volumes only need to store the values of the samples, because their addresses in memory can be easily computed from a 3D position. A common type of structured volumes are regular grids.

Structured regular volumes are created by passing the type string "`structuredRegular`" to `ospNewVolume`. Structured volumes are represented through an `OSPData` 3D array `data` (which may or may not be shared with the application). The voxel data must be laid out in xyz-order[2] and can be compact (best for performance) or can have a stride between voxels, specified through the `byteStride1` parameter when creating the `OSPData`. Only 1D strides are supported, additional strides between scanlines (2D, `byteStride2`) and slices (3D, `byteStride3`) are not.

The parameters understood by structured volumes are summarized in the table below.

[1] The number of items to be copied is defined by the size of the source array

[2] For consecutive memory addresses the x-index of the corresponding voxel changes the quickest.

Table 30: Configuration parameters for structured regular volumes.

Type	Name	Default	Description
vec3f	gridOrigin	$(0, 0, 0)$	origin of the grid in object-space
vec3f	gridSpacing	$(1, 1, 1)$	size of the grid cells in object-space
OSPData	data		the actual voxel 3D *data*
int	filter	`OSP_VOLUME_FILTER_TRILINEAR`	filter used for reconstructing the field, also allowed is `OSP_VOLUME_FILTER_NEAREST`
int	gradientFilter	same as `filter`	filter used during gradient computations

The size of the volume is inferred from the size of the 3D array `data`, as is the type of the voxel values (currently supported are: `OSP_UCHAR`, `OSP_SHORT`, `OSP_USHORT`, `OSP_FLOAT`, and `OSP_DOUBLE`).

Structured Spherical Volume

Structured spherical volumes are also supported, which are created by passing a type string of "`structuredSpherical`" to `ospNewVolume`. The grid dimensions and parameters are defined in terms of radial distance r, inclination angle θ, and azimuthal angle ϕ, conforming with the ISO convention for spherical coordinate systems. The coordinate system and parameters understood by structured spherical volumes are summarized below.

[Coordinate system of structured spherical volumes.][imgStructuredSphericalCoords]

Table 31: Configuration parameters for structured spherical volumes.

Type	Name	Default	Description
vec3f	gridOrigin	$(0, 0, 0)$	origin of the grid in units of (r, θ, ϕ); angles in degrees
vec3f	gridSpacing	$(1, 1, 1)$	size of the grid cells in units of (r, θ, ϕ); angles in degrees
OSPData	data		the actual voxel 3D *data*
int	filter	`OSP_VOLUME_FILTER_TRILINEAR`	filter used for reconstructing the field, also allowed is `OSP_VOLUME_FILTER_NEAREST`
int	gradientFilter	same as `filter`	filter used during gradient computations

The dimensions (r, θ, ϕ) of the volume are inferred from the size of the 3D array `data`, as is the type of the voxel values (currently supported are: `OSP_UCHAR`, `OSP_SHORT`, `OSP_USHORT`, `OSP_FLOAT`, and `OSP_DOUBLE`).

These grid parameters support flexible specification of spheres, hemispheres, spherical shells, spherical wedges, and so forth. The grid extents (computed as `[gridOrigin, gridOrigin + (dimensions - 1) * gridSpacing]`) however must be constrained such that:

- $r \geq 0$
- $0 \leq \theta \leq 180$
- $0 \leq \phi \leq 360$

Adaptive Mesh Refinement (AMR) Volume

OSPRay currently supports block-structured (Berger-Colella) AMR volumes. Volumes are specified as a list of blocks, which exist at levels of refinement in potentially overlapping regions. Blocks exist in a tree structure, with coarser refinement level blocks containing finer blocks. The cell width is equal for all blocks at the same refinement level, though blocks at a coarser level have a larger cell width than finer levels.

There can be any number of refinement levels and any number of blocks at any level of refinement. An AMR volume type is created by passing the type string "`amr`" to `ospNewVolume`.

Blocks are defined by three parameters: their bounds, the refinement level in which they reside, and the scalar data contained within each block.

Note that cell widths are defined *per refinement level*, not per block.

Table 32: Configuration parameters for AMR volumes.

Type	Name	Default	Description
`OSPAMRMethod`	method	`OSP_AMR_CURRENT`	`OSPAMRMethod` sampling method. Supported methods are:
			`OSP_AMR_CURRENT`
			`OSP_AMR_FINEST`
			`OSP_AMR_OCTANT`
float[]	cellWidth	NULL	array of each level's cell width
box3i[]	block.bounds	NULL	*data* array of grid sizes (in voxels) for each AMR block
int[]	block.level	NULL	array of each block's refinement level
OSPData[]	block.data	NULL	*data* array of OSPData containing the actual scalar voxel data, only `OSP_FLOAT` is supported as `OSPDataType`
vec3f	gridOrigin	$(0, 0, 0)$	origin of the grid in world-space
vec3f	gridSpacing	$(1, 1, 1)$	size of the grid cells in world-space

Lastly, note that the `gridOrigin` and `gridSpacing` parameters act just like the structured volume equivalent, but they only modify the root (coarsest level) of refinement.

In particular, OSPRay's / Open VKL's AMR implementation was designed to cover Berger-Colella [1] and Chombo [2] AMR data. The `method` parameter above determines the interpolation method used when sampling the volume.

OSP_AMR_CURRENT finds the finest refinement level at that cell and interpolates through this "current" level

OSP_AMR_FINEST will interpolate at the closest existing cell in the volume-wide finest refinement level regardless of the sample cell's level

OSP_AMR_OCTANT interpolates through all available refinement levels at that cell. This method avoids discontinuities at refinement level boundaries at the cost of performance

Details and more information can be found in the publication for the implementation [3].

1. M.J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics." Journal of Computational Physics 82.1 (1989): 64-84. DOI: 10.1016/0021-9991(89)90035-1

2. M. Adams, P. Colella, D.T. Graves, J.N. Johnson, N.D. Keen, T.J. Ligocki, D.F. Martin. P.W. McCorquodale, D. Modiano. P.O. Schwartz, T.D. Sternberg, and B. Van Straalen, "Chombo Software Package for AMR Applications – Design Document", Lawrence Berkeley National Laboratory Technical Report LBNL-6616E.

3. I. Wald, C. Brownlee, W. Usher, and A. Knoll, "CPU volume rendering of adaptive mesh refinement data". SIGGRAPH Asia 2017 Symposium on Visualization – SA '17, 18(8), 1–8. DOI:

10.1145/3139295.3139305

Unstructured Volume

Unstructured volumes can have their topology and geometry freely defined. Geometry can be composed of tetrahedral, hexahedral, wedge or pyramid cell types. The data format used is compatible with VTK and consists of multiple arrays: vertex positions and values, vertex indices, cell start indices, cell types, and cell values. An unstructured volume type is created by passing the type string "`unstructured`" to `ospNewVolume`.

Sampled cell values can be specified either per-vertex (`vertex.data`) or per-cell (`cell.data`). If both arrays are set, `cell.data` takes precedence.

Similar to a mesh, each cell is formed by a group of indices into the vertices. For each vertex, the corresponding (by array index) data value will be used for sampling when rendering, if specified. The index order for a tetrahedron is the same as `VTK_TETRA`: bottom triangle counterclockwise, then the top vertex.

For hexahedral cells, each hexahedron is formed by a group of eight indices into the vertices and data values. Vertex ordering is the same as `VTK_HEXAHEDRON`: four bottom vertices counterclockwise, then top four counterclockwise.

For wedge cells, each wedge is formed by a group of six indices into the vertices and data values. Vertex ordering is the same as `VTK_WEDGE`: three bottom vertices counterclockwise, then top three counterclockwise.

For pyramid cells, each cell is formed by a group of five indices into the vertices and data values. Vertex ordering is the same as `VTK_PYRAMID`: four bottom vertices counterclockwise, then the top vertex.

To maintain VTK data compatibility, the `index` array may be specified with cell sizes interleaved with vertex indices in the following format: $n, id_1, ..., id_n, m, id_1, ..., id_m$. This alternative `index` array layout can be enabled through the `indexPrefixed` flag (in which case, the `cell.type` parameter must be omitted).

Table 33: Configuration parameters for unstructured volumes.

Type	Name	Default	Description
vec3f[]	vertex.position		*data* array of vertex positions
float[]	vertex.data		*data* array of vertex data values to be sampled
uint32[] / uint64[]	index		*data* array of indices (into the vertex array(s)) that form cells
bool	indexPrefixed	false	indicates that the `index` array is compatible to VTK, where the indices of each cell are prefixed with the number of vertices
uint32[] / uint64[]	cell.index		*data* array of locations (into the index array), specifying the first index of each cell
float[]	cell.data		*data* array of cell data values to be sampled
uint8[]	cell.type		*data* array of cell types (VTK compatible), only set if `indexPrefixed = false` false. Supported types are:
			`OSP_TETRAHEDRON`
			`OSP_HEXAHEDRON`
			`OSP_WEDGE`
			`OSP_PYRAMID`
bool	hexIterative	false	hexahedron interpolation method, defaults to fast non-iterative version which could have rendering inaccuracies may appear if hex is not parallelepiped
bool	precomputedNormals	false	whether to accelerate by precomputing, at a cost of 12 bytes/face
int	maxIteratorDepth	6	do not descend further than to this BVH depth during interval iteration

VDB Volume

VDB volumes implement a data structure that is very similar to the data structure outlined in Museth [1], they are created by passing the type string "vdb" to ospNewVolume.

The data structure is a hierarchical regular grid at its core: Nodes are regular grids, and each grid cell may either store a constant value (this is called a tile), or child pointers. Nodes in VDB trees are wide: Nodes on the first level have a resolution of 32^3 voxels, on the next level 16^3, and on the leaf level 8^3 voxels. All nodes on a given level have the same resolution. This makes it easy to find the node containing a coordinate using shift operations (see [1]). VDB leaf nodes are implicit in OSPRay / Open VKL: they are stored as pointers to user-provided data.

[Topology of VDB volumes.][imgVdbStructure]

VDB volumes interpret input data as constant cells (which are then potentially filtered). This is in contrast to structuredRegular volumes, which have a vertex-centered interpretation.

The VDB implementation in OSPRay / Open VKL follows the following goals:

- Efficient data structure traversal on vector architectures.
- Enable the use of industry-standard .vdb files created through the OpenVDB library.
- Compatibility with OpenVDB on a leaf data level, so that .vdb file may be loaded with minimal overhead.

VDB volumes have the following parameters:

Table 34: Configuration parameters for VDB volumes.

Type	Name	Description
int	maxIteratorDepth	do not descend further than to this depth during interval iteration, the maximum value and the default is 3
int	maxSamplingDepth	do not descend further than to this depth during sampling, the maximum value and the default is 3
uint32[]	node.level	level on which each input node exists, may be 1, 2 or 3 (levels are counted from the root level = 0 down)
vec3i[]	node.origin	the node origin index (per input node)
OSPData[]	node.data	*data* arrays with the node data (per input node). Nodes that are tiles are expected to have single-item arrays. Leaf-nodes with grid data expected to have compact 3D arrays in zyx layout (z changes most quickly) with the correct number of voxels for the level. Only OSP_FLOAT is supported as field OSPDataType.
int	filter	filter used for reconstructing the field, default is OSP_VOLUME_FILTER_TRILINEAR, alternatively OSP_VOLUME_FILTER_NEAREST.
int	gradientFilter	filter used for reconstructing the field during gradient computations, default same as filter

1. Museth, K. VDB: High-Resolution Sparse Volumes with Dynamic Topology. ACM Transactions on Graphics 32(3), 2013. DOI: 10.1145/2487228.2487235

Particle Volume

Particle volumes consist of a set of points in space. Each point has a position, a radius, and a weight typically associated with an attribute. Particle volumes are created by passing the type string "`particle`" to `ospNewVolume`.

A radial basis function defines the contribution of that particle. Currently, we use the Gaussian radial basis function

$$\phi(P) = w \exp\left(-\frac{(P-p)^2}{2r^2}\right),$$

where P is the particle position, p is the sample position, r is the radius and w is the weight. At each sample, the scalar field value is then computed as the sum of each radial basis function ϕ, for each particle that overlaps it.

The OSPRay / Open VKL implementation is similar to direct evaluation of samples in Reda et al. [2]. It uses an Embree-built BVH with a custom traversal, similar to the method in [1].

Table 35: Configuration parameters for particle volumes.

Type	Name	Default	Description
vec3f[]	particle.position		*data* array of particle positions
float[]	particle.radius		*data* array of particle radii
float[]	particle.weight	NULL	optional *data* array of particle weights, specifying the height of the kernel.
float	radiusSupportFactor	3.0	The multiplier of the particle radius required for support. Larger radii ensure smooth results at the cost of performance. In the Gaussian kernel, the radius is one standard deviation (σ), so a value of 3 corresponds to 3σ.
float	clampMaxCumulativeValue	0	The maximum cumulative value possible, set by user. All cumulative values will be clamped to this, and further traversal (RBF summation) of particle contributions will halt when this value is reached. A value of zero or less turns this off.
bool	estimateValueRanges	true	Enable heuristic estimation of value ranges which are used in internal acceleration structures as well as for determining the volume's overall value range. When set to `false`, the user *must* specify `clampMaxCumulativeValue`, and all value ranges will be assumed [0, `clampMaxCumulativeValue`]. Disabling this switch may improve volume commit time, but will make volume rendering less efficient.
int	maxIteratorDepth	6	do not descend further than to this BVH depth during interval iteration

1. A. Knoll, I. Wald, P. Navratil, A. Bowen, K. Reda, M.E., Papka, and K. Gaither, "RBF Volume Ray Casting on Multicore and Manycore CPUs", 2014, Computer Graphics Forum, 33: 71–80. doi:10.1111/cgf.12363

2. K. Reda, A. Knoll, K. Nomura, M. E. Papka, A. E. Johnson and J. Leigh, "Visualizing large-scale atomistic simulations in ultra-resolution immersive environments", 2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), Atlanta, GA, 2013, pp. 59–65.

Transfer Function

Transfer functions map the scalar values of volumes to color and opacity and thus they can be used to visually emphasize certain features of the volume. To create a new transfer function of given type `type` use

```
OSPTransferFunction ospNewTransferFunction(const char *type);
```

The returned handle can be assigned to a volumetric model (described below) as parameter "`transferFunction`" using `ospSetObject`.

One type of transfer function that is supported by OSPRay is the linear transfer function, which interpolates between given equidistant colors and opacities. It is create by passing the string "`piecewiseLinear`" to `ospNewTransferFunction` and it is controlled by these parameters:

Table 36: Parameters accepted by the linear transfer function.

Type	Name	Description
vec3f[]	color	*data* array of RGB colors
float[]	opacity	*data* array of opacities
vec2f	valueRange	domain (scalar range) this function maps from

The arrays `color` and `opacity` can be of different length.

VolumetricModels

Volumes in OSPRay are given volume rendering appearance information through VolumetricModels. This decouples the physical representation of the volume (and possible acceleration structures it contains) to rendering-specific parameters (where more than one set may exist concurrently). To create a volume instance, call

```
OSPVolumetricModel ospNewVolumetricModel(OSPVolume volume);
```

The passed volume can be `NULL` as long as the volume to be used is passed as a parameter. If both a volume is specified on object creation and as a parameter, the parameter value is used. If the parameter value is later removed, the volume object passed on object creation is again used.

Table 37: Parameters understood by VolumetricModel.

Type	Name	Default	Description
OSPTransferFunction	transferFunction		*transfer function* to use
float	densityScale	1.0	makes volumes uniformly thinner or thicker
float	anisotropy	0.0	anisotropy of the (Henyey-Greenstein) phase function in [-1, 1] (*path tracer* only), default to isotropic scattering
OSPVolume	volume		optional [volume] object this model references

Geometries

Geometries in OSPRay are objects that describe intersectable surfaces. To create a new geometry object of given type `type` use

```
OSPGeometry ospNewGeometry(const char *type);
```

Note that in the current implementation geometries are limited to a maximum of 2^{32} primitives.

Mesh

A mesh consisting of either triangles or quads is created by calling `ospNewGeometry` with type string "`mesh`". Once created, a mesh recognizes the following parameters:

Table 38: Parameters defining a mesh geometry.

Type	Name	Description
vec3f[]	vertex.position	*data* array of vertex positions
vec3f[]	vertex.normal	*data* array of vertex normals
vec4f[] / vec3f[]	vertex.color	*data* array of vertex colors (RGBA/RGB)
vec2f[]	vertex.texcoord	*data* array of vertex texture coordinates
vec3ui[] / vec4ui[]	index	*data* array of (either triangle or quad) indices (into the vertex array(s))

The data type of index arrays differentiates between the underlying geometry, triangles are used for a index with `vec3ui` type and quads for `vec4ui` type. Quads are internally handled as a pair of two triangles, thus mixing triangles and quads is supported by encoding some triangle as a quad with the last two vertex indices being identical (`w=z`).

The `vertex.position` and `index` arrays are mandatory to create a valid mesh.

Subdivision

A mesh consisting of subdivision surfaces, created by specifying a geometry of type "`subdivision`". Once created, a subdivision recognizes the following parameters:

Table 39: Parameters defining a Subdivision geometry.

Type	Name	Description
vec3f[]	vertex.position	*data* array of vertex positions
vec4f[]	vertex.color	optional *data* array of vertex colors (RGBA)
vec2f[]	vertex.texcoord	optional *data* array of vertex texture coordinates
float	level	global level of tessellation, default 5
uint[]	index	*data* array of indices (into the vertex array(s))
float[]	index.level	optional *data* array of per-edge levels of tessellation, overrides global level
uint[]	face	optional *data* array holding the number of indices/edges (3 to 15) per face, defaults to 4 (a pure quad mesh)
vec2i[]	edge-Crease.index	optional *data* array of edge crease indices
float[]	edge-Crease.weight	optional *data* array of edge crease weights
uint[]	vertex-Crease.index	optional *data* array of vertex crease indices
float[]	vertex-Crease.weight	optional *data* array of vertex crease weights
uchar	mode	subdivision edge boundary mode, supported modes are:
		`OSP_SUBDIVISION_NO_BOUNDARY`
		`OSP_SUBDIVISION_SMOOTH_BOUNDARY` (default)
		`OSP_SUBDIVISION_PIN_CORNERS`
		`OSP_SUBDIVISION_PIN_BOUNDARY`
		`OSP_SUBDIVISION_PIN_ALL`

The `vertex` and `index` arrays are mandatory to create a valid subdivision surface. If no `face` array is present then a pure quad mesh is assumed (the number of indices must be a multiple of 4). Optionally supported are edge and vertex creases.

Spheres

A geometry consisting of individual spheres, each of which can have an own radius, is created by calling `ospNewGeometry` with type string "`sphere`". The spheres will not be tessellated but rendered procedurally and are thus perfectly round. To allow a variety of sphere representations in the application this geometry allows a flexible way of specifying the data of center position and radius within a *data* array:

Table 40: Parameters defining a spheres geometry.

Type	Name	Default	Description
vec3f[]	sphere.position		*data* array of center positions
float[]	sphere.radius	NULL	optional *data* array of the per-sphere radius
vec2f[]	sphere.texcoord	NULL	optional *data* array of texture coordinates (constant per sphere)
float	radius	0.01	default radius for all spheres (if `sphere.radius` is not set)

Curves

A geometry consisting of multiple curves is created by calling `ospNewGeometry` with type string "`curve`". The parameters defining this geometry are listed in the table below.

Table 41: Parameters defining a curves geometry.

Type	Name	Description
vec4f[]	vertex.position_radius	*data* array of vertex position and per-vertex radius
vec2f[]	vertex.texcoord	*data* array of per-vertex texture coordinates
vec4f[]	vertex.color	*data* array of corresponding vertex colors (RGBA)
vec3f[]	vertex.normal	*data* array of curve normals (only for "ribbon" curves)
vec4f[]	vertex.tangent	*data* array of curve tangents (only for "hermite" curves)
uint32[]	index	*data* array of indices to the first vertex or tangent of a curve segment
uchar	type	`OSPCurveType` for rendering the curve. Supported types are:
		`OSP_FLAT`
		`OSP_ROUND`
		`OSP_RIBBON`
		`OSP_DISJOINT`
uchar	basis	`OSPCurveBasis` for defining the curve. Supported bases are:
		`OSP_LINEAR`
		`OSP_BEZIER`
		`OSP_BSPLINE`
		`OSP_HERMITE`
		`OSP_CATMULL_ROM`

Positions in `vertex.position_radius` parameter supports per-vertex varying radii with data type `vec4f[]` and instantiate Embree curves internally for the relevant type/basis mapping.

The following section describes the properties of different curve basis' and how they use the data provided in data buffers:

OSP_LINEAR The indices point to the first of 2 consecutive control points in the vertex buffer. The first control point is the start and the second control point the end of the line segment. The curve goes through all control points listed in the vertex buffer.

OSP_BEZIER The indices point to the first of 4 consecutive control points in the vertex buffer. The first control point represents the start point of the curve, and the 4th control point the end point of the curve. The Bézier basis is interpolating, thus the curve does go exactly through the first and fourth control vertex.

OSP_BSPLINE The indices point to the first of 4 consecutive control points in the vertex buffer. This basis is not interpolating, thus the curve does in general not go through any of the control points directly. Using this basis, 3 control points can be shared for two continuous neighboring curve segments, e.g., the curves $(p0, p1, p2, p3)$ and $(p1, p2, p3, p4)$ are C1 continuous. This feature make this basis a good choice to construct continuous multi-segment curves, as memory consumption can be kept minimal.

OSP_HERMITE It is necessary to have both vertex buffer and tangent buffer for using this basis. The indices point to the first of 2 consecutive points in the vertex buffer, and the first of 2 consecutive tangents in the tangent buffer. This basis is interpolating, thus does exactly go through the first and second control point, and the first order derivative at the begin and end matches exactly the value specified in the tangent buffer. When connecting two segments continuously, the end point and tangent of the previous segment can be shared.

OSP_CATMULL_ROM The indices point to the first of 4 consecutive control points in the vertex buffer. If $(p0, p1, p2, p3)$ represent the points then this basis goes through $p1$ and $p2$, with tangents as $(p2 - p0)/2$ and $(p3 - p1)/2$.

The following section describes the properties of different curve types' and how they define the geometry of a curve:

OSP_FLAT This type enables faster rendering as the curve is rendered as a connected sequence of ray facing quads.

OSP_ROUND This type enables rendering a real geometric surface for the curve which allows closeup views. This mode renders a sweep surface by sweeping a varying radius circle tangential along the curve.

OSP_RIBBON The type enables normal orientation of the curve and requires a normal buffer be specified along with vertex buffer. The curve is rendered as a flat band whose center approximately follows the provided vertex buffer and whose normal orientation approximately follows the provided normal buffer. Not supported for basis `OSP_LINEAR`.

OSP_DISJOINT Only supported for basis `OSP_LINEAR`; the segments are open and not connected at the joints, i.e., the curve segments are either individual cones or cylinders.

Boxes

OSPRay can directly render axis-aligned bounding boxes without the need to convert them to quads or triangles. To do so create a boxes geometry by calling `ospNewGeometry` with type string "`box`".

Table 42: Parameters defining a boxes geometry.

Type	Name	Description
box3f[]	box	*data* array of boxes

Planes

OSPRay can directly render planes defined by plane equation coefficients in its implicit form $ax + by + cz + d = 0$. By default planes are infinite but their extents can be limited by defining optional bounding boxes. A planes geometry can be created by calling `ospNewGeometry` with type string "`plane`".

Table 43: Parameters defining a planes geometry.

Type	Name	Description
vec4f[]	plane.coefficients	*data* array of plane coefficients (a, b, c, d)
box3f[]	plane.bounds	optional *data* array of bounding boxes

Isosurfaces

OSPRay can directly render multiple isosurfaces of a volume without first tessellating them. To do so create an isosurfaces geometry by calling `ospNewGeometry` with type string "`isosurface`". The appearance information of the surfaces is set through the Geometric Model. Per-isosurface colors can be set by passing per-primitive colors to the Geometric Model, in order of the isosurface array.

Table 44: Parameters defining an isosurfaces geometry.

Type	Name	Description
float	isovalue	single isovalues
float[]	isovalue	*data* array of isovalues
OSPVolume	volume	handle of the [Volume] to be isosurfaced

GeometricModels

Geometries are matched with surface appearance information through GeometricModels. These take a geometry, which defines the surface representation, and applies either full-object or per-primitive color and material information. To create a geometric model, call

```
OSPGeometricModel ospNewGeometricModel(OSPGeometry geometry);
```

The passed geometry can be `NULL` as long as the geometry to be used is passed as a parameter. If both a geometry is specified on object creation and as a parameter, the parameter value is used. If the parameter value is later removed, the geometry object passed on object creation is again used.

Color and material are fetched with the primitive ID of the hit (clamped to the valid range, thus a single color or material is fine), or mapped first via the `index` array (if present). All parameters are optional, however, some renderers (notably the *path tracer*) require a material to be set. Materials are either handles of `OSPMaterial`, or indices into the `material` array on the [renderer], which allows to build a *world* which can be used by different types of renderers.

An `invertNormals` flag allows to invert (shading) normal vectors of the rendered geometry. That is particularly useful for clipping. By changing normal vectors orientation one can control whether inside or outside of the clipping geometry is being removed. For example, a clipping geometry with normals oriented outside clips everything what's inside.

Table 45: Parameters understood by GeometricModel.

Type	Name	Description
OSPMaterial / uint32	material	optional [material] applied to the geometry, may be an index into the `material` parameter on the [renderer] (if it exists)
vec4f	color	optional color assigned to the geometry
OSPMaterial[] / uint32[]	material	optional *data* array of (per-primitive) materials, may be an index into the `material` parameter on the renderer (if it exists)
vec4f[]	color	optional *data* array of (per-primitive) colors
uint8[]	index	optional *data* array of per-primitive indices into `color` and `material`
bool	invertNormals	inverts all shading normals (Ns), default false
OSPGeometry	geometry	optional [geometry] object this model references

Lights

To create a new light source of given type `type` use

```
OSPLight ospNewLight(const char *type);
```

All light sources accept the following parameters:

Table 46: Parameters accepted by all lights.

Type	Name	Default	Description
vec3f	color	white	color of the light
float	intensity	1	intensity of the light (a factor)
uchar	intensityQuantity		`OSPIntensityQuantity` to set the radiative quantity represented by `intensity`. The default value depends on the light source.
bool	visible	true	whether the light can be directly seen

In OSPRay the `intensity` parameter of a light source can correspond to different types of radiative quantities. The type of the value represented by a light's `intensity` parameter is set using `intensityQuantity`, which accepts values from the enum type `OSPIntensityQuantity`. The supported types of `OSPIntensityQuantity` differ between the different light sources (see documentation of each specific light source).

Table 47: Types of radiative quantities used to interpret a light's `intensity` parameter.

Name	Description
OSP_INTENSITY_QUANTITY_POWER	the overall amount of light energy emitted by the light source into the scene, unit is W
OSP_INTENSITY_QUANTITY_INTENSITY	the overall amount of light emitted by the light in a given direction, unit is W/sr
OSP_INTENSITY_QUANTITY_RADIANCE	the amount of light emitted by a point on the light source in a given direction, unit is W/sr/m^2
OSP_INTENSITY_QUANTITY_IRRADIANCE	the amount of light arriving at a surface point, assuming the light is oriented towards to the surface, unit is W/m^2

The following light types are supported by most OSPRay renderers.

Directional Light / Distant Light

The distant light (or traditionally the directional light) is thought to be far away (outside of the scene), thus its light arrives (almost) as parallel rays. It is created by passing the type string "`distant`" to `ospNewLight`. The distant light supports `OSP_INTENSITY_QUANTITY_RADIANCE` and `OSP_INTENSITY_QUANTITY_IRRADIANCE` (default) as `intensityQuantity` parameter value. In addition to the *general parameters* understood by all lights the distant light supports the following special parameters:

Table 48: Special parameters accepted by the distant light.

Type	Name	Description
vec3f	direction	main emission direction of the distant light
float	angularDiameter	apparent size (angle in degree) of the light

Setting the angular diameter to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling (like the *path tracer*). For instance, the apparent size of the sun is about 0.53°.

Point Light / Sphere Light

The sphere light (or the special case point light) is a light emitting uniformly in all directions from the surface toward the outside. It does not emit any light toward the inside of the sphere. It is created by passing the type string "`sphere`" to `ospNewLight`. The point light supports `OSP_INTENSITY_QUANTITY_POWER`, `OSP_INTENSITY_QUANTITY_INTENSITY` (default) and `OSP_INTENSITY_QUANTITY_RADIANCE` as `intensityQuantity` parameter value. In addition to the *general parameters* understood by all lights the sphere light supports the following special parameters:

Table 49: Special parameters accepted by the sphere light.

Type	Name	Description
vec3f	position	the center of the sphere light, in world-space
float	radius	the size of the sphere light

Setting the radius to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling (like the *path tracer*).

Spotlight / Photometric Light

The spotlight is a light emitting into a cone of directions. It is created by passing the type string "`spot`" to `ospNewLight`. The spotlight supports `OSP_INTENSITY_QUANTITY_POWER`, `OSP_INTENSITY_QUANTITY_INTENSITY` (default) and `OSP_INTENSITY_QUANTITY_RADIANCE` as `intensityQuantity` parameter value. In addition to the *general parameters* understood by all lights the spotlight supports the special parameters listed in the table.

Table 50: Special parameters accepted by the spotlight.

Type	Name	Default	Description
vec3f	position	$(0, 0, 0)$	the center of the spotlight, in world-space
vec3f	direction	$(0, 0, 1)$	main emission direction of the spot
float	openingAngle	180	full opening angle (in degree) of the spot; outside of this cone is no illumination
float	penumbraAngle	5	size (angle in degree) of the "penumbra", the region between the rim (of the illumination cone) and full intensity of the spot; should be smaller than half of `openingAngle`
float	radius	0	the size of the spotlight, the radius of a disk with normal `direction`
float	innerRadius	0	in combination with `radius` turns the disk into a ring
float[]	intensityDistribution		luminous intensity distribution for photometric lights; can be 2D for asymmetric illumination; values are assumed to be uniformly distributed
vec3f	c0		orientation, i.e., direction of the C0-(half)plane (only needed if illumination via `intensityDistribution` is asymmetric)

[Angles used by the spotlight.][imgSpotLight]

Setting the radius to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling (like the *path tracer*). Additionally setting the inner radius will result in a ring instead of a disk emitting the light.

Measured light sources (IES, EULUMDAT, ...) are supported by providing an `intensityDistribution` *data* array to modulate the intensity per direction. The mapping is using the C-γ coordinate system (see also below figure): the values of the first (or only) dimension of `intensityDistribution` are uniformly mapped to γ in [0–π]; the first intensity value to 0, the last value to π, thus at least two values need to be present. If the array has a second dimension then the intensities are not rotational symmetric around `direction`, but are accordingly mapped to the C-halfplanes in [0–2π]; the first "row" of values to 0 and 2π, the other rows such that they have uniform distance to its neighbors. The orientation of the C0-plane is specified via `c0`. A combination of using an `intensityDistribution` and `OSP_INTENSITY_QUANTITY_POWER` as `intensityQuantity` is not supported at the moment.

[C-γ coordinate system for the mapping of `intensityDistribution` to the spotlight.][imgSpotCoords]

Quad Light

The quad[3] light is a planar, procedural area light source emitting uniformly on one side into the half-space. It is created by passing the type string "quad" to ospNewLight. The quad light supports OSP_INTENSITY_QUANTITY_POWER, OSP_INTENSITY_QUANTITY_INTENSITY and OSP_INTENSITY_QUANTITY_RADIANCE (default) as intensityQuantity parameter. In addition to the *general parameters* understood by all lights the quad light supports the following special parameters:

Table 51: Special parameters accepted by the quad light.

Type	Name	Description
vec3f	position	world-space position of one vertex of the quad light
vec3f	edge1	vector to one adjacent vertex
vec3f	edge2	vector to the other adjacent vertex

[Defining a quad light which emits toward the reader.][imgQuadLight]

The emission side is determined by the cross product of edge1×edge2. Note that only renderers that use stochastic sampling (like the path tracer) will compute soft shadows from the quad light. Other renderers will just sample the center of the quad light, which results in hard shadows.

HDRI Light

The HDRI light is a textured light source surrounding the scene and illuminating it from infinity. It is created by passing the type string "hdri" to ospNewLight. The HDRI light only accepts OSP_INTENSITY_QUANTITY_RADIANCE as intensityQuantity parameter value. In addition to the *general parameters* the HDRI light supports the following special parameters:

Table 52: Special parameters accepted by the HDRI light.

Type	Name	Description
vec3f	up	up direction of the light in world-space
vec3f	direction	direction to which the center of the texture will be mapped to (analog to *panoramic camera*)
OSPTexture	map	environment map in latitude / longitude format

[Orientation and Mapping of an HDRI Light.][imgHDRILight]

Note that the *SciVis renderer* only shows the HDRI light in the background (like an environment map) without computing illumination of the scene.

Ambient Light

The ambient light surrounds the scene and illuminates it from infinity with constant radiance (determined by combining the *parameters* `` `color` `` and intensity <#lights>`__). It is created by passing the type string "ambient" to ospNewLight. The ambient light supports OSP_INTENSITY_QUANTITY_RADIANCE and OSP_INTENSITY_QUANTITY_IRRADIANCE (default) as intensityQuantity parameter value.

Note that the *SciVis renderer* uses ambient lights to control the color and intensity of the computed ambient occlusion (AO).

[3] actually a parallelogram

Sun-Sky Light

The sun-sky light is a combination of a `distant` light for the sun and a procedural `hdri` light for the sky. It is created by passing the type string "`sunSky`" to `ospNewLight`. The sun-sky light surrounds the scene and illuminates it from infinity and can be used for rendering outdoor scenes. The radiance values are calculated using the Hošek-Wilkie sky model and solar radiance function. The sun-sky light only accepts `OSP_INTENSITY_QUANTITY_RADIANCE` as `intensityQuantity` parameter value. In addition to the *general parameters* the following special parameters are supported:

Table 53: Special parameters accepted by the `sunSky` light.

Type	Name	Default	Description
vec3f	up	$(0, 1, 0)$	zenith of sky in world-space
vec3f	direction	$(0, -1, 0)$	main emission direction of the sun
float	turbidity	3	atmospheric turbidity due to particles, in [1–10]
float	albedo	0.3	ground reflectance, in [0–1]
float	horizonExtension	0.01	extend the sky dome by stretching the horizon, fraction of the lower hemisphere to cover, in [0–1]

The lowest elevation for the sun is restricted to the horizon.

Note that the *SciVis renderer* only computes illumination from the sun (yet the sky is still shown in the background, like an environment map).

Emissive Objects

The *path tracer* will consider illumination by *geometries* which have a light emitting material assigned (for example the *Luminous* material).

Scene Hierarchy

Groups

Groups in OSPRay represent collections of GeometricModels and VolumetricModels which share a common local-space coordinate system. To create a group call

```
OSPGroup ospNewGroup();
```

Groups take arrays of geometric models, volumetric models and clipping geometric models, but they are optional. In other words, there is no need to create empty arrays if there are no geometries or volumes in the group.

By adding `OSPGeometricModels` to the `clippingGeometry` array a clipping geometry feature is enabled. Geometries assigned to this parameter will be used as clipping geometries. Any supported geometry can be used for clipping. The only requirement is that it has to distinctly partition space into clipping and non-clipping one. These include: spheres, boxes, infinite planes, closed meshes, closed subdivisions and curves. All geometries and volumes assigned to `geometry` or `volume` will be clipped. Use of clipping geometry that is not closed (or infinite) will result in rendering artifacts. User can decide which part of space is clipped by changing shading normals orientation with the `invertNormals` flag of the [GeometricModel]. When more than single clipping geometry is defined all clipping areas will be "added" together – an union of these areas will be applied.

Table 54: Parameters understood by groups.

Type	Name	Default	Description
OSPGeometricModel[]	geometry	NULL	*data* array of *GeometricModels*
OSPVolumetricModel[]	volume	NULL	*data* array of *VolumetricModels*
OSPGeometricModel[]	clippingGeometry	NULL	*data* array of *GeometricModels* used for clipping
bool	dynamicScene	false	use RTC_SCENE_DYNAMIC flag (faster BVH build, slower ray traversal), otherwise uses RTC_SCENE_STATIC flag (faster ray traversal, slightly slower BVH build)
bool	compactMode	false	tell Embree to use a more compact BVH in memory by trading ray traversal performance
bool	robustMode	false	tell Embree to enable more robust ray intersection code paths (slightly slower)

Note that groups only need to re re-committed if a geometry or volume changes (surface/scalar field representation). Appearance information on `OSPGeometricModel` and `OSPVolumetricModel` can be changed freely, as internal acceleration structures do not need to be reconstructed.

Instances

Instances in OSPRay represent a single group's placement into the world via a transform. To create and instance call

```
OSPInstance ospNewInstance(OSPGroup);
```

Table 55: Parameters understood by instances.

Type	Name	Default	Description
affine3f	xfm	identity	world-space transform for all attached geometries and volumes

World

Worlds are a container of scene data represented by *instances*. To create an (empty) world call

```
OSPWorld ospNewWorld();
```

Objects are placed in the world through an array of instances. Similar to *groups*, the array of instances is optional: there is no need to create empty arrays if there are no instances (though there will be nothing to render).

Applications can query the world (axis-aligned) bounding box after the world has been committed. To get this information, call

```
OSPBounds ospGetBounds(OSPObject);
```

The result is returned in the provided `OSPBounds`[4] struct:

[4] `OSPBounds` has essentially the same layout as the `OSP_BOX3F` `OSPDataType <#data>`__.

```
typedef struct {
    float lower[3];
    float upper[3];
} OSPBounds;
```

This call can also take `OSPGroup` and `OSPInstance` as well: all other object types will return an empty bounding box.

Finally, Worlds can be configured with parameters for making various feature/performance trade-offs (similar to groups).

Table 56: Parameters understood by worlds.

Type	Name	Default	Description
OSPInstance[]	instance	NULL	*data* array with handles of the *instances*
OSPLight[]	light	NULL	*data* array with handles of the *lights*
bool	dynamicScene	false	use RTC_SCENE_DYNAMIC flag (faster BVH build, slower ray traversal), otherwise uses RTC_SCENE_STATIC flag (faster ray traversal, slightly slower BVH build)
bool	compactMode	false	tell Embree to use a more compact BVH in memory by trading ray traversal performance
bool	robustMode	false	tell Embree to enable more robust ray intersection code paths (slightly slower)

Renderers

A renderer is the central object for rendering in OSPRay. Different renderers implement different features and support different materials. To create a new renderer of given type `type` use

```
OSPRenderer ospNewRenderer(const char *type);
```

General parameters of all renderers are

Table 57: Parameters understood by all renderers.

Type	Name	Default	Description
int	pixelSamples	1	samples per pixel
int	maxPathLength	20	maximum ray recursion depth
float	minContribution	0.001	sample contributions below this value will be neglected to speedup rendering
float	varianceThreshold	0	threshold for adaptive accumulation
float / vec3f / vec4f	backgroundColor	black, transparent	background color and alpha (RGBA), if no `map_backplate` is set
OSPTexture	map_backplate		optional *texture* image used as background (use texture type `texture2d`)
OSPTexture	map_maxDepth		optional screen-sized float *texture* with maximum far distance per pixel (use texture type `texture2d`)
OSPMaterial[]	material		optional *data* array of *materials* which can be indexed by a [GeometricModel]'s `material` parameter
uchar	pixelFilter	`OSP_PIXELFILTER_GAUSS`	`OSPPixelFilterType` to select the pixel filter used by the renderer for antialiasing. Possible pixel filters are listed below.

OSPRay's renderers support a feature called adaptive accumulation, which accelerates progressive *rendering* by stopping the rendering and refinement of image regions that have an estimated variance below the `varianceThreshold`. This feature requires a *framebuffer* with an `OSP_FB_VARIANCE` channel.

Per default the background of the rendered image will be transparent black, i.e., the alpha channel holds the opacity of the rendered objects. This eases transparency-aware blending of the image with an arbitrary background image by the application. The parameter `backgroundColor` or `map_backplate` can be used to already blend with a constant background color or backplate texture, respectively, (and alpha) during rendering.

OSPRay renderers support depth composition with images of other renderers, for example to incorporate help geometries of a 3D UI that were rendered with OpenGL. The screen-sized *texture* `map_maxDepth` must have format `OSP_TEXTURE_R32F` and flag `OSP_TEXTURE_FILTER_NEAREST`. The fetched values are used to limit the distance of primary rays, thus objects of other renderers can hide objects rendered by OSPRay.

OSPRay supports antialiasing in image space by using pixel filters, which are centered around the center of a pixel. The size ww of the filter depends on the selected filter type. The types of supported pixel filters are defined by the `OSPPixelFilterType` enum and can be set using the `pixelFilter` parameter.

Table 58: Pixel filter types supported by OSPRay for antialiasing in image space.

Name	Description
OSP_PIXELFILTER_POINT	a point filter only samples the center of the pixel, therefore the filter width is $w = 0$
OSP_PIXELFILTER_BOX	a uniform box filter with a width of $w = 1$
OSP_PIXELFILTER_GAUSS	a truncated, smooth Gaussian filter with a standard deviation of $\sigma = 0.5$ and a filter width of $w = 3$
OSP_PIXELFILTER_MITCHELL	the Mitchell-Netravali filter with a width of $w = 4$
OSP_PIXELFILTER_BLACKMAN_HARRIS	the Blackman-Harris filter with a width of $w = 3$

SciVis Renderer

The SciVis renderer is a fast ray tracer for scientific visualization which supports volume rendering and ambient occlusion (AO). It is created by passing the type string "`scivis`" to `ospNewRenderer`. In addition to the *general parameters* understood by all renderers, the SciVis renderer supports the following parameters:

Table 59: Special parameters understood by the SciVis renderer.

Type	Name	Default	Description
bool	shadows	false	whether to compute (hard) shadows
int	aoSamples	0	number of rays per sample to compute ambient occlusion
float	aoDistance	10^{20}	maximum distance to consider for ambient occlusion
float	volumeSamplingRate	1	sampling rate for volumes
bool	visibleLights	false	whether light sources are potentially visible (as in the *path tracer*, regarding each light's `visible`)

Note that the intensity (and color) of AO is deduced from an *ambient light* in the `lights` array.[5] If `aoSamples` is zero (the default) then ambient lights cause ambient illumination (without occlusion).

Ambient Occlusion Renderer

This renderer supports only a subset of the features of the *SciVis renderer* to gain performance. As the name suggest its main shading method is ambient occlusion (AO), *lights* are *not* considered at all and , Volume rendering is supported. The Ambient Occlusion renderer is created by passing the type string "`ao`" to `ospNewRenderer`. In addition to the *general parameters* understood by all renderers the following parameters are supported as well:

Table 60: Special parameters understood by the Ambient Occlusion renderer.

Type	Name	Default	Description
int	aoSamples	1	number of rays per sample to compute ambient occlusion
float	aoDistance	10^{20}	maximum distance to consider for ambient occlusion
float	aoIntensity	1	ambient occlusion strength
float	volumeSamplingRate	1	sampling rate for volumes

Path Tracer

The path tracer supports soft shadows, indirect illumination and realistic materials. This renderer is created by passing the type string "`pathtracer`" to `ospNewRenderer`. In addition to the *general parameters* understood by all renderers the path tracer supports the following special parameters:

[5] If there are multiple ambient lights then their contribution is added

Table 61: Special parameters understood by the path tracer.

Type	Name	Default	Description
int	lightSamples	all	number of random light samples per path vertex, per default all light sources are sampled
int	roulettePathLength	5	ray recursion depth at which to start Russian roulette termination
float	maxContribution	∞	samples are clamped to this value before they are accumulated into the framebuffer
bool	backgroundRefraction	false	allow for alpha blending even if background is seen through refractive objects like glass

The path tracer requires that *materials* are assigned to *geometries*, otherwise surfaces are treated as completely black.

The path tracer supports *volumes* with multiple scattering. The scattering albedo can be specified using the *transfer function*. Extinction is assumed to be spectrally constant.

Materials

Materials describe how light interacts with surfaces, they give objects their distinctive look. To let the given renderer create a new material of given type `type` call

```
OSPMaterial ospNewMaterial(const char *renderer_type, const char *material_type);
```

The returned handle can then be used to assign the material to a given geometry with

```
void ospSetObject(OSPGeometricModel, "material", OSPMaterial);
```

OBJ Material

The OBJ material is the workhorse material supported by both the *SciVis renderer* and the *path tracer* (the *Ambient Occlusion renderer* only uses the `kd` and `d` parameter). It offers widely used common properties like diffuse and specular reflection and is based on the MTL material format of Lightwave's OBJ scene files. To create an OBJ material pass the type string "`obj`" to `ospNewMaterial`. Its main parameters are

Table 62: Main parameters of the OBJ material.

Type	Name	Default	Description
vec3f	kd	white 0.8	diffuse color
vec3f	ks	black	specular color
float	ns	10	shininess (Phong exponent), usually in [2–10:sup:*4*]
float	d	opaque	opacity
vec3f	tf	black	transparency filter color
OSPTexture	map_bump	NULL	normal map

In particular when using the path tracer it is important to adhere to the principle of energy conservation, i.e., that the amount of light reflected by a surface is not larger than the light arriving. Therefore the path tracer issues a warning and renormalizes the color parameters if the sum of `Kd`, `Ks`, and `Tf` is larger than one in any color channel. Similarly important to mention is that almost all materials of the real world reflect at most only about 80% of the incoming light. So even for a white sheet of paper or white wall paint do better not set `Kd` larger than 0.8; otherwise rendering times

are unnecessary long and the contrast in the final images is low (for example, the corners of a white room would hardly be discernible, as can be seen in the figure below).

[Comparison of diffuse rooms with 100% reflecting white paint (left) and realistic 80% reflecting white paint (right), which leads to higher overall contrast. Note that exposure has been adjusted to achieve similar brightness levels.][imgDiffuseRooms]

If present, the color component of *geometries* is also used for the diffuse color `Kd` and the alpha component is also used for the opacity `d`.

Normal mapping can simulate small geometric features via the texture `map_Bump`. The normals n in the normal map are with respect to the local tangential shading coordinate system and are encoded as $(n+1)$, thus a texel $(0.5, 0.5, 1)$[6] represents the unperturbed shading normal $(0, 0, 1)$. Because of this encoding an sRGB gamma *texture* format is ignored and normals are always fetched as linear from a normal map. Note that the orientation of normal maps is important for a visually consistent look: by convention OSPRay uses a coordinate system with the origin in the lower left corner; thus a convexity will look green toward the top of the texture image (see also the example image of a normal map). If this is not the case flip the normal map vertically or invert its green channel.

[Normal map representing an exalted square pyramidal frustum.][imgNormalMap]

Note that `Tf` colored transparency is implemented in the SciVis and the path tracer but normal mapping with `map_Bump` is currently supported in the path tracer only.

All parameters (except `Tf`) can be textured by passing a *texture* handle, prefixed with "`map_`". The fetched texels are multiplied by the respective parameter value. If only the texture is given (but not the corresponding parameter), only the texture is used (the default value of the parameter is *not* multiplied). The color textures `map_Kd` and `map_Ks` are typically in one of the sRGB gamma encoded formats, whereas textures `map_Ns` and `map_d` are usually in a linear format (and only the first component is used). Additionally, all textures support *texture transformations*.

[Rendering of a OBJ material with wood textures.][imgMaterialOBJ]

Principled

The Principled material is the most complex material offered by the *path tracer*, which is capable of producing a wide variety of materials (e.g., plastic, metal, wood, glass) by combining multiple different layers and lobes. It uses the GGX microfacet distribution with approximate multiple scattering for dielectrics and metals, uses the Oren-Nayar model for diffuse reflection, and is energy conserving. To create a Principled material, pass the type string "`principled`" to `ospNewMaterial`. Its parameters are listed in the table below.

[6] respectively $(127, 127, 255)$ for 8 bit textures and $(32767, 32767, 65535)$ for 16 bit textures

Table 63: Parameters of the Principled material.

Type	Name	Default	Description
vec3f	baseColor	white 0.8	base reflectivity (diffuse and/or metallic)
vec3f	edgeColor	white	edge tint (metallic only)
float	metallic	0	mix between dielectric (diffuse and/or specular) and metallic (specular only with complex IOR) in [0–1]
float	diffuse	1	diffuse reflection weight in [0–1]
float	specular	1	specular reflection/transmission weight in [0–1]
float	ior	1	dielectric index of refraction
float	transmission	0	specular transmission weight in [0–1]
vec3f	transmissionColor	white	attenuated color due to transmission (Beer's law)
float	transmissionDepth	1	distance at which color attenuation is equal to transmissionColor
float	roughness	0	diffuse and specular roughness in [0–1], 0 is perfectly smooth
float	anisotropy	0	amount of specular anisotropy in [0–1]
float	rotation	0	rotation of the direction of anisotropy in [0–1], 1 is going full circle
float	normal	1	default normal map/scale for all layers
float	baseNormal	1	base normal map/scale (overrides default normal)
bool	thin	false	flag specifying whether the material is thin or solid
float	thickness	1	thickness of the material (thin only), affects the amount of color attenuation due to specular transmission
float	backlight	0	amount of diffuse transmission (thin only) in [0–2], 1 is 50% reflection and 50% transmission, 2 is transmission only
float	coat	0	clear coat layer weight in [0–1]
float	coatIor	1.5	clear coat index of refraction
vec3f	coatColor	white	clear coat color tint
float	coatThickness	1	clear coat thickness, affects the amount of color attenuation
float	coatRoughness	0	clear coat roughness in [0–1], 0 is perfectly smooth
float	coatNormal	1	clear coat normal map/scale (overrides default normal)
float	sheen	0	sheen layer weight in [0–1]
vec3f	sheenColor	white	sheen color tint
float	sheenTint	0	how much sheen is tinted from sheenColor toward baseColor
float	sheenRoughness	0.2	sheen roughness in [0–1], 0 is perfectly smooth
float	opacity	1	cut-out opacity/transparency, 1 is fully opaque

All parameters can be textured by passing a *texture* handle, prefixed with "`map_`" (e.g., "`map_baseColor`"). *texture transformations* are supported as well.

[Rendering of a Principled coated brushed metal material with textured anisotropic rotation and a dust layer (sheen) on top.][imgMaterialPrincipled]

CarPaint

The CarPaint material is a specialized version of the Principled material for rendering different types of car paints. To create a CarPaint material, pass the type string "`carPaint`" to `ospNewMaterial`. Its parameters are listed in the table below.

Table 64: Parameters of the CarPaint material.

Type	Name	Default	Description
vec3f	baseColor	white 0.8	diffuse base reflectivity
float	roughness	0	diffuse roughness in [0–1], 0 is perfectly smooth
float	normal	1	normal map/scale
vec3f float	flakeColor flakeDensity	Aluminium 0	color of metallic flakes density of metallic flakes in [0–1], 0 disables flakes, 1 fully covers the surface with flakes
float	flakeScale	100	scale of the flake structure, higher values increase the amount of flakes
float	flakeSpread	0.3	flake spread in [0–1]
float	flakeJitter	0.75	flake randomness in [0–1]
float	flakeRoughness	0.3	flake roughness in [0–1], 0 is perfectly smooth
float	coat	1	clear coat layer weight in [0–1]
float	coatIor	1.5	clear coat index of refraction
vec3f	coatColor	white	clear coat color tint
float	coatThickness	1	clear coat thickness, affects the amount of color attenuation
float	coatRoughness	0	clear coat roughness in [0–1], 0 is perfectly smooth
float	coatNormal	1	clear coat normal map/scale
vec3f	flipflopColor	white	reflectivity of coated flakes at grazing angle, used together with coatColor produces a pearlescent paint
float	flipflopFalloff	1	flip flop color falloff, 1 disables the flip flop effect

All parameters can be textured by passing a *texture* handle, prefixed with "`map_`" (e.g., "`map_baseColor`"). *texture transformations* are supported as well.

[Rendering of a pearlescent CarPaint material.][imgMaterialCarPaint]

Metal

The *path tracer* offers a physical metal, supporting changing roughness and realistic color shifts at edges. To create a Metal material pass the type string "`metal`" to `ospNewMaterial`. Its parameters are

Table 65: Parameters of the Metal material.

Type	Name	Default	Description
vec3f[]	ior	Aluminium	*data* array of spectral samples of complex refractive index, each entry in the form (wavelength, eta, k), ordered by wavelength (which is in nm)
vec3f	eta		RGB complex refractive index, real part
vec3f	k		RGB complex refractive index, imaginary part
float	roughness	0.1	roughness in [0–1], 0 is perfect mirror

The main appearance (mostly the color) of the Metal material is controlled by the physical parameters `eta` and `k`, the

wavelength-dependent, complex index of refraction. These coefficients are quite counter-intuitive but can be found in published measurements. For accuracy the index of refraction can be given as an array of spectral samples in `ior`, each sample a triplet of wavelength (in nm), eta, and k, ordered monotonically increasing by wavelength; OSPRay will then calculate the Fresnel in the spectral domain. Alternatively, `eta` and `k` can also be specified as approximated RGB coefficients; some examples are given in below table.

Table 66: Index of refraction of selected metals as approximated RGB coefficients, based on data from https://refractiveindex.info/.

Metal	eta	k
Ag, Silver	(0.051, 0.043, 0.041)	(5.3, 3.6, 2.3)
Al, Aluminium	(1.5, 0.98, 0.6)	(7.6, 6.6, 5.4)
Au, Gold	(0.07, 0.37, 1.5)	(3.7, 2.3, 1.7)
Cr, Chromium	(3.2, 3.1, 2.3)	(3.3, 3.3, 3.1)
Cu, Copper	(0.1, 0.8, 1.1)	(3.5, 2.5, 2.4)

The `roughness` parameter controls the variation of microfacets and thus how polished the metal will look. The roughness can be modified by a *texture* `map_roughness` (*texture transformations* are supported as well) to create notable edging effects.

[Rendering of golden Metal material with textured roughness.][imgMaterialMetal]

Alloy

The *path tracer* offers an alloy material, which behaves similar to *Metal*, but allows for more intuitive and flexible control of the color. To create an Alloy material pass the type string "`alloy`" to `ospNewMaterial`. Its parameters are

Table 67: Parameters of the Alloy material.

Type	Name	Default	Description
vec3f	color	white 0.9	reflectivity at normal incidence (0 degree)
vec3f	edgeColor	white	reflectivity at grazing angle (90 degree)
float	roughness	0.1	roughness, in [0–1], 0 is perfect mirror

The main appearance of the Alloy material is controlled by the parameter `color`, while `edgeColor` influences the tint of reflections when seen at grazing angles (for real metals this is always 100% white). If present, the color component of *geometries* is also used for reflectivity at normal incidence `color`. As in *Metal* the `roughness` parameter controls the variation of microfacets and thus how polished the alloy will look. All parameters can be textured by passing a *texture* handle, prefixed with "`map_`"; *texture transformations* are supported as well.

[Rendering of a fictional Alloy material with textured color.][imgMaterialAlloy]

Glass

The *path tracer* offers a realistic a glass material, supporting refraction and volumetric attenuation (i.e., the transparency color varies with the geometric thickness). To create a Glass material pass the type string "`glass`" to `ospNewMaterial`. Its parameters are

Table 68: Parameters of the Glass material.

Type	Name	Default	Description
float	eta	1.5	index of refraction
vec3f	attenuationColor	white	resulting color due to attenuation
float	attenuationDistance	1	distance affecting attenuation

For convenience, the rather counter-intuitive physical attenuation coefficients will be calculated from the user inputs in such a way, that the `attenuationColor` will be the result when white light traveled trough a glass of thickness `attenuationDistance`.

[Rendering of a Glass material with orange attenuation.][imgMaterialGlass]

ThinGlass

The *path tracer* offers a thin glass material useful for objects with just a single surface, most prominently windows. It models a thin, transparent slab, i.e., it behaves as if a second, virtual surface is parallel to the real geometric surface. The implementation accounts for multiple internal reflections between the interfaces (including attenuation), but neglects parallax effects due to its (virtual) thickness. To create a such a thin glass material pass the type string "`thinGlass`" to `ospNewMaterial`. Its parameters are

Table 69: Parameters of the ThinGlass material.

Type	Name	Default	Description
float	eta	1.5	index of refraction
vec3f	attenuationColor	white	resulting color due to attenuation
float	attenuationDistance	1	distance affecting attenuation
float	thickness	1	virtual thickness

For convenience the attenuation is controlled the same way as with the *Glass* material. Additionally, the color due to attenuation can be modulated with a *texture* `map_attenuationColor` (*texture transformations* are supported as well). If present, the color component of *geometries* is also used for the attenuation color. The `thickness` parameter sets the (virtual) thickness and allows for easy exchange of parameters with the (real) *Glass* material; internally just the ratio between `attenuationDistance` and `thickness` is used to calculate the resulting attenuation and thus the material appearance.

[Rendering of a ThinGlass material with red attenuation.][imgMaterialThinGlass]

[Example image of a colored window made with textured attenuation of the ThinGlass material.][imgColoredWindow]

MetallicPaint

The *path tracer* offers a metallic paint material, consisting of a base coat with optional flakes and a clear coat. To create a MetallicPaint material pass the type string "`metallicPaint`" to `ospNewMaterial`. Its parameters are listed in the table below.

Table 70: Parameters of the MetallicPaint material.

Type	Name	Default	Description
vec3f	baseColor	white 0.8	color of base coat
float	flakeAmount	0.3	amount of flakes, in [0–1]
vec3f	flakeColor	Aluminium	color of metallic flakes
float	flakeSpread	0.5	spread of flakes, in [0–1]
float	eta	1.5	index of refraction of clear coat

The color of the base coat `baseColor` can be textured by a *texture* `map_baseColor`, which also supports *texture transformations*. If present, the color component of *geometries* is also used for the color of the base coat. Parameter `flakeAmount` controls the proportion of flakes in the base coat, so when setting it to 1 the `baseColor` will not be visible. The shininess of the metallic component is governed by `flakeSpread`, which controls the variation of the orientation of the flakes, similar to the `roughness` parameter of *Metal*. Note that the effect of the metallic flakes is currently only computed on average, thus individual flakes are not visible.

[Rendering of a MetallicPaint material.][imgMaterialMetallicPaint]

Luminous

The *path tracer* supports the Luminous material which emits light uniformly in all directions and which can thus be used to turn any geometric object into a light source[7]. It is created by passing the type string "`luminous`" to `ospNewMaterial`. The amount of constant radiance that is emitted is determined by combining the general parameters of lights: `color and intensity <#lights>`__ (which essentially means that parameter `intensityQuantity` is not needed because it is always `OSP_INTENSITY_QUANTITY_RADIANCE`).

Table 71: Parameters accepted by the Luminous material.

Type	Name	Default	Description
vec3f	color	white	color of the emitted light
float	intensity	1	intensity of the light (a factor)
float	transparency	1	material transparency

[Rendering of a yellow Luminous material.][imgMaterialLuminous]

Texture

OSPRay currently implements two texture types (`texture2d` and `volume`) and is open for extension to other types by applications. More types may be added in future releases.

To create a new texture use

```
OSPTexture ospNewTexture(const char *type);
```

Texture2D

The `texture2d` texture type implements an image-based texture, where its parameters are as follows

Table 72: Parameters of `texture2d` texture type.

Type	Name	Description
int	format	`OSPTextureFormat` for the texture
int	filter	default `OSP_TEXTURE_FILTER_BILINEAR`, alternatively `OSP_TEXTURE_FILTER_NEAREST`
OSPData	data	the actual texel 2D *data*

The supported texture formats for `texture2d` are:

[7] If `geometryLights` is enabled in the *path tracer*.

Table 73: Supported texture formats by `texture2d`, i.e., valid constants of type `OSPTextureFormat`.

Name	Description
OSP_TEXTURE_RGBA8	8 bit [0–255] linear components red, green, blue, alpha
OSP_TEXTURE_SRGBA	8 bit sRGB gamma encoded color components, and linear alpha
OSP_TEXTURE_RGBA32F	32 bit float components red, green, blue, alpha
OSP_TEXTURE_RGB8	8 bit [0–255] linear components red, green, blue
OSP_TEXTURE_SRGB	8 bit sRGB gamma encoded components red, green, blue
OSP_TEXTURE_RGB32F	32 bit float components red, green, blue
OSP_TEXTURE_R8	8 bit [0–255] linear single component red
OSP_TEXTURE_RA8	8 bit [0–255] linear two components red, alpha
OSP_TEXTURE_L8	8 bit [0–255] gamma encoded luminance (replicated into red, green, blue)
OSP_TEXTURE_LA8	8 bit [0–255] gamma encoded luminance, and linear alpha
OSP_TEXTURE_R32F	32 bit float single component red
OSP_TEXTURE_RGBA16	16 bit [0–65535] linear components red, green, blue, alpha
OSP_TEXTURE_RGB16	16 bit [0–65535] linear components red, green, blue
OSP_TEXTURE_RA16	16 bit [0–65535] linear two components red, alpha
OSP_TEXTURE_R16	16 bit [0–65535] linear single component red

The size of the texture is inferred from the size of the 2D array `data`, which also needs have a compatible type to `format`. The texel data in `data` starts with the texels in the lower left corner of the texture image, like in OpenGL. Per default a texture fetch is filtered by performing bi-linear interpolation of the nearest 2×2 texels; if instead fetching only the nearest texel is desired (i.e., no filtering) then pass the `OSP_TEXTURE_FILTER_NEAREST` flag.

Texturing with `texture2d` image textures requires *geometries* with texture coordinates, e.g., a *mesh* with `vertex.texcoord` provided.

Volume Texture

The `volume` texture type implements texture lookups based on 3D object coordinates of the surface hit point on the associated geometry. If the given hit point is within the attached volume, the volume is sampled and classified with the transfer function attached to the volume. This implements the ability to visualize volume values (as colored by a transfer function) on arbitrary surfaces inside the volume (as opposed to an isosurface showing a particular value in the volume). Its parameters are as follows

Table 74: Parameters of `volume` texture type.

Type	Name	Description
OSPVolume	volume	[Volume] used to generate color lookups
OSPTransferFunction	transferFunction	[TransferFunction] applied to `volume`

TextureVolume can be used for implementing slicing of volumes with any geometry type. It enables coloring of the slicing geometry with a different transfer function than that of the sliced volume.

Texture Transformations

All materials with textures also offer to manipulate the placement of these textures with the help of texture transformations. If so, this convention shall be used: the following parameters are prefixed with "`texture_name.`").

Table 75: Parameters to define 2D texture coordinate transformations.

Type	Name	Description
linear2f	transform	linear transformation (rotation, scale)
float	rotation	angle in degree, counterclockwise, around center
vec2f	scale	enlarge texture, relative to center (0.5, 0.5)
vec2f	translation	move texture in positive direction (right/up)

Above parameters are combined into a single `affine2d` transformation matrix and the transformations are applied in the given order. Rotation, scale and translation are interpreted "texture centric", i.e., their effect seen by an user are relative to the texture (although the transformations are applied to the texture coordinates).

Table 76: Parameter to define 3D volume texture transformations.

Type	Name	Description
affine3f	transform	linear transformation (rotation, scale) plus translation

Similarly, volume texture placement can also be modified by an `affine3f` transformation matrix.

Cameras

To create a new camera of given type `type` use

```
OSPCamera ospNewCamera(const char *type);
```

All cameras accept these parameters:

Table 77: Parameters accepted by all cameras.

Type	Name	Description
vec3f	position	position of the camera in world-space
vec3f	direction	main viewing direction of the camera
vec3f	up	up direction of the camera
float	nearClip	near clipping distance
vec2f	imageStart	start of image region (lower left corner)
vec2f	imageEnd	end of image region (upper right corner)

The camera is placed and oriented in the world with `position`, `direction` and `up`. OSPRay uses a right-handed coordinate system. The region of the camera sensor that is rendered to the image can be specified in normalized screen-space coordinates with `imageStart` (lower left corner) and `imageEnd` (upper right corner). This can be used, for example, to crop the image, to achieve asymmetrical view frusta, or to horizontally flip the image to view scenes which are specified in a left-handed coordinate system. Note that values outside the default range of [0–1] are valid, which is useful to easily realize overscan or film gate, or to emulate a shifted sensor.

Perspective Camera

The perspective camera implements a simple thin lens camera for perspective rendering, supporting optionally depth of field and stereo rendering, but no motion blur. It is created by passing the type string "perspective" to ospNewCamera. In addition to the *general parameters* understood by all cameras the perspective camera supports the special parameters listed in the table below.

Table 78: Additional parameters accepted by the perspective camera.

Type	Name	Description
float	fovy	the field of view (angle in degree) of the frame's height
float	aspect	ratio of width by height of the frame (and image region)
float	apertureRadius	size of the aperture, controls the depth of field
float	focusDistance	distance at where the image is sharpest when depth of field is enabled
bool	architectural	vertical edges are projected to be parallel
uchar	stereoMode	OSPStereoMode for stereo rendering, possible values are:
		OSP_STEREO_NONE (default)
		OSP_STEREO_LEFT
		OSP_STEREO_RIGHT
		OSP_STEREO_SIDE_BY_SIDE
		OSP_STEREO_TOP_BOTTOM (left eye at top half)
float	interpupillaryDistance	distance between left and right eye when stereo is enabled, default 0.0635

Note that when computing the aspect ratio a potentially set image region (using imageStart & imageEnd) needs to be regarded as well.

In architectural photography it is often desired for aesthetic reasons to display the vertical edges of buildings or walls vertically in the image as well, regardless of how the camera is tilted. Enabling the architectural mode achieves this by internally leveling the camera parallel to the ground (based on the up direction) and then shifting the lens such that the objects in direction dir are centered in the image. If finer control of the lens shift is needed use imageStart & imageEnd. Because the camera is now effectively leveled its image plane and thus the plane of focus is oriented parallel to the front of buildings, the whole façade appears sharp, as can be seen in the example images below. The resolution of the *framebuffer* is not altered by imageStart/imageEnd.

[Example image created with the perspective camera, featuring depth of field.][imgCameraPerspective]

[Enabling the architectural flag corrects the perspective projection distortion, resulting in parallel vertical edges.][imgCameraArchitectural]

[Example 3D stereo image using stereoMode = OSP_STEREO_SIDE_BY_SIDE.][imgCameraStereo]

Orthographic Camera

The orthographic camera implements a simple camera with orthographic projection, without support for depth of field or motion blur. It is created by passing the type string "orthographic" to ospNewCamera. In addition to the *general parameters* understood by all cameras the orthographic camera supports the following special parameters:

Table 79: Additional parameters accepted by the orthographic camera.

Type	Name	Description
float	height	size of the camera's image plane in y, in world coordinates
float	aspect	ratio of width by height of the frame

For convenience the size of the camera sensor, and thus the extent of the scene that is captured in the image, can be controlled with the height parameter. The same effect can be achieved with imageStart and imageEnd, and

both methods can be combined. In any case, the `aspect` ratio needs to be set accordingly to get an undistorted image.

[Example image created with the orthographic camera.][imgCameraOrthographic]

Panoramic Camera

The panoramic camera implements a simple camera with support for stereo rendering. It captures the complete surrounding with a latitude / longitude mapping and thus the rendered images should best have a ratio of 2:1. A panoramic camera is created by passing the type string "`panoramic`" to `ospNewCamera`. It is placed and oriented in the scene by using the *general parameters* understood by all cameras.

Table 80: Additional parameters accepted by the panoramic camera.

Type	Name	Description
uchar	stereoMode	`OSPStereoMode` for stereo rendering, possible values are:
		`OSP_STEREO_NONE` (default)
		`OSP_STEREO_LEFT`
		`OSP_STEREO_RIGHT`
		`OSP_STEREO_SIDE_BY_SIDE`
		`OSP_STEREO_TOP_BOTTOM` (left eye at top half)
float	interpupillaryDistance	distance between left and right eye when stereo is enabled, default 0.0635

[Latitude / longitude map created with the panoramic camera.][imgCameraPanoramic]

Picking

To get the world-space position of the geometry (if any) seen at [0–1] normalized screen-space pixel coordinates `screenPos_x` and `screenPos_y` use

```
void ospPick(OSPPickResult *,
    OSPFrameBuffer,
    OSPRenderer,
    OSPCamera,
    OSPWorld,
    float screenPos_x,
    float screenPos_y);
```

The result is returned in the provided `OSPPickResult` struct:

```
typedef struct {
    int hasHit;
    float worldPosition[3];
    OSPInstance instance;
    OSPGeometricModel model;
    uint32_t primID;
} OSPPickResult;
```

Note that `ospPick` considers exactly the same camera of the given renderer that is used to render an image, thus matching results can be expected. If the camera supports depth of field then the center of the lens and thus the center of the circle of confusion is used for picking. Note that the caller needs to `ospRelease` the `instance` and `model` handles of `OSPPickResult` once the information is not needed anymore.

Framebuffer

The framebuffer holds the rendered 2D image (and optionally auxiliary information associated with pixels). To create a new framebuffer object of given size `size` (in pixels), color format, and channels use

```
OSPFrameBuffer ospNewFrameBuffer(int size_x, int size_y,
    OSPFrameBufferFormat format = OSP_FB_SRGBA,
    uint32_t frameBufferChannels = OSP_FB_COLOR);
```

The parameter `format` describes the format the color buffer has *on the host*, and the format that `ospMapFrameBuffer` will eventually return. Valid values are:

Table 81: Supported color formats of the framebuffer that can be passed to `ospNewFrameBuffer`, i.e., valid constants of type `OSPFrameBufferFormat`.

Name	Description
OSP_FB_NONE	framebuffer will not be mapped by the application
OSP_FB_RGBA8	8 bit [0–255] linear component red, green, blue, alpha
OSP_FB_SRGBA	8 bit sRGB gamma encoded color components, and linear alpha
OSP_FB_RGBA32F	32 bit float components red, green, blue, alpha

The parameter `frameBufferChannels` specifies which channels the framebuffer holds, and can be combined together by bitwise OR from the values of `OSPFrameBufferChannel` listed in the table below.

Table 82: Framebuffer channels constants (of type `OSPFrameBufferChannel`), naming optional information the framebuffer can store. These values can be combined by bitwise OR when passed to `ospNewFrameBuffer`.

Name	Description
OSP_FB_COLOR	RGB color including alpha
OSP_FB_DEPTH	euclidean distance to the camera (*not* to the image plane), as linear 32 bit float; for multiple samples per pixel their minimum is taken
OSP_FB_ACCUM	accumulation buffer for progressive refinement
OSP_FB_VARIANCE	for estimation of the current noise level if OSP_FB_ACCUM is also present, see *rendering*
OSP_FB_NORMAL	accumulated world-space normal of the first hit, as vec3f
OSP_FB_ALBEDO	accumulated material albedo (color without illumination) at the first hit, as vec3f

If a certain channel value is *not* specified, the given buffer channel will not be present. Note that OSPRay makes a clear distinction between the *external* format of the framebuffer and the internal one: The external format is the format the user specifies in the `format` parameter; it specifies what color format OSPRay will eventually *return* the framebuffer to the application (when calling `ospMapFrameBuffer`): no matter what OSPRay uses internally, it will simply return a 2D array of pixels of that format, with possibly all kinds of reformatting, compression/decompression, etc., going on in-between the generation of the *internal* framebuffer and the mapping of the externally visible one.

In particular, `OSP_FB_NONE` is a perfectly valid pixel format for a framebuffer that an application will never map. For example, an application driving a display wall may well generate an intermediate framebuffer and eventually transfer its pixel to the individual displays using an `OSPImageOperation` *image operation*.

The application can map the given channel of a framebuffer – and thus access the stored pixel information – via

```
const void *ospMapFrameBuffer(OSPFrameBuffer, OSPFrameBufferChannel = OSP_FB_COLOR);
```

Note that `OSP_FB_ACCUM` or `OSP_FB_VARIANCE` cannot be mapped. The origin of the screen coordinate system in OSPRay is the lower left corner (as in OpenGL), thus the first pixel addressed by the returned pointer is the lower

left pixel of the image.

A previously mapped channel of a framebuffer can be unmapped by passing the received pointer `mapped` to

```
void ospUnmapFrameBuffer(const void *mapped, OSPFrameBuffer);
```

The individual channels of a framebuffer can be cleared with

```
void ospResetAccumulation(OSPFrameBuffer);
```

This function will clear *all* accumulating buffers (`OSP_FB_VARIANCE`, `OSP_FB_NORMAL`, and `OSP_FB_ALBEDO`, if present) and resets the accumulation counter `accumID`. It is unspecified if the existing color and depth buffers are physically cleared when `ospResetAccumulation` is called.

If `OSP_FB_VARIANCE` is specified, an estimate of the variance of the last accumulated frame can be queried with

```
float ospGetVariance(OSPFrameBuffer);
```

Note this value is only updated after synchronizing with `OSP_FRAME_FINISHED`, as further described in *asynchronous rendering*. The estimated variance can be used by the application as a quality indicator and thus to decide whether to stop or to continue progressive rendering.

The framebuffer takes a list of pixel operations to be applied to the image in sequence as an `OSPData`. The pixel operations will be run in the order they are in the array.

Table 83: Parameters accepted by the framebuffer.

Type	Name	Description
OSPImageOperation[]	imageOperation	ordered sequence of image operations

Image Operation

Image operations are functions that are applied to every pixel of a frame. Examples include post-processing, filtering, blending, tone mapping, or sending tiles to a display wall. To create a new pixel operation of given type `type` use

```
OSPImageOperation ospNewImageOperation(const char *type);
```

Tone Mapper

The tone mapper is a pixel operation which implements a generic filmic tone mapping operator. Using the default parameters it approximates the Academy Color Encoding System (ACES). The tone mapper is created by passing the type string "`tonemapper`" to `ospNewImageOperation`. The tone mapping curve can be customized using the parameters listed in the table below.

Table 84: Parameters accepted by the tone mapper.

Type	Name	Default	Description
float	exposure	1.0	amount of light per unit area
float	contrast	1.6773	contrast (toe of the curve); typically is in [1–2]
float	shoulder	0.9714	highlight compression (shoulder of the curve); typically is in [0.9–1]
float	midIn	0.18	mid-level anchor input; default is 18% gray
float	midOut	0.18	mid-level anchor output; default is 18% gray
float	hdrMax	11.0785	maximum HDR input that is not clipped
bool	acesColor	true	apply the ACES color transforms

To use the popular "Uncharted 2" filmic tone mapping curve instead, set the parameters to the values listed in the table below.

Table 85: Filmic tone mapping curve parameters. Note that the curve includes an exposure bias to match 18% middle gray.

Name	Value
contrast	1.1759
shoulder	0.9746
midIn	0.18
midOut	0.18
hdrMax	6.3704
acesColor	false

Denoiser

OSPRay comes with a module that adds support for Intel® Open Image Denoise. This is provided as an optional module as it creates an additional project dependency at compile time. The module implements a "`denoiser`" frame operation, which denoises the entire frame before the frame is completed.

Rendering

Asynchronous Rendering

Rendering is by default asynchronous (non-blocking), and is done by combining a framebuffer, renderer, camera, and world.

What to render and how to render it depends on the renderer's parameters. If the framebuffer supports accumulation (i.e., it was created with `OSP_FB_ACCUM`) then successive calls to `ospRenderFrame` will progressively refine the rendered image.

To start an render task, use

```
OSPFuture ospRenderFrame(OSPFrameBuffer, OSPRenderer, OSPCamera, OSPWorld);
```

This returns an `OSPFuture` handle, which can be used to synchronize with the application, cancel, or query for progress of the running task. When `ospRenderFrame` is called, there is no guarantee when the associated task will begin execution.

Progress of a running frame can be queried with the following API function

```
float ospGetProgress(OSPFuture);
```

This returns the approximated progress of the task in [0-1].

Applications can cancel a currently running asynchronous operation via

```
void ospCancel(OSPFuture);
```

Applications can wait on the result of an asynchronous operation, or choose to only synchronize with a specific event. To synchronize with an `OSPFuture` use

```
void ospWait(OSPFuture, OSPSyncEvent = OSP_TASK_FINISHED);
```

The following are values which can be synchronized with the application

Table 86: Supported events that can be passed to `ospWait`.

Name	Description
OSP_NONE_FINISHED	Do not wait for anything to be finished (immediately return from `ospWait`)
OSP_WORLD_COMMITTED	Wait for the world to be committed (not yet implemented)
OSP_WORLD_RENDERED	Wait for the world to be rendered, but not post-processing operations (Pixel/Tile/Frame Op)
OSP_FRAME_FINISHED	Wait for all rendering operations to complete
OSP_TASK_FINISHED	Wait on full completion of the task associated with the future. The underlying task may involve one or more of the above synchronization events

Currently only rendering can be invoked asynchronously. However, future releases of OSPRay may add more asynchronous versions of API calls (and thus return `OSPFuture`).

Applications can query whether particular events are complete with

```
int ospIsReady(OSPFuture, OSPSyncEvent = OSP_TASK_FINISHED);
```

As the given running task runs (as tracked by the `OSPFuture`), applications can query a boolean [0,1] result if the passed event has been completed.

Applications can query how long an async task ran with

```
float ospGetTaskDuration(OSPFuture);
```

This returns the wall clock execution time of the task in seconds. If the task is still running, this will block until the task is completed. This is useful for applications to query exactly how long an asynchronous task executed without the overhead of measuring both task execution + synchronization by the calling application.

Asynchronously Rendering and ospCommit()

The use of either `ospRenderFrame` or `ospRenderFrame` requires that all objects in the scene being rendered have been committed before rendering occurs. If a call to `ospCommit()` happens while a frame is rendered, the result is undefined behavior and should be avoided.

Synchronous Rendering

For convenience in certain use cases, `ospray_util.h` provides a synchronous version of `ospRenderFrame`:

```
float ospRenderFrameBlocking(OSPFrameBuffer, OSPRenderer, OSPCamera, OSPWorld);
```

This version is the equivalent of:

```
ospRenderFrame
ospWait(f, OSP_TASK_FINISHED)
return ospGetVariance(fb)
```

This version is closest to `ospRenderFrame` from OSPRay v1.x.

Distributed rendering with MPI

The purpose of the MPI module for OSPRay is to provide distributed rendering capabilities for OSPRay. The module enables image- and data-parallel rendering across HPC clusters using MPI, allowing applications to transparently distribute rendering work, or to render data sets which are too large to fit in memory on a single machine.

The MPI module provides two OSPRay devices to allow applications to leverage distributed rendering capabilities. The `mpiOffload` device provides transparent image-parallel rendering, where the same OSPRay application written for local rendering can be replicated across multiple nodes to distribute the rendering work. The `mpiDistributed` device allows MPI distributed applications to use OSPRay for distributed rendering, where each rank can render and independent piece of a global data set, or hybrid rendering where ranks partially or completely share data.

MPI Offload Rendering

The `mpiOffload` device can be used to distribute image rendering tasks across a cluster without requiring modifications to the application itself. Existing applications using OSPRay for local rendering simply be passed command line arguments to load the module and indicate that the `mpiOffload` device should be used for image-parallel rendering. To load the module, pass `--osp:load-modules=mpi`, to select the MPIOffloadDevice, pass `--osp:device=mpiOffload`. For example, the `ospExamples` application can be run as:

```
mpirun -n <N> ./ospExamples --osp:load-modules=mpi --osp:device=mpiOffload
```

and will automatically distribute the image rendering tasks among the corresponding `N` nodes. Note that in this configuration rank 0 will act as a master/application rank, and will run the user application code but not perform rendering locally. Thus, a minimum of 2 ranks are required, one master to run the application and one worker to perform the rendering. Running with 3 ranks for example would now distribute half the image rendering work to rank 1 and half to rank 2.

If more control is required over the placement of ranks to nodes, or you want to run a worker rank on the master node as well you can run the application and the `ospray_mpi_worker` program through MPI's MPMD mode. The `ospray_mpi_worker` will load the MPI module and select the offload device by default.

```
mpirun -n 1 ./ospExamples --osp:load-modules=mpi --osp:device=mpiOffload \
  : -n <N> ./ospray_mpi_worker
```

If initializing the `mpiOffload` device manually, or passing parameters through the command line, the following parameters can be set:

Table 87: Parameters specific to the `mpiOffload` Device.

Type	Name	Default	Description
string	mpiMode	mpi	The mode to communicate with the worker ranks. `mpi` will assume you're launching the application and workers in the same mpi command (or split launch command). `mpi` is the only supported mode
uint	maxCommandBufferEntries	8192	Set the max number of commands to buffer before submitting the command buffer to the workers
uint	commandBufferSize	512 MiB	Set the max command buffer size to allow. Units are in MiB. Max size is 1.8GiB
uint	maxInlineDataSize	32 MiB	Set the max size of an OSPData which can be inline'd into the command buffer instead of being sent separately. Max size is half the commandBufferSize. Units are in MiB

The `maxCommandBufferEntries`, `commandBufferSize`, and `maxInlineDataSize` can

also be set via the environment variables: `OSPRAY_MPI_MAX_COMMAND_BUFFER_ENTRIES`, `OSPRAY_MPI_COMMAND_BUFFER_SIZE`, and `OSPRAY_MPI_MAX_INLINE_DATA_SIZE`, respectively.

MPI Distributed Rendering

While MPI Offload rendering is used to transparently distribute rendering work without requiring modification to the application, MPI Distributed rendering is targetted at use of OSPRay within MPI-parallel applications. The MPI distributed device can be selected by loading the `mpi` module, and manually creating and using an instance of the `mpiDistributed` device:

```
ospLoadModule("mpi");

OSPDevice mpiDevice = ospNewDevice("mpiDistributed");
ospDeviceCommit(mpiDevice);
ospSetCurrentDevice(mpiDevice);
```

Your application can either initialize MPI before-hand, ensuring that `MPI_THREAD_SERIALIZED` or higher is supported, or allow the device to initialize MPI on commit. Thread multiple support is required if your application will make MPI calls while rendering asynchronously with OSPRay. When using the distributed device each rank can specify independent local data using the OSPRay API, as if rendering locally. However, when calling `ospRenderFrameAsync` the ranks will work collectively to render the data. The distributed device supports both image-parallel, where the data is replicated, and data-parallel, where the data is distributed, rendering modes. The `mpiDistributed` device will by default use each rank in `MPI_COMM_WORLD` as a render worker; however, it can also take a specific MPI communicator to use as the world communicator. Only those ranks in the specified communicator will participate in rendering.

Table 88: Parameters specific to the distributed `mpiDistributed` Device.

Type	Name	Default	Description
void *	worldCommunicator	MPI_COMM_WORLD	The MPI communicator which the OSPRay workers should treat as their world

Table 89: Parameters specific to the distributed `OSPWorld`.

Type	Name	Default	Description
box3f[]	region	NULL	A list of bounding boxes which bound the owned local data to be rendered by the rank

Table 90: Parameters specific to the `mpiRaycast` renderer.

Type	Name	Default	Description
int	aoSamples	0	The number of AO samples to take per-pixel
float	aoDistance	10^{20}	The AO ray length to use. Note that if the AO ray would have crossed a rank boundary and ghost geometry is not available, there will be visible artifacts in the shading

Image Parallel Rendering in the MPI Distributed Device

If all ranks specify exactly the same data, the distributed device can be used for image-parallel rendering. This works identical to the offload device, except that the MPI-aware application is able to load data in parallel on each rank rather than loading on the master and shipping data out to the workers. When a parallel file system is available, this can improve data load times. Image-parallel rendering is selected by specifying the same data on each rank, and using any of the existing local renderers (e.g., `scivis`, `pathtracer`). See ospMPIDistributedTutorialReplicatedData for an example.

Data Parallel Rendering in the MPI Distributed Device

The MPI Distributed device also supports data-parallel rendering with sort-last compositing. Each rank can specify a different piece of data, as long as the bounding boxes of each rank's data are non-overlapping. The rest of the scene setup is similar to local rendering; however, for distributed rendering only the `mpiRaycast` renderer is supported. This renderer implements a subset of the `scivis` rendering features which are suitable for implementation in a distributed environment.

By default the aggregate bounding box of the instances in the local world will be used as the bounds of that rank's data. However, when using ghost zones for volume interpolation, geometry or ambient occlusion, each rank's data can overlap. To clip these non-owned overlap regions out a set of regions (the `region` parameter) can pass as a parameter to the `OSPWorld` being rendered. Each rank can specify one or more non-overlapping `box3f`'s which bound the portions of its local data which it is reponsible for rendering. See the ospMPIDistributedTutorialStructuredVolume for an example.

Finally, the MPI distributed device also supports hybrid-parallel rendering, where multiple ranks can share a single piece of data. For each shared piece of data the rendering work will be assigned image-parallel among the ranks. Partially-shared regions are determined by finding those ranks specifying data with the same bounds (matching regions) and merging them. See the ospMPIDistributedTutorialPartiallyReplicatedData for an example.

Interaction With User Modules

The MPI Offload rendering mode trivially supports user modules, with the caveat that attempting to share data directly with the application (e.g., passing a `void\ *` or other tricks to the module) will not work in a distributed environment. Instead, use the `ospNewSharedData` API to share data from the application with OSPRay, which will in turn be copied over the network to the workers.

The MPI Distributed device also supports user modules, as all that is required for compositing the distributed data are the bounds of each rank's local data.

13.1.2 Appendices

OSPRay Studio

Intel OSPRay Studio is an open source and interactive visualization and ray tracing application that leverages Intel OSPRay as its core rendering engine. It can be used to load complex scenes requiring high fidelity rendering or very large scenes requiring supercomputing resources.

The main control structure is a scene graph which allows users to create an abstract scene in a directed acyclical graph manner. Scenes can either be imported or created using scene graph nodes and structure support. The scenes can then be rendered either with OSPRay's pathtracer or scivis renderer.

More information can be found at the OSPRay Studio website.

OSPRay Plug-in for USD Hydra

The Intel® OSPRay Plug-in for USD Hydra is an open source plugin for Pixar's USD to extend USD's Hydra rendering framework with Intel® OSPRay. The OSPRay for Hydra Plug-in enables interactive scene preview by utilizing OSPRay's high quality renderers and the Intel® Open Image Denoise denoiser.

As part of the oneAPI Rendering Toolkit, OSPRay is highly-optimized for Intel® CPU architectures ranging from laptops to large-scale distributed HPC systems. HdOSPRay leverages the Intel® Rendering Framework to deliver interactive rendering for large-scale models at high levels of fidelity.

More information can be found at the OSPRay for Hydra Plug-in website.

ISPC Implicit SPMD Program Compiler

ISPC is a compiler for a variant of the C programming language, with extensions for "single program, multiple data" (SPMD) programming. Under the SPMD model, the programmer writes a program that generally appears to be a regular serial program, though the execution model is actually that a number of program instances execute in parallel on the hardware.

ISPC compiles a C-based SPMD programming language to run on the SIMD units of CPUs and GPUs; it frequently provides a 3x or more speedup on architectures with 4-wide vector SSE units and 5x-6x on archituctures with 8-wide AVX vector units, without any of the difficulty of writing intrinsics code. Parallelization across multiple cores is also supported by ispc, making it possible to write programs that achieve performance improvement that scales by both number of cores and vector unit size.

More information can be found at the ISPC Implicit SPMD Program Compiler website.

Future Considerations

Acknowledgment

CHAPTER

FOURTEEN

HTML AND PDF VERSIONS

This section describes the versions that were available at time of publication. See the latest specification for updates.

Table 1: oneAPI Versions Table

Version	Date	View
1.0 rev 2	10/21/2020	HTML PDF
1.0 rev 1	09/14/2020	HTML PDF
0.9	07/30/2020	HTML PDF
0.85	06/29/2020	HTML PDF
0.8	05/29/2020	HTML PDF
0.7	03/26/2020	HTML PDF
0.5	11/17/2019	HTML

14.1 Release Notes

14.1.1 1.0 rev 2

- Formatting fixes for PDF

14.1.2 1.0 rev 1

- oneMKL
  - Continuing modifications to all oneMKL domains
  - Add oneMKL Exceptions and Error Handling
- oneDNN
  - Updated the DPC++ interoperability API
  - Added section about namespaces
- oneTBB
  - Updated the named requirements
  - Adjusted the API according to the latest product changes
  - Improved the wording and descriptions for the flow graph API
  - Cleared build warnings
- oneDAL

  - Extended Programming model section, added description of the generic methods and descriptors, introduced task notation
  - Extended Data Management section, introduced CSV data source, array, extended description of the tables and accessors
  - Extended description of Error handling
  - Added description of the training methods for PCA algorithm, introduced K-means initialization algorithm
  - Extended Common interface section, added description of service data types used in the interfaces of the library
  - Modified the library's namespace
- oneVPL
  - A lot of cleaning to remove GEN/CPU bias and implementation details.
  - New organization of structures' description.
  - New entry point function to expose by any implementation: MFXInitialize which replaces obsolete MFXInit and MFXInitEx.
  - Interface to choose and set subdevices (mfxExtDeviceAffinityMask) for execution.
  - Header files cleaned up by removing MFX_VERSION checks (ala #if MFX_VERSION > 1.26) from headers.
- oneCCL
  - Updated operation attributes description
  - Updated naming in API

14.1.3 0.9

- oneMKL
  - Continuing modifications to oneMKL Architecture, BLAS and LAPACK domains
  - Significant refactoring and updating of Sparse BLAS, VM, RNG, and DFT domains API descriptions and structure
  - Add Summary Statistics domain
  - Add future considerations and acknowledgment to appendices
  - Change top-level namespace to oneapi::mkl
- oneDNN
  - Added the specification for `map_data()` / `unmap_data()` methods of oneDNN memory objects
  - Extended element-wise algorithms and post operations
  - Added queries for peephole and projection weights for LSTM
- oneDPL
  - API updates, including namespaces
  - Added detailed descriptions for execution policies, non-standard algorithms, iterator types, etc
- oneTBB
  - Editorial review changes

- oneDAL
  - Updated description of the public header files and namespaces
- DPC++
  - Revised requirements to refer to the SYCL 2020 provisional specification where previous extensions have migrated to the SYCL specification directly
  - Added additional extensions
  - Added additional requirements against the SYCL 2020 provisional specification
- oneVPL
  - Added section to specify mandatory and optional API features
  - Reformatting of entire specification
  - Bug fixes in API definitions
- oneCCL
  - Added multi-device communication API
  - Added key-value store API
  - Extended list of collective operations

14.1.4 0.85

- oneVPL
  - High-performance video decode supporting MPEG-2, MPEG-4/H.264/AVC, H.265/HEVC, AV1, VP9, MJPEG;
  - High-performance video encode supporting MPEG-2, MPEG-4/H.264/AVC, H.265/HEVC, AV1, VP9, MJPEG;
  - Video processing for composition, alpha blending, deinterlace, resize, rotate, denoise, procamp, crop, detail, frame rate conversion, and color conversion.
- oneDNN
  - Added individual primitive definitions providing mathematical operation definitions and explaining details of their use
  - Expanded the programming model section explaining device abstraction and its interoperability with DPC++
  - Added the data model section explaining supported data types and memory layouts
  - Added specification for primitive attributes explaining, among other things, low-precision inference and bfloat16 training

14.1.5 0.8

- Level Zero
  - Updated to 0.95
- oneMKL
  - Continuing modifications to oneMKL Architecture and BLAS domain
  - Significant refactoring and updating of LAPACK domain API descriptions and structure.
- oneTBB
  - Significant rewrite and reorganization
- oneDAL
  - Extended description of Data Management component, added description of basic elements of algorithms, and error handling mechanism
  - Added description of namespaces and structure of the header files
  - Added specification of kNN algorithm
  - Introduced math notations section, extended glossary section
- oneDNN
  - Detailed descriptions for data model (tensor formats and data types), and execution models

14.1.6 0.7

- DPC++: 10 new language extensions including performance features like sub-groups and atomics, as well as features to allow more concise programs.
- oneDNN: Major restructuring of the document, with high-level introduction to the concepts
- Level Zero: Updated to 0.91. Open source release of driver implementing the specification
- oneDAL: Major restructuring of the document, with high-level introduction to the concepts
- oneVPL: Added support for device selection, context sharing, workstream presets and configurations, video processing and encoding APIs to easily construct a video processing pipeline.
- oneMKL: Added USM APIs. Major restructuring of document. Added architecture section with overview of execution model, memory model and API design.

14.1.7 0.5

Initial public release

CHAPTER

FIFTEEN

LEGAL NOTICES AND DISCLAIMERS

This specification is a continuation of Intel's decades-long history of working with standards groups and industry/academia initiatives such as The Khronos Group*, to create and define specifications in an open and fair process to achieve interoperability and interchangeability. oneAPI is intended to be an open specification and we encourage you to help us make it better. Your feedback is optional, but to enable Intel to incorporate any feedback you may provide to this specification, and to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification, please submit your feedback under the terms and conditions below. Any contribution of your feedback to the oneAPI Specification does not prohibit you from also contributing your feedback directly to other standard bodies, including The Khronos Group under their respective submission policies.

By opening an issue, providing feedback, or otherwise contributing to the specification, you agree that Intel will be free to use, disclose, reproduce, modify, license, or otherwise distribute your feedback at its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

BIBLIOGRAPHY

[OpenCLSpec] Khronos OpenCL Working Group, The OpenCL Specification Version:2.1 Document Revision:24 Available from opencl-2.1.pdf

[SYCLSpec] Khronos®OpenCL™ Working Group — SYCL™ subgroup, SYCL™ Specification SYCL™ integrates OpenCL™ devices with modern C++, Version 1.2.1 Available from sycl-1.2.1.pdf

[Lloyd82] Stuart P Lloyd. *Least squares quantization in PCM*. IEEE Transactions on Information Theory 1982, 28 (2): 1982pp: 129–137.

[Bro07] Bro, R.; Acar, E.; Kolda, T. *Resolving the sign ambiguity in the singular value decomposition*. SANDIA Report, SAND2007-6422, Unlimited Release, October, 2007.

[Bentley80] J. L. Bentley. Multidimensional Divide and Conquer. Communications of the ACM, 23(4):214–229, 1980.

[Friedman17] J. Friedman, T. Hastie, R. Tibshirani. *The Elements of Statistical Learning Data Mining, Inference, and Prediction.* Springer, 2017.

[Zhang04] T. Zhang. Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms. ICML 2004: Proceedings Of The Twenty-First International Conference On Machine Learning, 919–926, 2004.

[Lang87] S. Lang. *Linear Algebra*. Springer-Verlag New York, 1987.

[Ping14] Ping Tak Peter and Eric Polizzi. *FEAST as a Subspace Iteration Eigensolver Accelerated by Approximate Spectral Projection.* 2014.

[Demmel90] J. W. Demmel and W. Kahan. *Accurate singular values of bidiagonal matrices*. SIAM J. Sci. Stat. Comput., 11 (1990), pp. 873-912.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y