SYCL^{TM} Provisional Specification

SYCL integrates OpenCL devices with modern C++ using a single source design

Version 2.2

Revision Date: – 2016/02/15

Khronos OpenCL Working Group — SYCL subgroup

Editor: Maria Rovatsou

Contents

List of Tables

Acknowledgements

Editors: Maria Rovatsou, Codeplay

Contributors:

- Eric Berdahl, Adobe
- Brian Sumner, AMD
- Gordon Brown, Codeplay
- Duncan McBain, Codeplay
- Ruyman Reyes, Codeplay
- Andrew Richards, Codeplay
- Maria Rovatsou, Codeplay
- Allen Hux, Intel
- Lee Howes, Qualcomm
- Chu-Cheow Lim, Qualcomm
- Jack Liu, Qualcomm
- Ronan Keryell, Xilinx

1. Introduction

SYCL is an open standard C++ programming model for OpenCL. SYCL v2.2 builds on the underlying concepts, portability and efficiency of OpenCL v2.2 while adding much of the ease of use and flexibility of C++. Developers using SYCL are able to write standard C++ code, with many of the techniques they are accustomed to, such as inheritance and templating. At the same time, developers have access to the full range of capabilities of OpenCL both through the features of the SYCL libraries and, where necessary, through interoperation with code written directly to use the OpenCL APIs. SYCL v2.2 is backwards compatible with the previous version of the SYCL standard, v1.2.

SYCL uses the braided parallelism model of parallelism, whereby a task graph is built up (at runtime) made up of data parallel and task parallel tasks running across different cores and devices within a system.

SYCL implements a shared source design which offers the power of source integration while allowing toolchains to remain flexible. The shared source design supports embedding of code intended to be compiled for an OpenCL device, for example a GPU, inline with host code. This embedding of code offers three primary benefits:

Simplicity For novice programmers, the separation of host and device source code in OpenCL can become complicated to deal with, particularly when similar kernel code is used for multiple different operations. A single compiler flow and integrated tool chain combined with libraries that perform a lot of simple tasks simplifies initial OpenCL programs to a minimum complexity. This reduces the learning curve for programmers new to OpenCL and allows them to concentrate on parallelization techniques rather than syntax.

Reuse C++'s type system allows for complex interactions between different code units and supports efficient abstract interface design and reuse of library code. For example, a *transform* or *map* operation applied to an array of data may allow specialization on both the operation applied to each element of the array and on the type of the data. The shared source design of SYCL enables this interaction to bridge the host code/device code boundary such that the device code to be specialized on both of these factors directly from the host code.

Efficiency Tight integration with the type system and reuse of library code enables a compiler to perform inlining of code and to produce efficient specialized device code based on decisions made in the host code without having to generate kernel source strings dynamically.

SYCL is designed to allow a compilation flow where the source file is passed through multiple different compilers, including a standard C++ host compiler of the developer's choice, and where the resulting application combines the results of these compilation passes. This is distinct from a single-source flow that might use language extensions that preclude the use of a standard host compiler. The SYCL standard does not preclude the use of a single compiler flow, but is designed to not require it.

The advantages of this design are two-fold. First, it offers better integration with existing tool chains. An application that already builds using a chosen compiler can continue to do so when SYCL code is added. Using the SYCL tools on a source file within a project will both compile for an OpenCL device and let the same source file be compiled using the same host compiler that the rest of the project is compiled with. Linking and library relationships are unaffected. This design simplifies porting of pre-existing applications to SYCL. Second, the design allows the optimal compiler to be chosen for each device where different vendors may provide optimised tool-chains.

SYCL is designed to be as close to standard C++14 as possible. In practice, this means that as long as no dependence is created on SYCL's integration with OpenCL, a standard C++14 compiler can compile the SYCL programs and they will run correctly on host CPU. Any use of specialized low-level features can be masked using the C pre-processor in the same way that compiler-specific intrinsics may be hidden to ensure portability between different host compilers.

SYCL retains the execution model, runtime feature set and device capabilities of the underlying OpenCL standards. Different devices within a system may support different versions, optional capabilities or extensions of the OpenCL standards. This version of the specification (version 2.2) provides a programming model that exposes the capabilities of all OpenCL devices up to and including OpenCL 2.2. The user must specify, for a given kernel, the required version and capabilities of the underlying OpenCL device which will execute that kernel and the user must ensure that the kernel only uses the capabilities of that device.

While the code can be written in standard C++ syntax with interoperability with standard C++ programs, there are C++ capabilities which are not supported in SYCL device code. For all currently specified devices, SYCL device code does not support virtual function calls, function pointers in general, exceptions, or runtime type information. There are other C++ features that are disallowed according to the capabilities of the device being compiled for. This means that users cannot rely on making use of C++ libraries inside SYCL device code unless those libraries have been limited to only use features supported by SYCL for the relevant device. The restrictions on device code are described in this specification.

The use of C++ features such as templates and inheritance on top of the OpenCL execution model opens a wide scope for innovation in software design for heterogeneous systems. Clean integration of device and host code within a single C++ type system enables the development of modern templated libraries that build simple, yet efficient, interfaces to offer more developers access to OpenCL capabilities and devices. SYCL is intended to serve as a foundation for innovation in programming models for heterogeneous systems that builds on an open and widely implemented standard foundation in the form of OpenCL.

SYCL exposes the parallelism of OpenCL's work-items, work-groups and sub-groups via a *hierarchical parallelism* syntax. This form cleanly layers parallel loops and synchronization points to avoid fragmentation of code and to efficiently map algorithms to CPU-style architectures.

OpenCL v2.2 supports *Shared Virtual Memory* (or SVM) which enables pointers to be shared between host and device code. Different OpenCL devices support different optional capabilities of SVM. To enable the widest level of portability, SYCL v2.2 supports the RAII buffer/image/accessor model of data access from SYCL v1.2. Also, to support the full capabilities of OpenCL 2.2 capable devices, SYCL supports different levels of OpenCL SVM. Developers using SYCL 2.2 can decide whether to use the buffer/image/accessor model to achieve maximum portability and to allow the system to extract parallelism from the application. Or, developers can choose to use one or more of the levels of SVM to allow greater user control over data management and sharing.

To summarize, SYCL enables OpenCL kernels to be written inside C++ source files. This means that software developers can develop and use generic algorithms and data structures using standard C++ template techniques, while still supporting the multi-platform, multi-device heterogeneous execution of OpenCL. The specification has been designed to enable implementation across as wide a variety of platforms as possible as well as ease of integration with other platform-specific technologies, thereby letting both users and implementers build on top of SYCL as an open platform for heterogeneous processing innovation.

2. SYCL Architecture

This chapter builds on the structure of the OpenCL specification's architecture chapter to explain how SYCL overlays the OpenCL 2.2 specification and inherits its capabilities and restrictions as well as the additional features it provides on top of OpenCL 2.2.

2.1 Overview

SYCL is an open industry standard for programming a heterogeneous system. The design of SYCL allows standard C++ source code to be written such that it can run on either an OpenCL device or on the host CPU. SYCL 2.2 is designed to be backwards compatible with SYCL 1.2, but updated with support for the features of OpenCL 2.2 capable devices. As OpenCL 2.2 contains a number of optional capabilities and extensions, the user should specify the capabilities of the device that code is designed to execute on.

The terminology used for SYCL inherits that of OpenCL with some SYCL-specific additions. The code that can run on either an OpenCL device or host CPU is called a *kernel*. To ensure maximum backward-compatibility, a software developer can produce a program that mixes standard OpenCL C kernels and OpenCL API code with SYCL code and expect fully compatible interoperation.

The target users of SYCL are C++ programmers who want all the performance and portability features of OpenCL, but with the flexibility to use higher-level C++ abstractions across the host/device code boundary. Developers can use most of the abstraction features of C++, such as templates, classes and operator overloading. However, some C++ language features are not permitted inside kernels, due to the limitations imposed by the capabilities of the underlying OpenCL standard. These features include virtual functions, virtual inheritance, throwing/catching exceptions, and run-time type-information. These features are available outside kernels as normal. Within these constraints, developers can use abstractions defined by SYCL, or they can develop their own on top. These capabilities make SYCL ideal for library developers, middleware providers and applications developers who want to separate low-level highly-tuned algorithms or data structures that work on heterogeneous systems from higher-level software development. OpenCL developers can produce templated algorithms that are easily usable by developers in other fields.

2.2 The SYCL Platform and Device Model

The SYCL platform model is based on the OpenCL platform model, but there are a few additional abstractions available to programmers.

The OpenCL model consists of a host with one or more OpenCL *platforms*. Each OpenCL platform can have one or more OpenCL devices. Each device can support different versions of OpenCL, with different optional capabilities and extensions. SYCL abstracts this to have a host and zero or more OpenCL devices, of which the host itself is also a SYCL *device*.

A SYCL application is implemented as both host code and device kernel code. The host code portion of a SYCL application runs on a host processor according to the models native to the host platform. The SYCL application code submits *command groups* to *queues* that are executed on devices. A SYCL device (which may be the host or an OpenCL device) executes the commands on the device. A command group consists of synchronization and data movement commands, as well as user-defined *kernels*, which are embedded within the C++ source code of the SYCL program.

A SYCL device has considerable latitude on how computations are mapped onto the device's processing elements. When processing elements within a compute unit execute the same sequence of statements across the processing elements, the control flow is said to be *converged*. Hardware optimized for executing a single stream of instructions over multiple processing elements is well suited to converged control flows. When the control flow varies from one processing elements to another, it is said to be *diverged*. While a kernel always begins execution with a converged control flow, due to branching statements within a kernel, converged and diverged control flows may occur within a single kernel. This provides a great deal of flexibility in the algorithms that can be implemented with SYCL.

2.2.1 Platform Mixed Version Support

OpenCL is designed to support devices with different capabilities on a single machine. This includes devices which conform to different versions of the OpenCL specification and devices which support different extensions to the OpenCL specification. There are three important sets of capabilities to consider for a SYCL device: the platform version, the version of a device and the extensions supported.

The SYCL system presents the user with a set of devices, which can be queried for their capabilities. If a SYCL device is the SYCL host device, then there are a limited number of queries, due to the relative simplicity of host device capabilities. If a SYCL device is an OpenCL device, then SYCL provides all of the OpenCL device query operations.

Also, in SYCL v2.2, command groups can specify the capability requirements of the command group, via template parameters to the `execution_handle` type in the queue submit method call. This allows users of SYCL to specify what special requirements (such as the level of Shared Virtual Memory, or SVM) that is required for correct data access and kernel execution. This allows the SYCL runtime to ensure the device capabilities match the requirements of the command group, as well passing down to the underlying OpenCL runtime any optional capabilities that may need to be configured.

The device version is an indication of the device's capabilities, as represented by the device information returned by the `cl::sycl::device::get_info()` method. Examples of attributes associated with the device version are resource limits and information about functionality beyond the core OpenCL specification's requirements. The version returned corresponds to the highest version of the OpenCL specification for which the device is conformant, but is not higher than the version of the device's platform which bounds the overall capabilities of the runtime operating the device.

In OpenCL, a device has a *language version*. In SYCL, the source language is independent of the device version. SYCL 2.2 source code can run on a variety of devices (including OpenCL 1.2 devices), if the capabilities specified by the user in the command group `execution_handle` are within the capabilities of the underlying device.

2.3 SYCL Execution Model

Execution of a SYCL program occurs in two parts: *kernels* that execute on either the host CPU, or one or more *OpenCL devices*, and a *host program* that executes on the host CPU. The host program defines the context for the kernels and manages their execution. Like OpenCL, SYCL is capable of running kernels on multiple device types. However, SYCL builds on top of OpenCL due to the integration into a host toolchain by providing an ability to run kernel code directly on the CPU without interacting with an OpenCL runtime. This is distinct from running on the CPU via an OpenCL device and can be used when no OpenCL platform is available on the machine.

In OpenCL, *queues* contain *commands*, which can include data transfer operations, synchronization commands, or *kernels* submitted for execution. In SYCL, the commands are grouped together into a functor object called *command group*. Command groups associate sets of data movement operations with kernels that will be enqueued together on an underlying OpenCL queue, through the command group `handler` object which is created for every command group functor. These data transfer operations may be needed to make available data that the kernel needs or to return its results to other devices.

When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel body executes for each point in this index space. This kernel instance is called a *work-item* and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary by using the work-item global ID to specialize the computation.

Work-items are organized into work-groups. The work-groups provide a more coarse-grained decomposition of the index space. Each work-group is assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are each assigned a local ID, unique within the work-group, so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. Within a work-group, work-items may be divided into sub-groups. The mapping of work-items to sub-groups is implementation-defined and may be queried at runtime. While sub-groups may be used in multi-dimensional work-groups, each sub-group is 1-dimensional and any given work-item may query which sub-group it is a member of.

The index space supported in SYCL is called an `nd_range`. An `nd_range` is an N-dimensional index space, where n is one, two or three. In SYCL, the `nd_range` is represented via the `nd_range<N>` class. An `nd_range<N>` is made up of a global range and a local range, each represented via values of type `range<N>` and a global offset, represented via a value of type `id<N>`. The types `nd_range<N>` and `id<N>` are each N-dimensional arrays of integers. The iteration space defined via an `range<N>` is an N-dimensional index space starting at the nd_range's global offset and being of the size of its global range, split into work-groups of the size of its local range.

Each work-item in the `nd_range` is identified by a value of type `nd_item<N>`. The type `nd_item<N>` encapsulates a global ID, local ID and work-group ID, all of type `id<N>`, the iteration space offset also of type `id<N>`, as well as global and local ranges and synchronization operations necessary to make work-groups useful. Work-groups are assigned IDs using a similar approach to that used for work-item global IDs. Work-items are assigned to a work-group and given a local ID with components in the range from zero to the size of the work-group in that dimension minus one. Hence, the combination of a work-group ID and the local-ID within a work-group uniquely defines a work-item.

SYCL allows a simplified execution model in which the workgroup size is left undefined. A kernel invoked over a `range<N>`, instead of an `nd_range<N>` is executed within an iteration space of undefined workgroup size. In this case, less information is available to each work-item through the simpler `item<N>` class.

2.3.1 Execution Model: Queues, Command Groups and Contexts

In OpenCL, a developer must create a *context* to be able to execute commands on a device. Creating a context involves choosing a *platform* and a list of *devices*. In SYCL, contexts, platforms and devices all exist, but the user can choose whether to specify them or have the SYCL implementation create them automatically. The minimum required object for submitting work to devices in SYCL is the *queue*, which contains references to a platform, device and context internally.

The resources managed by SYCL are:

1. *Platforms*: All features of OpenCL are implemented by platforms. A platform can be viewed as a given hardware vendor's runtime and the devices accessible through it. Some devices will only be accessible to one vendor's runtime and hence multiple platforms may be present. SYCL manages the different platforms for the user. In SYCL, a platform resource is accessible through a `cl::sycl::platform` object. SYCL also provides a host platform object, which only contains a single host device.

2. *Contexts*: Any OpenCL resource that is acquired by the user is attached to a context. A context contains a collection of devices that the host can use and manages memory objects that can be shared between the devices. Data movement between devices within a context may be efficient and hidden by the underlying runtime while data movement between contexts must involve the host. A given context can only wrap devices owned by a single platform. In SYCL, a context resource is accessible through by a `cl::sycl::context` object.

3. *Devices*: Platforms provide one or more devices for executing kernels. In SYCL, a device is accessible through a `cl::sycl::device` object. SYCL provides the abstract `cl::sycl::device_selector` class which the user can subclass to define how the runtime should select the best device from all available platforms for the user to use. For ease of use, SYCL provides a set of predefined concrete `device_selector` instances that select devices based on common criteria, such as type of device. SYCL, unlike OpenCL, defines a host device, which means any work that uses the host device will execute on the host and not on any OpenCL device.

4. *Command groups*: SYCL groups OpenCL *commands* into a functor given a unique command group `handler` object to perform all the necessary work required to correctly process host data on a device using a kernel. In this way, they group the commands of transferring and processing these data in order to enqueue them on a device for execution. Command groups are submitted to a SYCL queue. The handler object can be optionally templated with capabilities, which determine the required capabilities of the device and context executing the command group. These capabilities may include features like level of shared virtual memory required by the command-group, which impacts scheduling and how data is shared and synchronized between host and command groups.

5. *Kernels*: The SYCL functions that run on SYCL devices (i.e. either an OpenCL device, or the host) are defined as C++ functors or lambda functions. In SYCL, all kernels must have a *name*, which must be a globally-accessible C++ typename. This is required to enable kernels compiled with one compiler to be linked to host code compiled with a different compiler. For functors, the typename of the functor is sufficient as the kernel *name*, but for C++11 lambda functions, the user must provide a user- defined *name*.

6. *Program Objects*: OpenCL objects that store implementation data for the SYCL kernels. These objects are only required for advanced use in SYCL and are encapsulated in the `cl::sycl::program` class.

7. *Command-queues*: SYCL kernels execute in command queues. The user must create a queue, which references an associated context, platform and device. The context, platform and device may be chosen automatically, or specified by the user. In SYCL, command queues are accessible through `cl::sycl::queue`

objects.

The command-queue schedules commands submitted to it for execution on a device. Commands launched by the host execute asynchronously with respect to the host thread, and not necessarily ordered with respect to each other. It is the responsibility of the SYCL implementation to ensure that the different commands execute in an order which preserves SYCL semantics. This means that a SYCL implementation must map, move or copy data between host and device memory, execute kernels and perform synchronization between different queues, devices and the host in a way that matches the semantics defined in this specification. If a command group runs on an OpenCL device, then this is expected to be achieved by enqueuing the right memory and synchronization commands to the queue to ensure correct execution. If a command group runs on the host device, then this is expected to be achieved by host-specific synchronization as well as by ensuring that no OpenCL device is simultaneously using any required data.

2.3.2 Execution Model: Command Queues

SYCL command groups are submitted to command-queues, via a command group handler. Each command-queue is associated with a single device. The commands placed into the command-queue fall into one of three types:

- Kernel-enqueue commands: Enqueue a kernel for execution on a device.
- Memory commands: Transfer data between the host and device memory, between memory objects, or map and unmap memory objects from the host address space.
- Synchronization commands: Explicit synchronization points that define order constraints between commands.

Mostly, with SYCL, individual OpenCL commands are enqueued to OpenCL command queues automatically, using the data access encapsulated in SYCL objects, such as buffers and accessors as well as the enqueuing of kernels from parallel kernel invocation functions, such as `parallel_for`.

In addition to commands submitted from the host command-queue, a kernel running on a device can enqueue commands to a device-side command queue. This results in child kernels enqueued by a kernel executing on a device (the parent kernel). Regardless of whether the command-queue resides on the host or a device, each command passes through six states.

1. **Queued:** The command is enqueued to a command-queue. A command may reside in the queue until it is flushed either explicitly or implicitly by some other command.
2. **Submitted:** The command is flushed from the command-queue and submitted for execution on the device. Once flushed from the command-queue, a command will execute after any prerequisites for execution are met.
3. **Ready:** All prerequisites constraining execution of a command have been met. The command, or for a kernel-enqueue command the collection of work groups associated with a command, is placed in a device work-pool from which it is scheduled for execution.
4. **Running:** Execution of the command starts. For the case of a kernel-enqueue command, one or more work-groups associated with the command start to execute.
5. **Ended:** Execution of a command ends. When a Kernel-enqueue command ends, all of the work-groups associated with that command have finished their execution. Immediate side effects, i.e. those associated

with the kernel but not necessarily with its child kernels, are visible to other units of execution. These side effects include updates to values in global memory.

6. **Complete:** The command and its child commands have finished execution and the status of the event object, if any, associated with the command is set to `event_status::complete`.

The execution states and the transitions between them are summarized in Figure 2.1.

These states and the concept of a device work-pool are conceptual elements of the execution model. An implementation of OpenCL has considerable freedom in how these are exposed to a program. Five of the transitions, however, are directly observable through a profiling interface. These profiled states are shown in Figure 2.1.

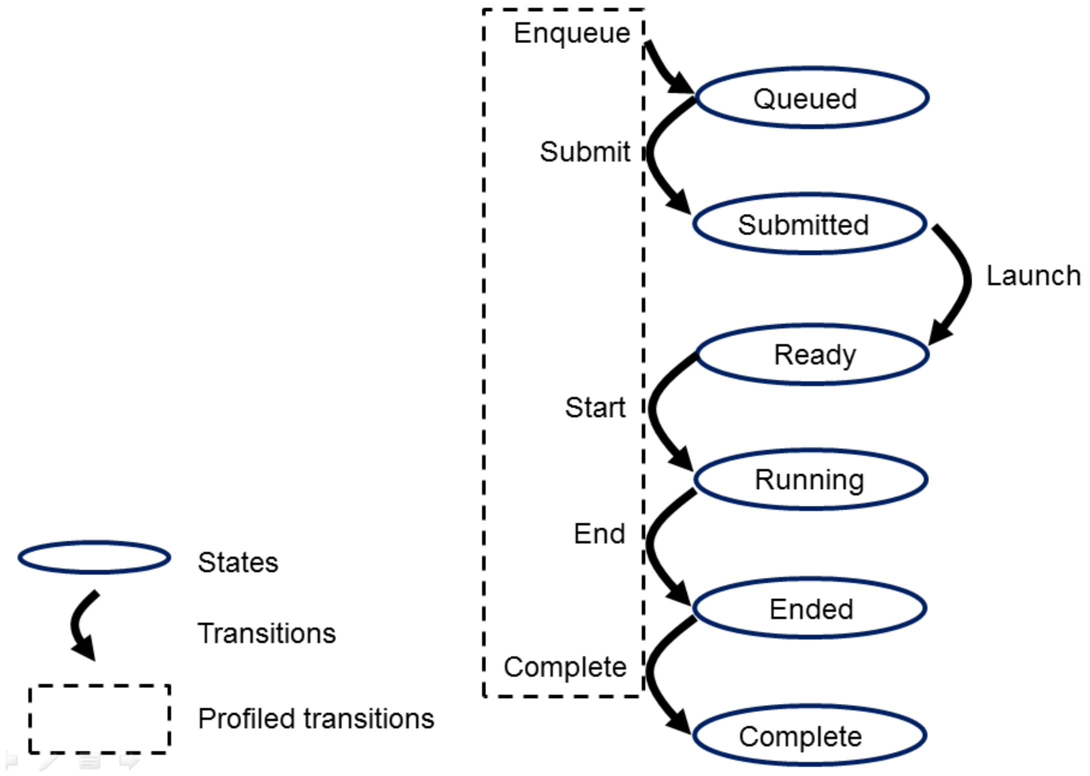

Figure 2.1: The states and transitions between states as defined in the OpenCL 2.2 execution model.

Commands communicate their status through `event` objects. Successful completion is indicated by setting the event status associated with a command to `event_status::complete`. Unsuccessful completion results in abnormal termination of the command which is indicated by setting the event status to a negative value. In this case, the command-queue associated with the abnormally terminated command and all other command-queues in the same context may no longer be available and their behavior is implementation defined.

A command submitted to a device will not launch until prerequisites that constrain the order of commands have been resolved. These prerequisites have three main sources:

1. Commands that access `buffer` or `image` objects via `accessor` objects are scheduled by the SYCL runtime to ensure that accesses to buffers and images are ordered to preserve correct access semantics.

2. In the case of command groups that are specified to run with *fine grained buffer sharing*, synchronization

is performed on the SVM allocations that are registered to be accessed by the command group. As with buffers and images, the synchronization ensures correctly ordered access semantics.

3. They may arise from commands submitted to a command-queue that constrain the order in which commands are launched. For example, commands that follow a command queue barrier will not launch until all commands prior to the barrier are complete.

Most SYCL operations are non-blocking. Enqueueing a command group on the host does not block host execution, instead it enqueues work to a command queue. The only main blocking operations are accessing data on host, which includes construction of host accessors and destruction of buffers (in the case where the buffer needs to ensure results are available on the host after destruction).

Multiple command-queues can be present within a single context. Multiple command- queues execute commands independently. Command groups enqueued to different queues by SYCL are scheduled correctly by the SYCL runtime to ensure correct data access ordering.

When a kernel-invocation command such as `parallel_for` submits a kernel for execution, an index space is defined. The kernel, its captured arguments (in the case of a C++11 lambda) or its data members (in the case of a kernel functor) and the parameters that define the index space define a kernel-instance. When a kernel-instance executes on a device, the kernel functor or lambda executes for each point in the defined index space. Each of these executing kernel methods are called a work-item. The work-items associated with a given kernel-instance are managed by the device in groups called work-groups. These work-groups define a coarse grained decomposition of the *Index* space. Work-groups are further divided into sub-groups, which provide an additional level of control over execution.

Work-items have a global ID based on their coordinates within the Index space. They can also be defined in terms of their work-group and the local ID within a work-group. The details of this mapping are described in the following section.

2.3.3 Execution Model: Mapping work-items onto an nd_range

The index space supported by OpenCL is called an `nd_range`. An `nd_range` is an N-dimensional index space, where N is one, two or three. The `nd_range` is decomposed into work-groups forming blocks that cover the Index space. An `nd_range` is defined by three integer arrays of length N:

- The extent of the index space (or global size) in each dimension.
- An offset index F indicating the initial value of the indices in each dimension (zero by default).
- The size of a work-group (local size) in each dimension.

Each work-items global ID is an N-dimensional tuple. The global ID components are values in the range from F, to F plus the number of elements in that dimension minus one.

If a kernel is compiled within a SYCL 2.2 command group handler, the size of work-groups in an `nd_range` (the local size) need not be the same for all work-groups. In this case, any single dimension for which the global size is not divisible by the local size will be partitioned into two regions. One region will have work-groups that have the same number of work items as was specified for that dimension by the programmer (the local size). The other region will have work-groups with less than the number of work items specified by the local size parameter in that dimension (the remainder work-groups). Work-group sizes could be non-uniform in multiple dimensions, potentially producing work-groups of up to 4 different sizes in a 2D range and 8 different sizes in a 3D range.

Each work-item is assigned to a work-group and given a local ID to represent its position within the work-group. A work-item's local ID is an N-dimensional tuple with components in the range from zero to the size of the work-group in that dimension minus one.

Work-groups are assigned IDs similarly. The number of work-groups in each dimension is not directly defined but is inferred from the local and global `nd_range` provided when a kernel-instance is enqueued. A work-group's ID is an N- dimensional tuple with components in the range 0 to the ceiling of the global size in that dimension divided by the local size in the same dimension. As a result, the combination of a work-group ID and the local-ID within a work-group uniquely defines a work-item. Each work-item is identifiable in two ways; in terms of a global index, and in terms of a work-group index plus a local index within a work group.

For example, consider the 2-dimensional index space in Figure2.2. We input the index space for the work-items (G_x, G_y), the size of each work-group (S_x, S_y) and the global ID offset (F_x, F_y). The global ndices define an G_x by G_y index space where the total number of work-items is the product of G_x and G_y. The local indices define an S_x by S_y index space where the number of work-items in a single work-group is the product of S_x and S_y. Given the size of each work-group and the total number of work-items we can compute the number of work-groups. A 2-dimensional index space is used to uniquely identify a work- group. Each work-item is identified by its global ID (g_x, g_y) or by the combination of the work-group ID (w_x, w_y), the size of each work-group (S_x,S_y) and the local ID (s_x, s_y) inside the work-group such that

$$g_x, g_y) = (w_x * S_x + s_x + F_x, w_y * S_y + s_y + F_y) \tag{2.1}$$

The number of work-groups can be computed as:

$$W_x, W_y) = (ceil(G_x/S_x), ceil(G_y/S_y)) \tag{2.2}$$

Given a global ID and the work-group size, the work-group ID for a work-item is computed as:

$$w_x, w_y) = ((g_x s_x F_x)/S_x, (g_y s_y F_y)/S_y) \tag{2.3}$$

Within a work-group, work-items may be divided into sub-groups. The mapping of work-items to sub-groups is implementation-defined and may be queried at runtime. While sub-groups may be used in multi-dimensional work-groups, each sub-group is 1-dimensional and any given work-item may query which sub-group it is a member of.

Work items are mapped into sub-groups through a combination of decisions and during the compilation from SPIR-V to device-specific binary as well as the parameters of the dispatch. The mapping to sub-groups is invariant for the duration of a kernels execution, across dispatches of a given kernel with the same work-group dimensions, between dispatches and query operations consistent with the dispatch parameterization, and from one work-group to another within the dispatch (excluding the trailing edge work-groups in the presence of non-uniform work-group sizes). In addition, all sub-groups within a work-group will be the same size, apart from the sub-group with the maximum index which may be smaller if the size of the work-group is not evenly divisible by the size of the sub- groups.

In the degenerate case, a single sub-group must be supported for each work-group. In this situation all sub-group scope functions are equivalent to their work- group level equivalents.

2.3.4 Execution Model: Execution of kernel-instances

The work carried out by a SYCL program occurs through the execution of kernel- instances on compute devices. To understand the details of SYCLs execution model, we need to consider how a kernel object moves from the

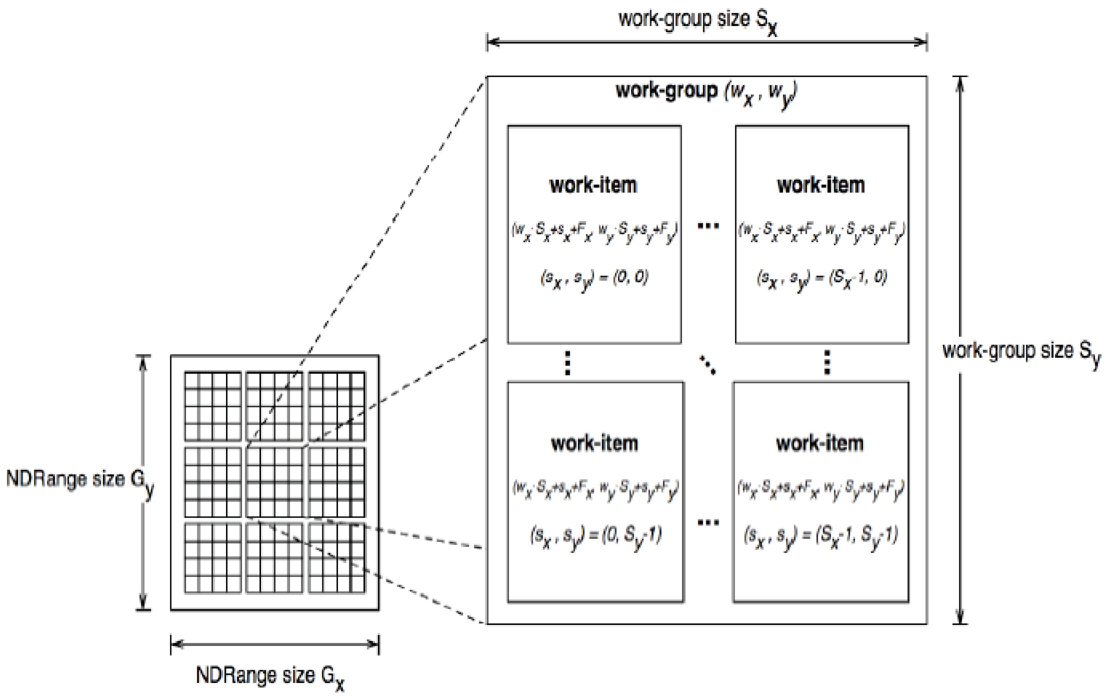
work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Figure 2.2: An example of an NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs. In this case, we assume that in each dimension, the size of the work-group evenly divides the global NDRange size (i.e. all work-groups have the same size) and that the offset is equal to zero

kernel-enqueue command, into a command-queue, executes on a device, and completes.

A kernel-object is defined from a functor or lambda within a command group. The host program enqueues a kernel-object to the command queue along with the `nd_range`, and the work-group decomposition. These define a kernel- instance. In addition, the SYCL scheduler will determine a correct ordering of command execution, based on preserving correct data access semantics.

If a command-group uses *Fine Grained System Sharing*, then the SYCL implementation is not given any information about data access within the kernel. This means that the SYCL scheduler cannot ensure any correct schedule and so it is the user's responsibility to schedule code correctly.

For any command-group that *is not* fine grained system sharing, the kernel instances appear to launch and then execute in that same order; where we use the term *appear* to emphasize that when there are no dependencies between command groups and hence differences in the order that commands execute cannot be observed in a program, an implementation can reorder commands.

Once these conditions are met, the kernel-instance is launched and the work- groups associated with the kernel-instance are placed into a pool of ready to execute work-groups. This pool is called a work-pool. The work-pool may be implemented in any manner as long as it assures that work-groups placed in the pool will eventually execute. The device schedules work-groups from the work-pool for execution on the compute units of the device. The kernel-enqueue command is complete when all work-groups associated with the kernel-instance end their execution, updates to global memory associated with a command are visible globally, and the device signals successful completion by setting the event associated with the kernel-enqueue command to `event_status::complete`.

While a command-queue is associated with only one device, a single device may be associated with multiple command-queues all feeding into the single work-pool. A device may also be associated with command queues associated with different contexts within the same platform, again all feeding into the single work-pool. The device will pull work-groups from the work-pool and execute them on one or several compute units in any order; possibly interleaving execution of work- groups from multiple commands. A conforming implementation may choose to serialize the work-groups so a correct algorithm cannot assume that work-groups will execute in parallel. There is no safe and portable way to synchronize across the independent execution of work-groups since once in the work-pool, they can execute in any order.

The work-items within a single sub-group execute concurrently but not necessarily in parallel (i.e. they are not guaranteed to make independent forward progress). Therefore, only high-level synchronization constructs (e.g. sub-group functions such as barriers) that apply to all the work-items in a sub-group are well defined and included in SYCL.

Sub-groups execute concurrently within a given work-group and with appropriate device support may make independent forward progress with respect to each other, with respect to host threads and with respect to any entities external to the SYCL system but running on an OpenCL device, even in the absence of work-group barrier operations. In this situation, sub-groups are able to internally synchronize using barrier operations without synchronizing with each other and may perform operations that rely on runtime dependencies on operations other sub- groups perform.

The work-items within a single work-group execute concurrently but are only guaranteed to make independent forward progress in the presence of sub-groups and device support. In the absence of this capability, only high-level synchronization constructs (e.g. work-group functions such as barriers) that apply to all the work-items in a work-group are well defined and included in SYCL for synchronization within the work-group.

In the absence of synchronization functions (e.g. a barrier), work-items within a sub-group may be serialized. In the presence of sub-group functions, work-items within a sub-group may be serialized before any given sub-group function, between dynamically encountered pairs of sub-group functions and between a work- group function and the end of the kernel.

In the absence of independent forward progress of constituent sub-groups, work- items within a work-group may be serialized before, after or between work-group synchronization functions.

2.3.5 Execution Model: Hierarchical Parallelism

In SYCL, the work-item, sub-group and work-group hierarchy can be represented by nested `parallel_for` sections. It is the responsibility of the SYCL system to execute these as if they are nested parallel loops. This means that no dependency checking is performed between different iterations of the same loop and there is no guarantee that the loops are executed either serial or parallel. The end of the hierarchical loops are synchronization points. It is the responsibility of the SYCL system to ensure that correct OpenCL synchronization code is inserted at the ends and starts of parallel loops.

To enable code at work-group or sub-group `parallel_for` scope to execute semantically correctly on OpenCL devices, a SYCL compiler may need to transform the code to ensure that only one work-item per sub-group or work-group executes that code. The nature of this transformation is not specified, however it may involve predication by work-item id, or via uniform execution across a group.

2.3.6 Execution Model: Device-side enqueue

Algorithms may need to generate additional work as they execute. In many cases, this additional work cannot be determined statically; so the work associated with a kernel only emerges at runtime as the kernel-instance executes. This capability could be implemented in logic running within the host program, but involvement of the host may add significant overhead and/or complexity to the application control flow. A more efficient approach would be to nest kernel-enqueue commands from inside other kernels. This **nested parallelism** can be realized by supporting the enqueuing of kernels on a device without direct involvement by the host program; so-called **device-side enqueue**.

Device-side kernel-enqueue operations are similar to host-side command groups. The kernel executing on a device (the parent kernel) enqueues a kernel-instance (the child kernel) to a device-side command queue. This is an out-of-order command-queue and follows the same behavior as the out-of-order command-queues exposed to OpenCL host programs. Kernels enqueued to a device side command-queue generate and use events to enforce order constraints. These events, however, are only visible to the parent kernel running on the device. When these prerequisite events take on the value `event_status::complete`, the work-groups associated with the child kernel are launched into the devices work pool. The device then schedules them for execution on the compute units of the device. Child and parent kernels execute asynchronously. However, a parent will not indicate that it is complete by setting its event to `event_status::complete` until all child kernels have ended execution and have signalled completion by setting any associated events to the value `event_status::complete`. Should any child kernel complete with an event status set to a negative value (i.e. abnormally terminate), the parent kernel will abnormally terminate and propagate the childs negative event value as the value of the parents event. If there are multiple children that have an event status set to a negative value, the selection of which childs negative event value is propagated is implementation-defined.

2.3.7 Execution Model: Synchronization

Synchronization refers to mechanisms that constrain the order of execution between two or more units of execution. Consider the following three domains of synchronization in SYCL:

- Work-group synchronization: Constraints on the order of execution for work-items in a single work-group
- Sub-group synchronization: Constraints on the order of execution for work-items in a single sub-group
- Command group synchronization: Constraints on the order of command groups launched for execution

Synchronization across all work-items within a single work-group is carried out using either an explicit work-group synchronization function, or implicitly at the start or end of hierarchical `parallel_for` loops.

Explicit work-group synchronization functions carry out collective operations across all the work-items in a work-group. Available collective operations are: barrier, reduction, broadcast, prefix sum, and evaluation of a predicate. A work- group function must occur within a converged control flow; i.e. all work-items in the work-group must encounter precisely the same work-group function. For example, if a work-group function occurs within a loop, the work-items must encounter the same work-group function in the same loop iterations. All the work- items of a work-group must execute the work-group function and complete reads and writes to memory before any are allowed to continue execution beyond the work- group function. Work-group functions that apply between work-groups are not provided in SYCL since OpenCL does not define forward-progress or ordering relations between work-groups, hence collective synchronization operations are not well defined.

Synchronization across all work-items within a single sub-group is carried out either explicitly using a sub-group function, or implicitly via hierarchical parallelism loops. The explicit sub-group functions carry out collective operations across all the work-items in a sub-group. Available collective operations are: barrier, reduction, broadcast, prefix sum, and evaluation of a predicate. A sub-group function must occur within a converged control flow; i.e. all work-items in the sub-group must encounter precisely the same sub-group function. For example, if a work-group function occurs within a loop, the work- items must encounter the same sub-group function in the same loop iterations. All the work-items of a sub-group must execute the sub-group function and complete reads and writes to memory before any are allowed to continue execution beyond the sub-group function. Synchronization between sub-groups must either be performed using work-group functions, or through memory operations. Using memory operations for sub-group synchronization should be used carefully as forward progress of sub-groups relative to each other is only supported optionally by OpenCL implementations.

A single command-group is added to a queue as a single atomic operation. This means that a single queue can have command-groups added to it from multiple host threads and each command-group is guaranteed to be enqueued as a single operation in a thread-safe way. The start and end of a command group are therefore distinct synchronization points. Command synchronization is defined in terms of these synchronization points.

For command groups that are specified to use fine grained system sharing, the SYCL system has no scheduling information available to it, so no analysis is performed in this case to determine a synchronized schedule and the user must add command-queue synchronization operations to ensure correct data access. For all other memory modes (buffers, images and fine-grained buffer sharing) the SYCL system ensures that the synchronization point enforces correct data ordering.

The synchronization point between a pair of command groups (A and B) assures that results of command group A happens-before command group B is launched. This requires that any updates to memory from command A complete are made available to other commands before the synchronization point completes. Likewise, this requires that command B waits until after the synchronization point before loading values from global memory. The concept of a synchronization point works in a similar fashion for commands such as a barrier that apply to two sets of commands. All the commands prior to the barrier must complete and make their results available to following commands. Furthermore, any commands following the barrier must wait for the commands prior to the barrier before loading values and continuing their execution.

These happens-before relationships are a fundamental part of the SYCL and OpenCL memory model. When applied at the level of commands, they are straightforward to define at a language level in terms of ordering relationships between different commands. Ordering memory operations inside different commands, however, requires rules more complex than can be captured by the high level concept of a synchronization point. These rules are described in detail in section 3.3.6 of the OpenCL 2.2 specification.

2.4 Memory Model

Work-items executing in a kernel have access to four distinct memory regions:

- *Global memory* is accessible to all work-items in all work-groups. Work-items can read from or write to any element of a global memory object. Reads and writes to global memory may be cached depending on the capabilities of the device. Global memory is persistent across kernel invocations, however there is no guarantee that two concurrently executing kernels can simultaneously write to the same memory object and expect correct results.

- *Constant memory* is a region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

- *Local Memory* is a distinct memory region shared between work-items in a single work-group and inaccessible to work-items in other work-groups. This memory region can be used to allocate variables that are shared by all work-items in that work-group. Work-group-level visibility allows local memory to be implemented as dedicated regions of memory on an OpenCL device where this is appropriate.

- *Private Memory* is a region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

The application running on the host can use SYCL buffer objects using instances of the `cl::sycl::buffer` class to allocate memory in the global address space, or can allocate specialized image memory using the `cl::sycl::image` class. In OpenCL, a memory object is attached to a specific context. In SYCL, a `cl::sycl::buffer` or `cl::sycl::image` object can encapsulate multiple underlying OpenCL memory objects and host memory allocations to enable the same buffer or image to be shared between multiple devices in different contexts, and hence different platforms. It is the responsibility of the SYCL implementation to ensure that a buffer or image object shared between multiple OpenCL contexts is moved between contexts using the correct synchronization and copy commands to preserve SYCL memory ordering semantics.

Alternatively, a SYCL application can use one of the OpenCL 2.2 shared virtual memory models. These memory models are described in the OpenCL 2.2 specification.

2.4.1 Access to memory

To access data in buffers or coarse-grained svm allocations inside a kernel, the user must create a `cl::sycl::accessor` object which parameterizes the type of access the kernel requires. The `cl::sycl::accessor` object specifies whether the access is via global memory, a shared virtual memory pointer, constant memory or image samplers (and their associated access functions). The accessor also specifies whether the access is read-only, write-only or read-write. An optional *discard* flag can be added to an accessor to tell the system to discard any previous contents of the data the accessor refers to. Atomic access can also be requested on an accessor which allows `cl::sycl::atomic` classes to be used via the accessor.

It is only possible to pass a pointer into host memory directly as a kernel parameter if the device is an OpenCL 2.2 device. For coarse-grained shared virtual memory, the pointer can only be validly passed as a parameter to the kernel if it is also passed in as a shared virtual memory accessor. For fine grained shared virtual memory, the pointer can only be validly passed as a parameter to the kernel if it is also registered in the command group as used by the kernel.

To allocate local memory within a kernel, the user can either pass a `cl::sycl::local_accessor` object to the kernel as a parameter, or can define a variable in workgroup scope inside `cl::sycl::parallel_for_work_group`.

Any variable defined inside a `cl::sycl::parallel_for` scope or `cl::sycl::parallel_for_work_item` scope will be allocated in private memory. Variables defined in functions called from workgroup scope (i.e. `cl::sycl::parallel_for_work_group` will also be local, while variables defined in functions called from workitem scope (i.e. `cl::sycl::parallel_for` or `cl::sycl::parallel_for_work_item`) will be allocated in private memory. Variables accessed in `cl::sycl::parallel_for_sub_group` scope are compiler-managed and so the specific memory used for these variables is implementation-defined.

Users can create accessors that reference sub-buffers as well as entire buffers.

Within kernels, accessors can be implicitly cast to C++ pointer types. The pointer types will contain a compile-time deduced address space. So, for example, if an accessor to global memory is cast to a C++ pointer, the C++ pointer type will have a global address space attribute attached to it. The address space attribute will be compile-time propagated to other pointer values when one pointer is initialized to another pointer value using a defined mechanism.

When developers need to explicitly state the memory space of a pointer value, one of the explicit pointer classes can be used. There is a different explicit pointer class for each address space: `cl::sycl::local_ptr`, `cl::sycl::global_ptr`, `cl::sycl::private_ptr`, or `cl::sycl::constant_ptr`. An accessor declared with one address space can be implicitly cast to an explicit pointer class for the same address space.

For templates that need to adapt to different address spaces, a `cl::sycl::multi_ptr` class is defined which is templated via a compile-time constant enumerator value to specify the address space.

2.4.2 Memory consistency

OpenCL uses a relaxed memory consistency model, i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times. This also applies to SYCL kernels.

Within kernels, the consistency of memory is that defined by the underlying OpenCL device, as well as the level of consistency specified in the command group `execution_handle`.

Memory consistency for `cl::sycl::buffer` and `cl::sycl::image` objects shared between enqueued commands is enforced at synchronization points derived from completion of enqueued commands. Consistency of such data between the OpenCL runtime and the host program is ensured via copy commands or map and unmap operations.

2.4.3 Atomic operations

Atomic operations can be performed on memory in buffers. The range of atomic operations available on a specific OpenCL device is limited by the atomic capabilities of that device. The `cl::sycl::atomic<T>` must be used for elements of a buffer to provide safe atomic access to the buffer from device code.

2.5 The SYCL programming model

A SYCL program is written in standard C++. Host code and device code is written in the same C++ source file, enabling instantiation of templated kernels from host code and also enabling kernel source code to be shared between host and device.

The C++ features used in SYCL are a subset of the C++14 standard features. Users will need to compile SYCL source code with C++ compilers which support the following C++ features:

- All C++14 features, apart from Run Time Type Information
- Exception handling
- C++11 lambda functions

- C++11 variadic templates
- C++11 template aliases
- C++11 rvalue references
- C++11 `std::function`, `std::string` and `std::vector`, although users can optionally define and use their own versions of these classes, which SYCL can use via template aliases.

SYCL programs are explicitly parallel and expose the full heterogeneous parallelism of the underlying machine model of OpenCL. This includes exposing the data-parallelism, multiple execution devices and multiple memory storage spaces of OpenCL. However, SYCL adds on top of OpenCL a higher level of abstraction allowing developers to hide much of the complexity from the source code, when a developer so chooses.

A SYCL program is logically split into host code and kernels. Host code is standard C++ code, as provided by whatever C++ compiler the developer chooses to use for the host code. The kernels are C++ *functors* (see C++ documentation for an explanation of functors) or C++11 lambda functions which have been designated to be compiled as SYCL kernels. SYCL will also accept OpenCL `cl_kernel` objects.

SYCL programs target heterogeneous systems. The kernels may be compiled and optimized for multiple different processor architectures with very different binary representations.

In SYCL, kernels are contained within command group functors, which include all of the data movement/mapping/copying required to correctly execute the kernel. A command group functor takes as a parameter a command group `execution_handle` class object which provides the interface for kernel invocations and then the functor itself can be submitted to a queue. The command group functor is executed on the host in order to add all the specified commands to the specified queue. Command group handlers can be optionally templated with required capabilities for the command group.

SYCL 2.2 supports the multiple shared virtual memory options of OpenCL. The level of shared virtual memory required for a command group must be specified via template parameters of the `execution_handle` object passed via the queue `submit` operation for the command group.

For OpenCL 1.2 devices, only the basic `buffer`/`image`/ `accessor` model of SYCL 1.2 is supported. All OpenCL 2.2 devices support coarse-grained shared virtual memory (SVM) which is also exposed via buffers and accessors, but the buffers are created using an `svm_allocator`. To use coarse-grained shared virtual memory in SYCL 2.2, users still need to create a `buffer` and a `accessor`, but inside the kernel, users can choose to use the same pointer as is used on the host. Also, to use coarse-grained shared virtual memory, users must create a `context` and ensure that all accesses to the SVM buffer use the same context.

For OpenCL 2.2 devices that support fine-grained buffer shared virtual memory, users must also use an `svm_allocator` to allocate the memory. The same `context` used to allocate the memory must also be used for all command groups that operate on the SVM memory. All command groups that access the fine grained SVM buffer must also register their access in the command group (using the `register_access` method on the handler) so that the scheduler can ensure the correct ordering of execution and the correct data sharing. The data can be accessed by the same pointer as on the host once inside the kernel, if the access is correctly registered.

For OpenCL 2.2 devices that support fine-grained system shared virtual memory, users do not need to do any special allocation of data or access data in any different way than on the host. This memory can be allocated using a normal C++ `new` or `malloc` and then accessed on device via pointers. There is no need to register access to the data inside the command group. However, the SYCL system cannot manage scheduling and synchronization because it has no information about access to data. So, for this mode of memory sharing, the user must manage all synchronization. Users should only specify system sharing in the `execution_handler` and `context` if they know that the devices they are targeting supports this feature and if they are prepared to manage their own synchronization.

2.5.1 Basic data parallel kernels

Data-parallel kernels that execute as multiple work-items and where no local synchronization is required are enqueued with the `cl::sycl::parallel_for` function parameterized by a `cl::sycl::range` parameter. These kernels will execute the kernel function body once for each work-item in the range. The range passed to `cl::sycl::parallel_for` represents the global size of an OpenCL kernel and will be divided into work-groups whose size is chosen by the SYCL runtime. Barrier synchronization is not valid within these work-groups.

2.5.2 Work-group data parallel kernels

Data parallel kernels can also execute in a mode where the set of work-items is divided into work-groups of user-defined dimensions. The user specifies the global range and local work-group size as parameters to the `cl::sycl::parallel_for` function with a `cl::sycl::nd_range` parameter. In this mode of execution, kernels execute over the `nd_range` in work-groups of the specified size. It is possible to share data among work-items within the same work-group in local or global memory and to synchronize between work-items in the same work-group by calling the `barrier` function on an `nd_item` object. All work-groups in a given `parallel_for` will be the same size and the global size defined in the `nd_range` must be a multiple of the work-group size in each dimension.

2.5.3 Hierarchical data parallel kernels

The SYCL compiler provides a way of specifying data parallel kernels that execute within work groups via a different syntax which highlights the hierarchical nature of the parallelism. This mode is purely a compiler feature and does not change the execution model of the kernel. Instead of calling `cl::sycl::parallel_for` the user calls `cl::sycl::parallel_for_work_group` with a `cl::sycl::range` value representing the number of work-groups to launch and optionally a second `cl::sycl::range` representing the size of each work-group for performance tuning. All code within the `parallel_for_work_group` scope effectively executes once per work-group. Within the `parallel_for_work_group` scope, it is possible to call `parallel_for_work_item` which creates a new scope in which all work-items within the current work-group execute. This enables a programmer to write code that looks like there is an inner work-item loop inside an outer work-group loop, which closely matches the effect of the execution model. All variables declared inside the `parallel_for_work_group` scope are allocated in workgroup local memory, whereas all variables declared inside the `parallel_for_work_item` scope are declared in private memory. All `parallel_for_work_item` calls within a given `parallel_for_work_group` execution must have the same dimensions.

For OpenCL 2.2 capable devices, there is a new intermediate level of the hierarchy: `parallel_for_sub_group` which occurs between the work-group and work-item scopes. Its size is controlled by the underlying OpenCL device, but can be queried. SYCL creates a new form of data storage for the `sub_group` scope which allows variables that are shared across a sub group. There is no associated address space for sub-group level variables.

2.5.4 Kernels that are not launched over parallel instances

Simple kernels for which only a single instance of the kernel function will be executed are enqueued with the `cl::sycl::single_task` function. The kernel enqueued takes no "work-item id" parameter and will only execute once. The behavior is logically equivalent to executing a kernel on a single compute unit with a single work-group comprising only one work-item. Such kernels may be enqueued on multiple queues and devices and as a result may, like any other OpenCL entity, be executed in task-parallel fashion.

2.5.5 Synchronization

In SYCL, synchronization can be either global or local within a work-group. The SYCL implementation may need to provide extra synchronization commands and host-side synchronization in order to enable synchronization across OpenCL contexts, but this is handled internally within the SYCL host runtime.

Synchronization between work-items in a single work-group is achieved using a work-group barrier. This matches the OpenCL C behaviour. All the work-items of a work-group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all. There is no mechanism for synchronization between work-groups. In SYCL, workgroup barriers are exposed through a method on the `cl::sycl::nd_item` class, which is only available inside kernels that are executed over workgroups. This ensures that developers can only use workgroup barriers inside workgroups.

For OpenCL 2.2 devices that support sub-groups, there are sub-group barriers along with work-group barriers. Also, OpenCL 2.2 devices support cross-work-group and cross-sub-group operations, which are also synchronization operations as well as enabling operations across all work-items in a work-group or sub-group.

Synchronization points in SYCL are exposed through the following operations:

- *Buffer destruction*: The destructors for `cl::sycl::buffer` and `cl::sycl::image` objects wait for all enqueued work on those objects to complete. If the objects were constructed with attached host memory, then the destructor copies the data back to host memory before returning. More complex forms of synchronization on buffer destruction can be specified by the user by constructing buffers with other kinds of references to memory, such as `shared_ptr` and `unique_ptr`.

- *Accessor construction*: The constructor for a host accessor waits for all kernels that modify the same buffer (or image) in any queues to complete and then copies data back to host memory before the constructor returns. Any command groups submitted to any queue will wait for the accessor to be destroyed.

- *Command group enqueue*: The SYCL scheduler internally ensures that any command groups added to queues have the correct event dependencies added to those queues to ensure correct operation. Adding command groups to queues never blocks. Instead any required synchronization is added to the queue and events of type `handler_event` are returned by the queue's submit function that contain event information related to the specific command group.

- *Interaction with OpenCL synchronization operations*: The user can obtain OpenCL events from command groups, images and buffers which will enable the user to add barrier packets to their own queues to correctly synchronize for buffer or image data dependencies.

- *Queue destruction*: The destructor for `cl::sycl::queue` objects waits for all commands executing on the queue to complete before the destructor returns.

- *Context destruction*: The destructor for `cl::sycl::context` objects waits for all commands executing on any queues in the context to complete before the destructor returns.

- *SYCL event objects*: SYCL provides `cl::sycl::event` objects which can be used for user synchronization. If synchronization is required between two different OpenCL contexts, then the SYCL runtime ensures that any extra host-based synchronization is added to enable the SYCL event objects to operate between contexts correctly.

2.5.6 Error handling

In SYCL, there are two types of error: synchronous errors that can be detected immediately, and asynchronous errors that can only be detected later. Synchronous errors, such as failure to construct an object, are reported immediately by the runtime throwing an exception. Asynchronous errors, such as an error occurring during execution of a kernel on a device, are reported via user-supplied asynchronous error-handlers.

A `cl::sycl::context` can be constructed with a user-supplied asynchronous error handler. If a `cl::sycl::queue` is constructed without a user-supplied context, then the user can supply an asynchronous error handler for the queue, otherwise errors on that queue will be reported to its context error handler.

Asynchronous errors are not reported immediately as they occur. The asynchronous error handler for a context or queue is called with a `cl::sycl::exception_list` object, which contains a list of asynchronously-generated exception objects, either on destruction of the context or queue that the error handler is associated with, or via an explicit `wait_and_throw` method call on an associated queue. This style of asynchronous error handling is similar to that proposed for an upcoming revision of the C++ standard.

2.5.7 Scheduling of kernels and data movement

Within the command group functor, accessor objects specify what data the command group will read and write. When enqueuing a command group functor, the runtime ensures that synchronization operations are also enqueued. By these means it ensures that the reads and writes are semantically equivalent to an in-order execution. Different command groups may execute out-of-order relative to each other, as long as read and write dependencies are enforced.

A command group functor can be submitted either to a single queue to be executed on, or a secondary queue can be provided as well. If a command group functor fails to be enqueued to the primary queue, then the system will attempt to enqueue it to the secondary queue, if given as a parameter to the submit function. If the command group functor fails to be queued to both of these queues, then a synchronous SYCL exception will be thrown.

It is possible that a command group may be successfully enqueued, but then asynchronously fail to run, for some reason. In this case, it may be possible for the runtime system to execute the command group functor on the secondary queue, instead of the primary queue. The situations where a SYCL runtime may be able to achieve this asynchronous fall-back is implementation- defined.

A command group functor at construction takes a command group `handler` as a parameter and anything within that scope is immediately executed and has to get the handler object as a parameter. The intention is that a user will perform calls to SYCL functions, methods, destructors and constructors inside that scope. These calls will be non-blocking on the host, but enqueue operations to the queue the command group is submitted at. All user functions within the command group functor will be called on the host as the command group functor is executed, but any runtime SYCL operations will be queued.

The scheduler must treat command groups atomically. So if two threads simultaneously enqueue two command groups onto the same queue, then each command group must be added to the queue as an atomic operation. The order of two simultaneously enqueued command groups relative to each other is undefined but the constituent commands must not interleave.

Command group functors are scheduled to enforce the ordering semantics of operations on memory objects (both buffers and images). These ordering rules apply regardless of whether the command groups are enqueued in the same context, queue, device or platform. Therefore, a SYCL implementation may need to produce extra synchronization operations between contexts, platforms, devices and queues using OpenCL constructs such as user events. How this is achieved is implementation defined. An implementation is free to re-order or parallelize command groups in queues as long as the ordering semantics on memory objects are not violated.

The ordering semantics on memory objects are:

1. The ordering rules apply based on the totality of accessors constructed in the command group. The order in which accessors are constructed within the command group is not relevant. If multiple accessors in the same command group operate on the same memory object, then the command group's access to that memory object is the union of the access permissions of the accessors.

2. Accessors can be created to operate on *sub-buffers*. A buffer may be overlaid with any number of sub-buffers. If two accessors are constructed to access the same buffer, but both are to non-overlapping sub-buffers of the buffer, then the two accessors are said to not *overlap*, otherwise the accessors do overlap. Overlapping is the test that is used to determine the scheduling order of command groups.

3. If a command group has any accessor with *discard* access to a memory object, then the scheduler does not need to preserve the previous contents of the memory object when scheduling the command group.

4. All other accessors must preserve normal read-write ordering and data access. This means the scheduler must ensure that a command group that reads a memory object must first copy or map onto the device the data that might be read. Reads must follow writes to memory objects or overlapping sub-buffers.

5. It is permissible for command groups that only read data to not copy that data back to the host or other devices after reading and for the scheduler to maintain multiple read-only copies of the data on multiple devices.

In OpenCL, there are in-order queues and out-of-order queues. In SYCL, the SYCL queues are scheduled irrespective of the underlying OpenCL queues, maintaining the SYCL in-order execution semantics.

It is worth noting that a SYCL queue does not necessarily map to only one OpenCL queue, however, the OpenCL queue that is given when interacting with the SYCL queue will retain any synchronization information is needed for synchronization with any other OpenCL queues spawned by the system.

An OpenCL implementation can require different queues for different devices and contexts. The synchronization required to ensure order between commands in different queues varies according to whether the queues have shared contexts. A SYCL implementation must determine the required synchronization to ensure the above ordering rules above are enforced.

SYCL provides *host accessors*. These accessors give temporary access to data in buffers on the host, outside the command group. Host accessors are the only kinds of accessors that can be created outside command groups. Creation of a host accessor is a blocking operation: all command groups that read or write data in the buffer or image that the host accessor targets must have completed before the host thread will continue. All data being written in an enqueued command group to the buffer or image must be completed and written to the associated host memory before the host accessor constructor returns. Any subsequently enqueued command group that accesses overlapping data in the buffer or image of the host accessor will block and not start execution until the host accessor (and any copies) has been destroyed. This approach guarantees that there is no concurrent access to a memory object between the host thread and any SYCL device.

If a user creates a SYCL buffer, image or accessor from an OpenCL object, then the SYCL runtime will correctly manage synchronization and copying of data between the OpenCL memory object for the lifetime of the SYCL buffer, image or accessor constructed from it. If a user makes use of the underlying OpenCL memory object at the same time as a SYCL buffer, image or accessor is live, then the behaviour is undefined.

2.5.8 Managing object lifetimes

SYCL does not initialize any OpenCL features until a `cl::sycl::context` object is created. A user does not need to explicitly create a `cl::sycl::context` object, but they do need to explicitly create a `cl::sycl::queue` object, for which a `cl::sycl::context` object will be implicitly created if not provided by the user.

All OpenCL objects encapsulated in SYCL objects are reference-counted and will be destroyed once all references have been released. This means that a user needs only create a SYCL queue (which will automatically create an OpenCL context) for the lifetime of their application to initialize and release the OpenCL context safely.

When an OpenCL object that is encapsulated in a SYCL object is copied in C++, then the underlying OpenCL object is not duplicated, but its OpenCL reference count is incremented. When the original or copied SYCL object is destroyed, then the OpenCL reference count is decremented.

There is no global state specified to be required in SYCL implementations. This means, for example, that if the user creates two queues without explicitly constructing a common context, then a SYCL implementation does not have to create a shared context for the two queues. Implementations are free to share or cache state globally for performance, but it is not required.

Memory objects can be constructed with or without attached host memory. If no host memory is attached at the point of construction, then destruction of that memory object is non-blocking. The user may use C++ standard pointer classes for sharing the host data with the user application and for defining blocking, or non-blocking behavior of the buffers and images.

If host memory is attached by using a raw pointer, then the default behavior is followed, which is that the destructor will block until any command groups operating on the memory object have completed, then, if the contents of the memory object is modified on a device those contents are copied back to host and only then does the destructor return. Instead of a raw pointer, a `unique_ptr` may be provided, which uses move semantics for initializing and using the associated host memory. In this case, the behavior of the buffer in relation to the user application will be non-blocking on destruction. In the case where host memory is shared between the user application and the SYCL runtime, then the reference counter of the `shared_ptr` is determining whether the buffer needs to copy data back on destruction and in that case the blocking or non-blocking behavior depends on the user application.

The only blocking operations in SYCL (apart from explicit wait operations) are:

- Host accessor constructor, which waits for any kernels enqueued before its creation that write to the corresponding object to finish and be copied back on host memory before it starts processing. The host accessor does not necessarily copy back to the same host memory as the one initially given by the user.

- Memory object destruction, in the case where copies back to host memory have to be done
- Queue destruction, as all enqueued kernels need to finish executing first.
- Context destruction, as all enqueued kernels need to finish executing first.
- Device destruction, as all enqueued kernels need to finish executing first.
- Platform destruction, as all enqueued kernels need to finish executing first.

2.5.9 Device discovery and selection

A user specifies which queue to submit a command group functor on and each queue is targeted to run on a specific device (and context). A user can specify the actual device on queue creation, or they can specify a *device selector* which causes the SYCL runtime to choose a device based on the user's provided preferences. Specifying a selector causes the SYCL runtime to perform device discovery. No device discovery is performed until a SYCL selector is passed to a queue constructor. Device topology may be cached by the SYCL runtime, but this is not required.

Device discovery will return both OpenCL devices and platforms as well as a SYCL host platform and SYCL host device. The host device allows queue creation and running of kernels, but does not support OpenCL-specific features. It is an error for a user to request an underlying OpenCL device for the SYCL host device.

2.5.10 Interfacing with OpenCL

All SYCL objects which encapsulate an OpenCL object (such as contexts or queues) can be constructed from the OpenCL object. The constructor takes one argument, the OpenCL object, and performs an OpenCL `retain` operation on the OpenCL object to increase its reference count. The destructor for the SYCL object performs an OpenCL `release` operation on the OpenCL object. The copy construction semantics of the SYCL object ensure that each new SYCL copy of the object also does an OpenCL retain on the underlying object.

To obtain the underlying OpenCL object from a SYCL object, there is a `get` method on all relevant SYCL objects. The `get` method returns the underlying OpenCL object and also performs a `retain` operation on the object. It is the user's responsibility to `release` the OpenCL object when the user has finished with it.

SYCL images and buffers are treated differently in that SYCL image and buffer objects do not refer to an OpenCL context and may reference multiple underlying OpenCL image or buffer objects as well as host allocations. It is the accessors to the image and buffer objects that refer to an actual OpenCL context. Accessors provide synchronization in place of the events that the OpenCL runtime would use directly. Therefore, obtaining OpenCL `cl_mem` objects from SYCL is achieved via special accessor classes which can return OpenCL `cl_mem` and `cl_event` objects. SYCL memory objects can be constructed from `cl_mem` objects, but the SYCL system is free to copy from the OpenCL memory object into another memory object or host memory, to achieve normal SYCL semantics, for as long as the SYCL memory object is live.

No SYCL object is guaranteed to have only one underlying OpenCL object created, however, every SYCL object is required to have an OpenCL object which an OpenCL program can interface with and have all synchronization points refer to it.

2.6 Anatomy of a SYCL application

Below is an example of a typical SYCL application which schedules a job to run in parallel on any OpenCL device.

```
 1  #include <CL/sycl.hpp>
 2  #include <iostream>
 3
 4  int main() {
 5    using namespace cl::sycl;
 6
 7    int data[1024];  // initialize data to be worked on
 8
 9    // By including all the SYCL work in a {} block, we ensure
10    // all SYCL tasks must complete before exiting the block
11    {
12      //  create a queue to enqueue work to
13      queue myQueue;
14
15      // wrap our data variable in a buffer
16      buffer<int, 1> resultBuf(data, range<1>(1024));
17
18      // create a command group to issue commands to the queue
19      myQueue.submit([&](execution_handle& cgh) {
20        // request access to the buffer
21        auto writeResult = resultBuf.get_access<access::write>(cgh);
22
23        // enqueue a prallel_for task
24        cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
25          writeResult[idx] = static_cast<int>(idx[0]);
26        });  // end of the kernel function
27      });    // end of our commands for this queue
28    }        // end of scope, so we wait for the queued work to complete
29
30    // print result
31    for (int i = 0; i < 1024; i++)
32      std::cout<<"data["<<i<<"] = "<<data[i]<<std::endl;
33
34    return 0;
35  }
```

At line 1, we “#include” the SYCL header files, which provide all of the SYCL features that will be used.

A SYCL application has three scopes which specify the different sections; *application scope*, *command group scope* and *kernel scope*. The *kernel scope* specifies a single kernel function that will be, or has been, compiled by a *device compiler* and executed on a *device*. In this example *kernel scope* is defined by lines 23 to 25. The *command group scope* specifies a unit of work which will comprise of a *kernel function* and *accessors*. In this example *command group scope* is defined by lines 18 to 26. The *application scope* specifies all other code outside of a *command group scope*. These three scopes are used to control the application flow and the construction and lifetimes of the various objects used within SYCL.

A *kernel function* is the scoped block of code that will be compiled using a device compiler. This code may be defined by the body of a lambda function, by the `operator()` function of a function object or by the binary `cl_kernel` entity generated from an OpenCL C string. Each instance of the *kernel function* will be executed as a single, though not necessarily entirely independent, flow of execution and has to adhere to restrictions on what operations may be allowed to enable device compilers to safely compile it to a range of underlying devices.

The `parallel_for` function is templated with a class, in this case called `class simple_test`. This class is used only as a name to enable the kernel (compiled with a device compiler) and the host code (possibly compiled with a different host compiler) to be linked. This is required because C++ lambda functions have no name that a linker could use to link the kernel to the host code.

The `parallel_for` method creates an instance of a kernel object. The kernel object is the entity that will be enqueued within a `command_group`. In the case of `parallel_for` the *kernel function* will be executed over the given range from 0 to 1023.

A *kernel function* can only be defined within a *command group scope*. Command group scope is the syntactic scope wrapped by the construction of a command group functor object as seen on line 18. The command group functor takes as a parameter a command group `handler` which is a runtime constructed object. The command group functor contains all the the operations to be enqueued on the queue this functor will be submitted to. In this case the constructor used for `myQueue` on line 13 is the default constructor, which allows the queue to select the best underlying device to execute on, leaving the decision up to the runtime.

In SYCL, data that is required within a *kernel function* must be contained within a *buffer* or *image*. We construct a buffer on line 16. Access to the buffer is controlled via an *accessor* which is constructed on line 21 through the `get_access` method of the buffer. The *buffer* is used to keep track of access to the data and the *accessor* is used to request access to the data on a queue, as well as to track the dependencies between *kernel functions*. In this example the *accessor* is used to write to the data buffer on line 25. All *buffers* must be constructed in the *application scope*, whereas all *accessors* must be constructed in the *command group scope*.

2.7 Memory objects

Memory objects in SYCL fall into one of three categories: *buffer* objects shared virtual memory *allocations* and *image* objects. A buffer object stores a one-, two- or three-dimensional collection of elements that are stored linearly directly back to back in the same way C or C++ stores arrays. An SVM allocation creates a region of memory referenced via a pointer where the pointer itself can be shared between the host and OpenCL devices, provided that all command groups accessing the data via this pointer are within the same `context`. An image object is used to store a one-, two- or three-dimensional texture, frame- buffer or image that may be stored in an optimized and device-specific format in memory and must be accessed through specialized operations.

Elements of a buffer object can be a scalar data type (such as an int, float), vector data type, or a user-defined structure. In SYCL, a buffer object is a templated type (`cl::sycl::buffer`), parameterized by the element type and number of dimensions. An image object is stored in one of a limited number of formats. The elements of an image object are selected from a list of predefined image formats which are provided by an underlying OpenCL implementation. Images are encapsulated in the `cl::sycl::image` type, which is templated by the number of dimensions in the image. The minimum number of elements in a memory object is one.

The fundamental differences between a buffer and an image object are:

- Elements in a buffer are stored in an array of 1, 2 or 3 dimensions and can be accessed using an accessor by a kernel executing on a device. The accessors for kernels can be converted within a kernel into C++ pointer types, or the `cl::sycl::global_ptr`, `cl::sycl::constant_ptr` classes. Elements of an image are stored in a format that is opaque to the user and cannot be directly accessed using a pointer. SYCL provides image accessors and samplers to allow a kernel to read from or write to an image.

- For a buffer object, the data is stored in the same format as it is accessed by the kernel, but in the case of an image object the data format used to store the image elements may not be the same as the data format used inside the kernel. Image elements are always a 4-component vector (each component can be a float or signed/unsigned integer) in a kernel. The SYCL accessor and sampler methods to read from an image convert an image element from the format it is stored into a 4-component vector. Similarly, the SYCL accessor methods provided to write to an image convert the image element from a 4-component vector to the appropriate image format specified such as 4 8-bit elements, for example.

Both buffers and images may have one or more underlying OpenCL `cl_mem` objects. When a buffer or image is allocated on more than one OpenCL device, if these devices are on separate contexts then multiple `cl_mem` objects may be allocated for the memory object, depending on whether the object has actively been used on these devices yet or not.

Users may want fine-grained control of the synchronization, memory management and storage semantics of SYCL image or buffer objects. For example, a user may wish to specify the host memory for a memory object to use, but may not want the memory object to block on destruction.

Depending on the control and the use cases of the SYCL applications, well established C++ classes and patterns can be used for reference counting and sharing data between user applications and the SYCL runtime. For control over memory allocation on the host and mapping between host and device memory, C++ `allocator` classes are used that can be the pre-defined ones or user-defined. For better control of synchronization between a SYCL and a non SYCL application that share data `shared_ptr` and `mutex` classes are used. In the case where the user would not like the host side to block on destruction of buffers or images, as the data given to the buffers are for initialization only, the `unique_ptr` class can be used instead of a raw pointer to data.

For shared virtual memory allocations, as the user is provided with a pointer, it is the user's responsibility to manage allocation and deallocation (just as a user would have to with a normal C++ pointer). This can be achieved with C++ pointer types, such as `shared_ptr`. Also, the user must ensure that the pointer is only used in command groups that are within the same context.

2.8 SYCL for OpenCL Framework

The SYCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

- *SYCL C++ Template Library*: The template library layer provides a set of C++ templates and classes which provide the programming model to the user. It enables the creation of queues, buffers and images, as well as access to some underlying OpenCL features such as contexts, platforms, devices and program objects.

- *SYCL Runtime*: The SYCL runtime interfaces with the underlying OpenCL implementations and handles scheduling of commands in queues, moving of data between host and devices, manages contexts, programs, kernel compilation and memory management.

- *OpenCL Implementation(s)*: The SYCL system assumes the existence of one or more OpenCL implementations available on the host machine. If no OpenCL implementation is available, then the SYCL implementation provides only a SYCL-specific host device to run kernels on.

- *SYCL Device Compiler(s)*: The SYCL device compilers compile SYCL C++ kernels into a format which can be executed on an OpenCL device at runtime. There may be more than one SYCL device compiler in a SYCL implementation. The format of the compiled SYCL kernels is not defined. A SYCL device compiler may, or may not, also compile the host parts of the program.

2.9 SYCL device compiler

To enable SYCL to work on a variety of platforms, with different devices, operating systems, build systems and host compilers, SYCL provides a number of options to implementers to enable the compilation of SYCL kernels for devices, while still providing a unified programming model to the user.

2.9.1 Building a SYCL program

A SYCL program runs on a *host* and one or more OpenCL devices. This requires a compilation model that enables compilation for a variety of targets. There is only ever one host for the SYCL program, so the compilation of the source code for the host must happen once and only once. Both kernel and non-kernel source code is compiled for host.

The design of SYCL enables a single SYCL source file to be passed to multiple, different compilers. This is an implementation option and is not required. What this option enables is for an implementer to provide a device compiler only and not have to provide a host compiler. A programmer who uses such an implementation will compile the same source file twice: once with the host compiler of their choice and once with a device compiler. This approach allows the advantages of having a single source file for both host code and kernels, while still allowing users an independent choice of host and SYCL device compilers.

Only the kernels are compiled for OpenCL devices. Therefore, any compiler that compiles only for one or more devices must not compile non-kernel source code. Kernels are contained within C++ source code and may be dependent on lambda capture and template parameters, so compilation of the non-kernel code must determine lambda captures and template parameters, but not generate device code for non-kernel code.

Compilation of a SYCL program may follow either of the following options. The choice of option is made by the implementer:

1. *Separate compilation*: One or more device compilers compile just the SYCL kernels for one or more devices. The device compilers all produce header files for interfacing between the compiler and the runtime, which are integrated together with a tool that produces a single header file. The user compiles the source file with a normal C++ host compiler for their platform. The user must ensure that the host compiler is given the correct command-line arguments (potentially a macro) to ensure that the device compiler output header file is `#include`d from inside the SYCL header files.

2. *Single-source compiler*: In this approach, a single compiler may compile an entire source file for both host and one or more devices. It is the responsibility of the single-source compiler to enable kernels to be compiled correctly for devices and enqueued from the host.

An implementer of SYCL may choose an implementation approach from the options above.

2.9.2 Naming of kernels

SYCL kernels are extracted from C++ source files and stored in an implementation-defined format. When the SYCL runtime needs to enqueue a SYCL kernel, it is necessary for the runtime to load the kernel and pass it to an OpenCL runtime. This requires the kernel to have a globally-visible name to enable an association between the kernel invocation and the kernel itself. The association is achieved using a *kernel name*, which is a C++ typename. For a functor, the kernel name can be the same type as the functor itself, as long as the functor type is globally accessible. For a lambda function, there is no globally-visible name, so the user must provide one. In SYCL, the name is provided as a template parameter to the kernel invocation, e.g. `parallel_for<kernelname>`.

A device compiler should detect the kernel invocations (e.g. `parallel_for<kernelname>`) in the source code and compile the enclosed kernels, storing them with their associated type name. For details please refer to 9.2. The user can also extract OpenCL `cl_kernel` and `cl_program` objects for kernels by providing the typename of the kernel.

2.10 Language restrictions in kernels

The SYCL kernels are executed on SYCL devices and all of the functions called from a SYCL kernel is going to be compiled for the device by a SYCL device compiler. Due to restrictions of different versions of OpenCL capable devices, there are certain restrictions for SYCL kernels that vary by device. Those restrictions can be summarized as: kernels cannot include RTTI information, exception classes, recursive code, virtual functions or make use of C++ libraries that are not compiled for the device. For more details on language restrictions please refer to 9.3.

SYCL kernels use parameters that are captured by value in the command group scope described in 5.1 or are passed from the host to the device using the data management runtime classes of `cl::sycl::accessors`. Sharing data structures between host and device code imposes certain restrictions, such as you can only use user defined classes that are *C++11 standard layout* classes for the data structures.

For OpenCL 1.2 devices (which do not support shared virtual memory), no pointers initialized for host can be used on the device. For these devices, the only way of passing pointers to a kernel is through the usage of `cl::sycl::accessor` class, which supports the `cl::sycl::buffer` and `cl::sycl::image` classes. No hierarchical structures of these classes are supported and any other data containers need to be converted to the SYCL data management classes using the SYCL interface. For more details on the rules for kernel parameter passing, please refer to 5.9.

For OpenCL 2.2 devices (which do support shared virtual memory), pointers can be passed as parameters to kernels. However, there are restrictions on how the memory those pointers point to can be created that is dependent on the level of shared virtual memory that the device supports and that the user has requested.

Some types in SYCL vary according to pointer size or vary on the host according to the host ABI, such as `size_t` or `long`. It is the responsibility of the SYCL device compiler to ensure that the sizes of these types match the sizes on the host, to enable data of these types to be shared between host and device.

The OpenCL C function qualifier `__kernel` and the access qualifiers: `__read_only`, `__write_only` and `__read_write` are not exposed in SYCL via keywords, but instead encapsulated in SYCL's parameter passing system inside accessors. Users wishing to achieve the OpenCL equivalent of these qualifiers in SYCL should instead use SYCL accessors with equivalent semantics.

2.10.1 SYCL Linker

In SYCL only offline linking is supported for SYCL and OpenCL programs and libraries. In the case of linking C++ functions and methods to a SYCL application, where the definitions and declarations are not available in the same translation unit of the compiler, then the macro `SYCL_EXTERNAL` has to be provided.

Any OpenCL C function included in a pre-built OpenCL library can be defined as `extern "C"` function and the the OpenCL program has to be linked against any SYCL program that contains kernels using the external function. In this case, the data types used have to comply with the interoperability data types defined in 4.43.

2.10.2 Functions and datatypes available in kernels

Inside kernels, the functions and datatypes available are restricted by the underlying capabilities of OpenCL devices. All OpenCL C features are provided by C++ classes and functions, which are available on host and device.

2.11 Execution of kernels on the SYCL host device

SYCL enables kernels to run on either the host device or on OpenCL devices. When kernels run on an OpenCL device, then the features and behaviour of that execution follows the OpenCL specification, otherwise they follow the behaviour specified for the SYCL host device.

Any kernel enqueued to a host queue executes on the host device according to the same rules as the OpenCL devices.

Kernel math library functions on the host must conform to OpenCL math precision requirements.

The range of image formats supported by the host device is implementation- defined, but must match the minimum requirements of the OpenCL specification.

Some of the OpenCL extensions and optional features may be available on a SYCL host device, but since these are optional features and vendor specific extensions, the user must query the host device to determine availability. A SYCL implementer must state what OpenCL device features are available on their host device implementation.

The synchronization and data movement that occurs when a kernel is executed on the host may be implemented in a variety of ways on top of OpenCL. The actual mechanism is implementation-defined.

3. SYCL Runtime Library

The SYCL programming interface provides a C++ abstraction of the OpenCL 2.2 functionality and feature set. This section describes all the available classes and interfaces of SYCL, focusing on the C++ interface of the underlying runtime. In this section, we define all the classes and methods for the SYCL API which are available for SYCL host and OpenCL devices. This section also describes the synchronization rules and OpenCL API interoperability rules which guarantee that all the methods, including constructors, of the SYCL classes are thread safe.

It is assumed that the OpenCL API is also available to the developer at the same time as SYCL.

3.1 Header files and namespaces

SYCL provides one standard header file: `"CL/sycl.hpp"`, which needs to be included in every SYCL program.

All SYCL classes, constants, types and functions are defined within the `cl::sycl` namespace.

3.2 C++ Standard library classes required for the interface

The SYCL programming interfaces make extensive use of vectors, strings and function objects to carry information. Moreover, SYCL provides user-customizable smart pointer and mutex classes which default to using the STL string, vector, function, mutex and smart pointer classes, unless defined otherwise. These types are exposed internally as `cl::sycl::vector_class`, `cl::sycl::string_class`, `cl::sycl::function_class`, `cl::sycl::mutex_class`, `cl::sycl::unique_ptr_class`, `cl::sycl::shared_ptr_class`, `cl::sycl::weak_ptr_class` and `cl::sycl::future`.

It is possible to disable the STL versions of these classes when required. A common reason for doing this is to specify a custom allocator to move memory management under the control of the SYCL user. This is achieved by defining `CL_SYCL_NO_STD_VECTOR`, `CL_SYCL_NO_STD_STRING`, `CL_SYCL_NO_STD_FUNCTION`, `CL_SYCL_NO_STD_MUTEX`, `CL_SYCL_NO_UNIQUE_PTR`, `CL_SYCL_NO_SHARED_PTR`, `CL_SYCL_NO_WEAK_PTR`, `CL_SYCL_NO_FUTURE`, respectively, before including `"CL/sycl.hpp"`, and by replacing the template aliases in the `cl::sycl` namespace as necessary.

```
#include <functional>
#include <memory>
#include <mutex>
#include <string>
#include <vector>

namespace cl {
namespace sycl {
    #define CL_SYCL_NO_STD_VECTOR
    template < class T, class Alloc = std::allocator<T> >
    using vector_class = std::vector<T, Alloc>;

    #define CL_SYCL_NO_STD_STRING
    using string_class = std::string;

    #define CL_SYCL_NO_STD_FUNCTION
    template<class R, class... ArgTypes>
    using function_class = std::function<R(ArgTypes...)>;

    #define CL_SYCL_NO_STD_MUTEX
    using mutex_class = std::mutex;

    #define CL_SYCL_NO_STD_UNIQUE_PTR
    template <class T, class D = std::default_delete<T>>
    using unique_ptr_class = std::unique_ptr<T[], D>;

    #define CL_SYCL_NO_STD_SHARED_PTR
    template <class T>
    using shared_ptr_class = std::shared_ptr<T>;

    #define CL_SYCL_NO_STD_WEAK_PTR
    template <class T>
    using weak_ptr_class = std::weak_ptr<T>;

}  // sycl
}  // cl

#include <CL/sycl.hpp>
```

3.3 Device selection class

The class `device_selector` is a functor which enables the SYCL runtime to choose the best device based on heuristics specified by the user, or by one of the built-in device selectors. The built-in device selectors are listed in Table 3.3. The `device_selector` constructors and methods are described in tables 3.1 and 3.2.

```
namespace cl {
namespace sycl {
class device_selector {
 public:
  device_selector();

  device_selector(const device_selector &selector);

  virtual ~device_selector();

  device select_device() const;

  virtual int operator()(const device &device) const = 0;
};
}  // namespace sycl
}  // namespace cl
```

Constructors	Description
`device_selector()`	Default device selector constructor for the abstract class.
`device_selector(const device_selector &selector)`	Copy constructor.
	End of table

Table 3.1: Constructors of the `device_selector` class

Methods	Description
`device select_device()const`	Returns a selected device using the functor operator defined in sub-classes `operator()(const device &device)`.
`virtual int operator()(const device &device)const`	This pure virtual operator allows the customization of device selection. It defines the behavior of the `device_selector` functor called by the SYCL runtime on device selection. It returns a “score” for each device in the system and the highest rated device will be used by the SYCL runtime.
	End of table

Table 3.2: Methods for the `device_selector` class

`operator()` is an abstract method which returns a “score” per-device. At the stage where the SYCL runtime selects a device, the system will go through all the available devices in the system and choose the one with the highest score as computed by the current device selection class. If a device has a negative score it will never be chosen. While OpenCL devices may or may not be available, the SYCL host device is always available, so the developer is able to choose the SYCL host device as a fall-back device. Selection of the SYCL host device will allow execution of CPU-compiled versions of kernels scheduled on queues created against that device.

The system also provides built-in device selectors, including selectors which choose a device based on the default behavior of the system. An important note is that the system is not required to have global state and its behavior is defined by the platforms the developer chooses to target.

The *default_selector* is the selector that incorporates the *default* behavior of the system, and it is implicitly used by the system for the creation of the queue when no other *device_selector* or underlying OpenCL identifier is provided. The method the default selector uses to rank and select devices is implementation-defined. The *default_selector* will choose the SYCL host device if there are no OpenCL devices available.

SYCL device selectors	Description
`default_selector`	Devices selected by the heuristics of the system. If no OpenCL device is found then it defaults to the SYCL host device.
	Continued on next page

Table 3.3: Standard device selectors included with all SYCL implementations.

SYCL device selectors	Description
`gpu_selector`	Select devices according to device type `info::device::device_type::gpu` from all the available OpenCL devices. If no OpenCL GPU device is found, the selector fails.
`cpu_selector`	Select devices according to device type `info::device_type::cpu` from all the available devices and heuristics. If no OpenCL CPU device is found, the selector fails.
`host_selector`	Selects the SYCL host CPU device that does not require an OpenCL runtime.
`capabilities_selector(exec_capabilities = exec_capabilities::opencl22core)`	Selects the devices according to the capabilities the application needs to expose. The default capabilities for SYCL 2.2 are the `exec_capabilities::opencl22core`, which expose the core functionality of OpenCL 2.2. The available values are: • `exec_capabilities::opencl12`, for OpenCL 1.2 execution capabilities, • `exec_capabilities::opencl22`, for core OpenCL 2.2 execution capabilities, • `exec_capabilities::svm_coarse_grain`, for coarse-grain buffer shared virtual memory, • `exec_capabilities::svm_fine_buffer_sharing`, for fine-grain buffer shared virtual memory without atomics, • `exec_capabilities::svm_fine_buffer_sharing_atomics`, for fine-grain buffer shared virtual memory with atomics, • `exec_capabilities::svm_fine_system_sharing_atomics`, for fine-grain system shared virtual memory with atomics.
`svm::fine_grain_buffer_selector`	Selects the devices that support fine grain shared virtual memory with buffer sharing and no atomics. If no such devices are found then selector fails.
`svm::fine_grain_buffer_atomics_selector`	Selects the devices that support fine grain shared virtual memory with buffer sharing and atomics. If no such devices are found then selector fails.

Continued on next page

Table 3.3: Standard device selectors included with all SYCL implementations.

SYCL device selectors	Description
svm::fine_grain_system_selector	Selects the devices that support fine grain shared virtual memory no atomics. If no such devices are found then selector fails.
svm::fine_grain_system_atomics_selector	Selects the devices that support fine grain shared virtual memory with system sharing and atomics. If no such devices are found then the selector fails.
	End of table

Table 3.3: Standard device selectors included with all SYCL implementations.

3.4 Platform class

The `platform` class represents a SYCL platform: a collection of related SYCL supported devices. Each platform may be either an OpenCL platform, or it may be the SYCL host platform, containing only the SYCL host device. The host platform is the default platform and can be useful when no OpenCL platform is available or during the development process, especially for debugging. The platform class offers a selection of static methods to obtain information about the platforms available at runtime. The SYCL host platform reports itself as a valid SYCL platform. The constructors and methods of the `platform` class are listed in Tables3.4 and 3.5.

```
namespace cl {
namespace sycl {
namespace info {
enum class device_type : unsigned int {
  cpu,
  gpu,
  accelerator,
  custom,
  defaults,
  host,
  all
}
}  // info

class platform {
 public:
  platform();

  explicit platform(cl_platform_id platformID);

  explicit platform(device_selector &devSelector);

  platform(const platform &rhs);

  platform &operator=(const platform &rhs);

  ~platform();

  //The OpenCL cl_platform_id or nullptr for SYCL host.
  cl_platform_id get() const;

  // Returns all the available OpenCL platforms and the SYCL host platform
  static vector_class<platform> get_platforms();

  //Returns the devices available in this platform
  vector_class<device> get_devices(
    info::device_type = info::device_type::all) const;

  //Returns the corresponding descriptor information for all SYCL platforms
  //(OpenCL and host)
  template <info::platform param>
  typename info::param_traits<info::platform, param>::type get_info()
    const;

  //Returns the available extensions for all SYCL platforms( OpenCL and host)
  bool has_extension(const string_class &extension) const;

  //True if the platform is host
  bool is_host() const;
};
}  // namespace sycl
}  // namespace cl
```

The SYCL host platform is not an OpenCL platform. The *get()* method will trigger an *invalid_object_error* exception of type *device_error*. The SYCL host platform will be included in the output of the static function *get_platforms*.

Constructors	**Description**
`platform()`	Default constructor for platform, which corresponds to the host platform. Returns errors via the SYCL exception class.
`explicit platform(cl::platform platformId)`	Construct a platform object from an OpenCL platform id. Returns errors via the SYCL exception class.
`explicit platform(const device_selector &devSelector)`	Construct a platform object from the device returned by a device selector of the user's choice. Returns errors via the SYCL exception class.
`platform(const platform &rhs)`	Copy constructor.
`platform &operator=(const platform &rhs)`	Assignment operator.
	End of table

Table 3.4: Constructors of `platform` class

The default constructor will create an instance of the platform class where the underlying platform will be the SYCL host platform by default.

Methods	Description
`cl::platform get ()const`	Returns the *cl::platform* of the underlying OpenCL platform. If the platform is not a valid OpenCL platform, for example if it is the SYCL host, an *invalid_object_error* exception will be triggered of type *device_error*.
`static vector_class<platform> get_platforms ()const`	Returns all available platforms in the system.
`vector_class<device> get_devices( info::device_type = info::device_type::all)const`	Returns all the available devices for this platform, of type *device_type*, which is defaulted to `info::device_type::all`
`template <info::platform param> typename info::param_traits<info::platform, param>::type get_info()const`	Queries OpenCL information for the underlying *cl::platform*.
`bool has_extension(const string_class & extension) const`	Specifies whether a specific extension is supported on the platform.
`bool is_host()const`	Returns true if this is a SYCL host platform.
	End of table

Table 3.5: Methods of `platform` class

3.4.1 Platform information descriptors

A SYCL platform can be queried for all of the following information using the `get_info` function. All SYCL contexts have valid devices for them, including the SYCL host device. The available information is in table 3.6. The interface of all available platform descriptors in the appendix C.1.

Platform Descriptors	Return type	Description
`info::platform::profile`	`string_class`	Returns the profile name supported by the implementation. Can be either FULL_PROFILE or EMBEDDED_PROFILE.
`info::platform::version`	`string_class`	OpenCL software driver version string in the form major_number.minor_number
`info::platform::name`	`string_class`	Name of the platform.
`info::platform::vendor`	`string_class`	String provided by the platform vendor.
`info::platform::extensions`	`string_class`	A space-separated list of extension names supported by the platform.
`info::platform::host_timer_resolution`	`cl_ulong`	Returns the resolution of the host timer in nanoseconds.
		End of table

Table 3.6: Platform information descriptors.

3.5 Context class

The *context* class encapsulates an OpenCL context, which is implicitly created and the lifetime of the context instance defines the lifetime of the underlying OpenCL context instance. On destruction *clReleaseContext* is called. The default context is the SYCL host context containing only the SYCL host device.

The constructors and methods of the `context` class are listed in Tables 3.7 and 3.8.

3.5.1 Context interface

```
namespace cl {
namespace sycl {
class context {
 public:
  context();

  explicit context(async_handler asyncHandler = nullptr);

  context(cl_context clContext, async_handler asyncHandler = nullptr);

  context(const device_selector &deviceSelector,
          info::gl_context_interop interopFlag,
          async_handler asyncHandler = nullptr);

  context(const device &dev, info::gl_context_interop interopFlag,
          async_handler asyncHandler = nullptr);

  context(const platform &plt, info::gl_context_interop interopFlag,
          async_handler asyncHandler = nullptr);

  context(const vector_class<device> &deviceList, info::gl_context_interop interopFlag,
          async_handler asyncHandler = nullptr);

  context(const context &rhs);

  context &operator=(const context &rhs);

  ~context();

  cl_context get() const;

  bool is_host() const;

  platform get_platform() const;

  vector_class<device> get_devices() const;

  template <info::context param>
  typename param_traits<info::context, param>::type get_info() const;
};
} // namespace sycl
} // namespace cl
```

Constructors	Description
`context ()`	Default constructor that creates a SYCL host context. Returns synchronous errors via the SYCL exception class.
`explicit context(async_handler a_handler = nullptr)`	Constructs a context object for SYCL host using an *async_handler* 5.11.2 for handling asynchronous errors.
`context(const capability_selector &device_selection, async_handler a_handler = nullptr)`	Constructs a context using the `capability_selector` object. The context is constructed using implementation heuristics for choosing all of the devices belonging to a platform that satisfy the capability_selector requirements.
`context(const device_selector &device_selection, info::gl_context_interop flag, async_handler a_handler = nullptr)`	Constructs a context object using a `device_selector` object. The context is constructed with a single device retrieved from the `device_selector` object provided. Returns synchronous errors via the SYCL exception class and asynchronous errors are handled via the *async_handler* 5.11.2, if provided.
`context(const device &dev, info::gl_context_interop flag, async_handler a_handler = nullptr)`	Constructs a context object using a device object. Returns synchronous errors via the SYCL exception class and asynchronous errors are handled via the *async_-handler*5.11.2, if provided.
`context(const platform &plt, info::gl_context_interop flag, async_handler a_handler = nullptr)`	Constructs a context object using a platform object. Returns synchronous errors via the SYCL exception class and asynchronous errors are handled via the *async_handler* 5.11.2, if provided.
`context(const vector_class<device> & deviceList, info::gl_context_interop flag, async_handler a_handler = nullptr)`	Constructs a context object using a vector_class of device objects. Returns synchronous errors via the SYCL exception class and asynchronous errors are handled via the *async_handler* 5.11.2, if provided.
`context (cl::context context, async_handler a_handler = nullptr)`	Context constructor, where the underlying OpenCL context is given as a parameter. The constructor executes a retain on the *cl::context*. Returns synchronous errors via the SYCL exception class and asynchronous errors are handled via the *async_handler* 5.11.2, if provided.
`context(const context &rhs)`	Constructs a context object from another context object and retains the *cl_context* object if the context is not SYCL host.
	End of table

Table 3.7: Constructors of the `context` class.

Methods	Description
`cl::context get ()const`	Returns the underlying *cl_context* object, after retaining the cl_context. Retains a reference to the returned `cl_context` object. Caller should release it when finished. If the context is a SYCL host context, then an *invalid_object_error* exception of the type *device_error*.
`bool is_host ()const`	Specifies whether the context is in SYCL Host Execution Mode.
`template <info::context param> typename param_traits <info::context, param>::type get_info ()const`	Queries OpenCL information for the underlying *cl_context*.
`platform get_platform()const`	Returns the SYCL platform that the context is initialized for.
`vector_class<device> get_devices()const`	Returns the set of devices that are part of this context.

Table 3.8: Methods of context class

3.5.2 Context information descriptors

A SYCL context can be queried for all of the following information using the `get_info` function. All SYCL contexts have valid devices for them, including the SYCL host device. The available information is in table 3.9. The interface of all available context descriptors in the appendix C.2.

Context Descriptors	Return type	Description
`info::context::reference_count`	`cl_uint`	Return the context reference count.
`info::context::num_devices`	`cl_uint`	Return the number of devices in context.
`info::context::devices`	`vector_class<cl_device_id>`	Return the list of devices in context.
`info::context::gl_interop`	`info::gl_context_interop`	Boolean value which specifies whether the context is used for OpenCL/OpenGL interoperability according to the OpenCL 1.2 or OpenCL 2.2 extensions specification document [2].
		End of table

Table 3.9: Context information descriptors

On construction of a context, it is possible to supply an asynchronous error handler function object. If supplied, then asynchronous errors can be reported to the error handler. Asynchronous errors are only reported to the user when a queue attached to the context is destroyed or has its `wait_and_throw()` method called.

3.6 Device class

The SYCL `device` class encapsulates a particular SYCL device on which kernels may be executed. The SYCL device may be an OpenCL device or it may be a SYCL host device representing the host CPU. In the OpenCL device case, it should have a valid `cl_device_id` and `cl_platform_id` available. The `cl_device_id` for the SYCL host device is not going to be available through the OpenCL interface, as it is not an OpenCL device. In the case where the SYCL device is constructed from an existing `cl_device_id` the system will call `clRetainDevice`. On destruction, the runtime will call `clReleaseDevice`. It is the user's responsibility to make sure that the device object with `cl_device_id` is a valid object during the lifetime of the `device` class.

3.6.1 Device interface

The constructors and methods of the `device` class are listed in Tables 3.10 and 3.11.

```
namespace cl {
namespace sycl {
class device {
 public:
  device();

  explicit device(cl_device_id deviceId);

  explicit device(const device_selector &deviceSelector);

  device(const device &rhs);

  device &operator=(const device &rhs);

  ~device();

  // The OpenCL cl_platform_id or nullptr for SYCL host.
  cl_device_id get() const;

  bool is_host() const;

  bool is_cpu() const;

  bool is_gpu() const;

  bool is_accelerator() const;

  platform get_platform() const;

  // Returns all the available OpenCL devices and the SYCL host device
  static vector_class<device> get_devices(
      info::device_type deviceType = info::device_type::all);

  template <info::device param>
  typename info::param_traits<info::device, param>::type
  get_info<info::device>() const;

  bool has_extension(const string_class &extension) const;

  vector_class<device> create_sub_devices(
                              info::device_partition_type partitionType,
                              info::device_partition_property partitionProperty,
                              info::device_affinity_domain affinityDomain) const;


};
}  // namespace sycl
}  // namespace cl
```

The default constructor will create an instance of the SYCL host device.

The developer can partition existing devices through the `create_sub_devices` API. More documentation on this is in the OpenCL 2.2 specification [3, sec. 4.3]. It is valid to construct a SYCL device directly from an OpenCL sub-device.

Information about the SYCL device may be queried through the `get_info` method. The developer can also query the device instance for the `cl_device_id` which will be nullptr if the SYCL device is the host device. The `get_platform` method will return the corresponding `platform` object.

To facilitate the different options for SYCL devices, there are methods that check the type of device. The method `is_host()` returns true if the device is actually the host. In the case where an OpenCL device has been initialized through this API, the methods `is_cpu()` , `is_gpu()` and `is_accelerator()` return true if the OpenCL device is CPU, GPU or an accelerator.

Constructors	**Description**
`device ()`	Default constructor for the device. It chooses a device using *host_selector*. Returns errors via C++ exception class.
`explicit device (const device_selector & deviceSelector)`	Constructs a device class instance using the *device_selector* provided. Returns errors via C++ exceptionss.
`explicit device (cl_device_id deviceId)`	Constructs a device class instance using *cl_device_id* of the OpenCL device. Returns synchronous errors via the SYCL exception class. Retains a reference to the OpenCL device and if this device was an OpenCL sub-device the device should be released by the caller when it is no longer needed.
`device (const device &rhs)`	Copy constructor. Returns synchronous errors via the SYCL exception class.
`device &operator=(const device &rhs)`	Assignment constructor. Returns synchronous errors via the SYCL exception class.
	End of table

Table 3.10: Constructors of the `device` class

Methods	Description
`cl_device_id get ()const`	Returns the *cl_device_id* of the underlying OpenCL platform. Returns synchronous errors via the SYCL exception class. Retains a reference to the returned `cl_device_id` object. Caller should release it when finished. In the case where this is the SYCL host device, it will trigger an *invalid_object_error* exception of type *device_error*.
`platform get_platform ()const`	Returns the platform of the device. Returns synchronous errors via the SYCL exception class.
`bool is_host ()const`	Returns true if the device is a SYCL host device.
`bool is_cpu ()const`	Returns true if the device is an OpenCL CPU device.
`bool is_gpu ()const`	Returns true if the device is an OpenCL GPU device.
`bool is_accelerator ()const`	Returns true if the device is an OpenCL accelerator device.
`template <info::device param> typename info::param_traits <info::device, param>::type get_info ()const`	Queries the device for OpenCL *info::device_info*. Returns synchronous errors via the SYCL exception class.
`bool has_extension (const string_class &extension) const`	Specifies whether a specific extension is supported on the device.
`vector_class<device> create_sub_devices ( info::device_partition_type partitionType, info::device_partition_property partitionProperty, info::device_affinity_domain affinityDomain) const`	Partitions the device into sub devices based upon the properties provided. Returns synchronous errors via SYCL exception classes.
`static vector_class<device> get_devices ( info::device_type deviceType = info::device_type::all)`	Returns a list of all available devices. Returns synchronous errors via SYCL exception classes.
	End of table

Table 3.11: Methods of the `device` class

3.6.2 Device information descriptors

A SYCL device can be queried for all of the following information using the `get_info` function. All SYCL devices have valid queries for them, including the SYCL host device. The available information is in table 3.12. The interface of all available device descriptors in the appendix C.3.

Device Descriptors	Return type	Description
`info::device::type`	`info::device_type`	The SYCL device type. Currently supported values are: cpu, gpu, accelerator, defaults, custom, host, all.
`info::device::vendor_id`	`cl_uint`	A unique SYCL device vendor identifier. An example of a unique device identifier could be the PCIe ID. The SYCL host device has to report a valid vendor id.
`info::device::max_compute_units`	`cl_uint`	The number of parallel compute units on the SYCL device. A work-group executes on a single compute unit. The minimum value is 1.
`info::device::max_work_item_dimensions`	`cl_uint`	Maximum dimensions that specify the global and local work-item IDs used by the data parallel execution model. The minimum value is 3 for devices that are not of type `info::device_type::custom`.
`info::device::max_work_item_sizes`	`id<3>`	Maximum number of work-items that can be specified in each dimension of the work-group to the nd_range. The minimum value is (1, 1, 1) for devices that are not of type `info::device_type::custom`.
`info::device::max_work_group_size`	`size_t`	Maximum number of work-items in a work-group executing a kernel on a single compute unit, using the data parallel execution model. The minimum value is 1.
`info::device::preferred_vector_width_char` `info::device::preferred_vector_width_short` `info::device::preferred_vector_width_int` `info::device::preferred_vector_width_long` `info::device::preferred_vector_width_float` `info::device::preferred_vector_width_double` `info::device::preferred_vector_width_half`	`cl_uint`	Preferred native vector width size for builtin scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. If double precision is not supported, `info::device_preferred_width_double` must return 0. If the `cl_khr_fp16` extension is not supported, `info::device::preferred_vector_width_half` must return 0.

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
info::device:: native_vector_width_char info::device:: native_vector_width_short info::device:: native_vector_width_int info::device:: native_vector_width_long info::device:: native_vector_width_float info::device:: native_vector_width_double info::device:: native_vector_width_half	cl_uint	Returns the native ISA vector width. The vector width is defined as the number of scalar elements that can be stored in the vector. If double precision is not supported, info ::device::native_vector_width_double must return 0. If the cl_khr_fp16 extension is not supported, info::device:: native_vector_width_half must return 0.
info::device:: max_clock_frequency	cl_uint	Maximum configured clock frequency of the device in MHz.
info::device::address_bits	cl_uint	The default compute device address space size specified as an unsigned integer value in bits. Currently supported values are 32 or 64 bits.
info::device:: max_mem_alloc_size	cl_long	Max size of memory object allocation in bytes. The minimum value is max (1/4th of info::device::global_mem_size ,128*1024*1024) for devices that are not of type info::device_type::custom.
info::device::IL_version	string_class	The intermediate languages that can be supported for this device. Returns a space-separated list of IL version strings of the form ¡IL_Prefix¿_¡Major_Version¿.¡Minor_Version¿. For OpenCL 2.2, SPIR-V is a required IL prefix.
info::device::image_support	cl_bool	Is 1 if images are supported by the SYCL device and 0 otherwise.
info::device:: max_read_image_args	cl_uint	Max number of simultaneous image objects that can be read by a kernel. The minimum value is 128 if info::device::image_support is true.
info::device:: max_write_image_args	cl_uint	Max number of simultaneous image objects that can be written to by a kernel. The minimum value is 64 if info::device::image_support is true.
info::device:: max_read_write_image_args	cl_uint	Max number image objects as arguments to a kernel with write or read_write flags. The minimum value is 64 if info::device:: image_support is true.
info::device:: image2d_max_width	size_t	Max width of 2D image or 1D image, not created from a buffer object, in pixels. The minimum value is 16384 if info::device:: image_support is true.

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
info::device::image_2d_max_height	size_t	Max height of 2D image in pixels. The minimum value is 16384 if info::device::image_support is true.
info::device::image3d_max_width	size_t	Max width of 3D image in pixels. The minimum value is 2048 if info::device::image_support is true.
info::device::image3d_max_height	size_t	Max height of 3D image in pixels. The minimum value is 2048 if info::device::image_support is true.
info::device::image3d_max_depth	size_t	Max depth of 3D image in pixels. The minimum value is 2048 if info::device::image_support is true.
info::device::image_max_buffer_size	size_t	Max number of pixels for a 1D image created from a buffer object. The minimum value is 65536 if info::device::image_support is true.
info::device::image_max_array_size	size_t	Max number of images in a 1D or 2D image array. The minimum value is 2048 if info::device::image_support is true.
info::device::max_samplers	cl_uint	Maximum number of samplers that can be used in a kernel. The minimum value is 16 info::device::image_support is true.
info::device::image_pitch_alignment	cl_uint	The row pitch alignment size in pixels for 2D images created from a buffer. The value returned must be a power of 2. If the device does not support images, this value must be 0.
info::device::max_pipe_args	cl_uint	The maximum number of pipe objects that can be passed as arguments to a kernel. The minimum value is 16.
info:device::pipe_max_active_reservations	cl_uint	The maximum number of reservations that can be active for a pipe per work-item in a kernel. A work-group reservation is counted as one reservation per work-item. The minimum value is 1.
info::device::pipe_max_packet_size	cl_uint	The maximum size of pipe packet in bytes. The minimum value is 1024 bytes.
info::device::max_parameter_size	size_t	Max size in bytes of the arguments that can be passed to a kernel. The minimum value is 1024 for devices that are not of type info::device_type::custom. For this minimum value, only a maximum of 128 arguments can be passed to a kernel.
info::device::mem_base_addr_align	cl_uint	The minimum value is the size (in bits) of the largest SYCL built-in data type supported by the device is longlong16 for devices that are not of type info::device_type::custom.

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
`info::device::single_fp_config`	`info:: device_fp_config`	Describes single precision floating-point capability of the device. This is a bit-field that describes one or more of the following values: • info::fp_config::denorm : denorms are supported • info::fp_config::inf_nan : INF and quiet NaNs are supported. • info::fp_config::round_to_nearest: round to nearest even rounding mode supported • info::fp_config::round_to_zero : round to zero rounding mode supported • info::fp_config::round_to_inf : round to positive and negative infinity rounding modes supported • info::fp_config::fma : IEEE754-2008 fused multiply add is supported. • info::fp_config::correctly_rounded_divide_sqrt : divide and sqrt are correctly rounded as defined by the IEEE754 specification. • info::fp_config::soft_float : Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. The mandated minimum floating-point capability for devices that are not of type info::device_type::custom is: info::fp_config::round_to_nearest info::fp_config::inf_nan

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
info::device::double_fp_config	info::device_fp_config	Describes double precision floating-point capability of the SYCL device. This is a bit-field that describes one or more of the following values: • info::fp_config::denorm: denorms are supported • info::fp_config::inf_nan : INF and NaNs are supported. • info::fp_config::round_to_nearest : round to nearest even rounding mode supported. • info::fp_config::round_to_zero : round to zero rounding mode supported. • info::fp_config::round_to_inf : round to positive and negative infinity rounding modes supported. • info::fp_config::fma : IEEE754-2008 fused multiply-add is supported. • info::fp_config::soft_float : Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. Double precision is an optional feature so the mandated minimum double precision floating-point capability is 0. If double precision is supported by the device, then the minimum double precision floating-point capability must be: info::fp_config::fma | info::fp_config::round_to_nearest | info::fp_config::round_to_zero | info::fp_config::round_to_inf | info::fp_config::inf_nan | info::fp_config::denorm.
info::device::global_mem_cache_type	info::device_mem_cache_type	Type of global memory cache supported. Valid values are: none, read_only_cache, write_only_cache.
info::device::global_mem_cache_line_size	cl_uint	Size of global memory cache line in bytes.
info::device::global_mem_cache_size	cl_ulong	Size of global memory cache in bytes.
info::device::global_mem_size	cl_ulong	Size of global device memory in bytes.
info::device::max_constant_buffer_size	cl_ulong	Max size in bytes of a constant buffer allocation. The minimum value is 64 KB for devices that are not of type info::device_type::custom.
info::device::max_constant_args	cl_uint	Max number of constant arguments declared in a kernel. The minimum value is 8 for devices that are not of type info::device_type::custom.

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
`info::device:: max_global_variable_size`	`size_t`	The maximum number of bytes of storage that may be allocated for any single variable in program scope or inside a function declared in the global address space. The minimum value is 64 KB.
`info::device:: global_variable_preferred_total_size`	`size_t`	Maximum preferred total size, in bytes, of all program variables in the global address space. This is a performance hint. An implementation may place such variables in storage with optimized device access. This query returns the capacity of such storage. The minimum value is 0.
`info::device::local_mem_type`	`info:: local_mem_type`	Type of local memory supported. This can be set to info::local_mem_type::local implying dedicated local memory storage such as SRAM, or info::local_mem_type::global. For custom devices, info::local_mem_type::none can also be returned indicating no local memory support.
`info::device::local_mem_size`	`cl_ulong`	Size of local memory arena in bytes. The minimum value is 32 KB for devices that are not of type info::device_type::custom.
`info::device:: error_correction_support`	`cl_bool`	Is true if the device implements error correction for all accesses to compute device memory (global and constant). Is false if the device does not implement such error correction.
`info::device:: profiling_timer_resolution`	`size_t`	Describes the resolution of device timer. This is measured in nanoseconds.
`info::device::is_endian_little`	`cl_bool`	Is true if the SYCL device is a little endian device and false otherwise.
`info::device::is_available`	`cl_bool`	Is true if the device is available and false if the device is not available.
`info::device:: is_compiler_available`	`cl_bool`	Is false if the implementation does not have a compiler available to compile the program source. An OpenCL device that conforms to the OpenCL Embedded Profile may not have an online compiler available.
`info::device:: is_linker_available`	`cl_bool`	Is false if the implementation does not have a linker available. An OpenCL device that conforms to the OpenCL Embedded Profile may not have a linker available. However, it needs to be true if info::device::is_compiler_available is true.

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
`info::device::execution_capabilities`	`info::device_exec_capabilities`	Describes the execution capabilities of the device. This is a bit-field that describes one or more of the following values: • info::device_execution_capabilities::exec_kernel : The OpenCL device can execute OpenCL kernels. • info::device_execution_capabilities::exec_native_kernel : The OpenCL device can execute native kernels. The mandated minimum capability is: info::device_execution_capabilities::exec_kernel.
`info::device::opencl_queue_out_of_order_exec`	`cl_bool`	Check whether the opencl queue on host is executing out-of-order. Returns true if out-of-order is enabled.
`info::device::queue_profiling_enabled`	`cl_bool`	Check whether the opencl queue has profiling info enabled.
`info::device::device_queue_out_of_order_exec`	`cl_bool`	Check whether the `device_queue` is execution in an out of order mode. Returns true if out-of-order is enabled.
`info::device::device_queue_profiling_enabled`	`cl_bool`	Check whether the device_queue has profiling info enabled.
`info::device::device_queue_preferred_size`	`cl_uint`	The size of the device queue in bytes preferred by the implementation. Applications should use this size for the device queue to ensure good performance. The minimum value is 16 KB
`info::device::device_queue_max_size`	`cl_uint`	The max. size of the device queue in bytes. The minimum value is 256 KB for the full profile and 64 KB for the embedded profile
`info::device::max_device_queues`	`cl_uint`	Maximum number of device queues that can be created per context. The minimum value is 1.
`info::device::max_device_events`	`cl_uint`	The maximum number of events in use by a device_queue. These refer to events created when submitting command_groups on a device_queue and they haven't been released yet. The minimum value is 1024.
`info::device::built_in_kernels`	`string_class`	A semi-colon separated list of built-in kernels supported by the device. An empty string is returned if no built-in kernels are supported by the device.
`info::device::platform`	`cl_platform_id`	The platform associated with this device.
`info::device::name`	`string_class`	Device name string
`info::device::vendor`	`string_class`	Vendor name string.

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
`info::device::driver_version`	`string_class`	OpenCL software driver version string in the form major_number.minor_number
`info::device::profile`	`string_class`	OpenCL profile string. Returns the profile name supported by the device. The profile name returned can be one of the following strings: • FULL_PROFILE : if the device supports the OpenCL specification (functionality defined as part of the core specification and does not require any extensions to be supported). • EMBEDDED_PROFILE - if the device supports the OpenCL embedded profile.
`info::device::version`	`string_class`	OpenCL version string. Returns the OpenCL version supported by the device. This version string has the following format: `OpenCL<space><major_version.minor_version><space><vendor-specific-information>` The `major_version.minor_version` value returned will be 1.2.
`info::device::opencl_c_version`	`string_class`	OpenCL C version string. Returns the highest OpenCL C version supported by the compiler for this device that is not of type info::device_type::custom. This version string has the following format: `OpenCL<space>C<space><major_version.minor_version><space><vendor-specific-information>` The `major_version.minor_version` value returned must be 1.2 if info::device::version is OpenCL 1.2. The `major_version.minor_version` value returned must be 1.1 if info::device::version is OpenCL 1.1. The `major_version.minor_version` value returned can be 1.0 or 1.1 if info::device::version is OpenCL 1.0.
		Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
info::device::extensions	vector< string_class>	Returns a space separated list of extension names (the extension names themselves do not contain any spaces) supported by the device. The list of extension names returned can be vendor supported extension names and one or more of the following Khronos approved extension names: • cl_khr_int64_base_atomics • cl_khr_int64_extended_atomics • cl_khr_3d_image_writes • cl_khr_fp16 • cl_khr_gl_sharing • cl_khr_gl_event • cl_khr_d3d10_sharing • cl_khr_dx9_media_sharing • cl_khr_d3d11_sharing • cl_khr_depth_images • cl_khr_gl_depth_images • cl_khr_gl_msaa_sharing • cl_khr_initialize_memory • cl_khr_terminate_context • cl_khr_spir • cl_khr_srgb_image_writes The following approved Khronos extension names must be returned by all device that support OpenCL C 2.0: • cl_khr_byte_addressable_store • cl_khr_fp64 (for backward compatibility if double precision is supported) • cl_khr_3d_image_writes • cl_khr_image2d_from_buffer • ccl_khr_depth_images Please refer to the OpenCL 2.0 Extension Specification for a detailed description of these extensions.
info::device:: printf_buffer_size	size_t	Maximum size of the internal buffer that holds the output of printf calls from a kernel. The minimum value for the full profile is 1 MB.
info::device:: preferred_interop_user_sync	cl_bool	Is true if the devices preference is for the user to be responsible for synchronization, when sharing memory objects between OpenCL and other APIs such as DirectX, false if the device/implementation has a performant path for performing synchronization of memory object shared between OpenCL and other APIs such as DirectX.

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
info::device::parent_device	cl_device_id	Returns the cl_device_id of the parent device to which this sub-device belongs. If device is a root-level device, or the host device a NULL value is returned.
info::device:: partition_max_sub_devices	cl_uint	Returns the maximum number of subdevices that can be created when a device is partitioned. The value returned cannot exceed info ::device::device_max_compute_units
info::device:: partition_properties	vector_class < info:: device_partition_ property >	Returns the list of partition types supported by device. The is an array of info ::device_partition_property values drawn from the following list: • info::device_partition_property::partition_equally • info::device_partition_property::partition_by_counts • info::device_partition_by_affinity_domain If the device cannot be partitioned (i.e. there is no partitioning scheme supported by the device that will return at least two subdevices), a value of 0 will be returned.
info::device:: partition_affinity_domain	info:: device_affinity_ domain	Returns the list of supported affinity domains for partitioning the device using info::device_-affinity_domain. This is a bit-field that describes one or more of the following values: • info::device_affinity_domain::unsupported • info::device_affinity_domain::numa • info::device_affinity_domain::L4_cache • info::device_affinity_domain::L3_cache • info::device_affinity_domain::L2_cache • info::device_affinity_domain::L1_cache • info::device_affinity_domain::next_partitionable

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
`info::device::partition_type`	`vector< info:: device_partition _property >`	Returns the properties argument specified when creating sub devices, if device is a subdevice. In the case where the properties argument to creating sub devices is info::device_partition_property::partition_by_affinity_domain info::device_partition_property::partition_affinity_domain_next_partitionable, the affinity domain used to perform the partition will be returned. This can be one of the following values: • info::device_affinity_domain::unsupported • info::device_affinity_domain::numa • info::device_affinity_domain::L4_cache • info::device_affinity_domain::L3_cache • info::device_affinity_domain::L2_cache • info::device_affinity_domain::L1_cache Otherwise the implementation may either return a `param_value_size_ret` of 0 i.e. there is no partition type associated with device or can return a property value of 0 (where 0 is used to terminate the partition property list) in the memory that param_value points to.
`info::device::reference_count`	`cl_uint`	Returns the device reference count. If the device is a root-level device, a reference count of one is returned.
`info::device::svm_capabilities`	`svm:: capabilities`	Describes the various shared virtual memory (a.k.a. SVM) memory allocation types the device supports. Coarse-grain SVM allocations are required to be supported by all OpenCL 2.2 devices. This is an enum class that supports the following: • svm_coarse_grain_buffer • svm_coarse_grain_buffer_with_atomics • svm_fine_grain_buffer • svm_fine_grain_buffer_with_atomics • svm_fine_grain_system • svm_fine_grain_system_with_atomics
`info::device:: preferred_platform_atomic_alignment`	`cl_uint`	Returns the value representing the preferred alignment in bytes for OpenCL 2.2 fine-grained SVM atomic types. This query can return 0 which indicates that the preferred alignment is aligned to the natural size of the type.

Continued on next page

Table 3.12: Device information descriptors.

Device Descriptors	Return type	Description
`info::device::preferred_global_atomic_alignment`	`cl_uint`	Returns the value representing the preferred alignment in bytes for OpenCL 2.2 atomic types to global memory. This query can return 0 which indicates that the preferred alignment is aligned to the natural size of the type.
`info::device::preferred_local_atomic_alignment`	`cl_uint`	Returns the value representing the preferred alignment in bytes for OpenCL 2.2 atomic types to local memory. This query can return 0 which indicates that the preferred alignment is aligned to the natural size of the type.
`info::device::max_num_sub_groups`	`cl_uint`	Maximum number of sub-groups in a work-group that a device is capable of executing on a single compute unit, for any given kernel-instance running on the device. The minimum value is 1.
`info::device::sub_group_independent_forward_progress`	`cl_bool`	Is true if this device supports independent forward progress of sub-groups, false otherwise. If `cl_khr_subgroups` is supported by the device this must return true.
		End of table

Table 3.12: Device information descriptors.

3.7 Queue class

The class `queue` provided in SYCL is a scheduling queue for a device which provides the functionality of scheduling kernels on host or on an OpenCL device. A `device_queue` is an OpenCL default device queue for enqueuing work on a device without needing to schedule it on host. This functionality is available through usage of *nested parallelism* available to OpenCL 2.x devices. The SYCL `queue` is a host queue and is available only on host, and the `device_queue` is a device queue and only available on a SYCL device. The latter is part of the SYCL kernel library and not of the SYCL runtime library. Further description of the `device_queue` is in 5.5, 7.3.

The destructor of the SYCL `queue` waits for all execution on the queue and its device queues to end and then passes any exceptions that occurred asynchronously on the queue to the asynchronous error handler `async_handler`.

3.7.1 queue interface

The constructors and methods of the `queue` class are listed in Tables 3.13 and 3.14.

```
namespace cl {
namespace sycl {
class queue {
 public:
  explicit queue(async_handler asyncHandler = nullptr);

  queue(const device_selector &deviceSelector,
        async_handler asyncHandler = nullptr);

  queue(const context &syclContext, const device_selector &deviceSelector,
        async_handler asyncHandler = nullptr);

  queue(const context &syclContext, const device &syclDevice,
        async_handler asyncHandler = nullptr);

  queue(const context &syclContext, const device &syclDevice,
        info::queue_profiling profilingFlag, async_handler asyncHandler = nullptr);

  queue(const device &syclDevice, async_handler asyncHandler = nullptr);

  queue(const cl_command_queue &clQueue, async_handler asyncHandler = nullptr);

  queue(const queue &syclQueue);

  ~queue();

  cl_command_queue get() const;

  context get_context() const;

  device get_device() const;

  bool is_host() const;

  template <info::queue param>
  typename info::param_traits<info::queue, param>::type get_info() const;

  template <typename T>
  handler_event submit(T cgf);

  template <typename T>
  handler_event submit(T cgf, const queue &secondaryQueue);

  void wait();

  void wait_and_throw();

  void throw_asynchronous();
};
}  // namespace sycl
}  // namespace cl
```

Constructors	Description

`explicit queue (`
`    async_handler a_handler = )`
	Creates a SYCL queue using the default `device_selector` 3.3.
	The heuristics regarding the choice for platform and device for the default selector are implementation defined. All runtime objects necessary for the search and creation of the queue are default constructed.
	A SYCL queue corresponds to one or more OpenCL queues and behaves as an out-of-order queue. An OpenCL command queue can be retrieved and the SYCL application can interact with it with the guarantee that all internal synchronization is handled by the SYCL runtime.
	Returns synchronous errors regarding the creation of the queue and reports asynchronous errors via the *async_handler* 5.11.2 callback function class instance (if provided) in conjunction with the synchronization and throw methods.

`queue (const device_selector &selector,`
`    async_handler a_handler = )`
	Creates a queue for the device chosen by the `device_selector` 3.3 provided. If no device is selected, an error is reported. All runtime objects necessary for the search and creation of the queue are using the rules provided by `selector` and the rest are default constructed.
	A SYCL queue corresponds to one or more OpenCL queues. An OpenCL command queue can be retrieved and the SYCL application can interact with it with the guarantee that all internal synchronization is handled by the SYCL runtime.
	Returns synchronous errors regarding the creation of the queue and reports asynchronous errors via the *async_handler* 5.11.2 callback function class instance if and only if there is an `async_handler` provided and throw methods are used.

Continued on next page

Table 3.13: Constructors of the `queue` class.

Constructors	Description
`queue (const device &syclDevice, async_handler a_handler = )`	A queue is created for the SYCL *device* syclDevice. All runtime objects necessary for the creation of the queue are constructed given only the *device* 3.6 object. A SYCL queue corresponds to one or more OpenCL queues. An OpenCL command queue can be retrieved and the SYCL application can interact with it with the guarantee that all internal synchronization is handled by the SYCL runtime. Returns synchronous errors regarding the creation of the queue and reports asynchronous errors via the *async_handler* 5.11.2 callback function class instance if and only if there is an `async_handler` provided and throw methods are used.
`queue (const context & s_context, const device_selector &d_selector)`	This constructor chooses a device based on the provided `device_selector` 3.3, which needs to be in the SYCL context 3.5 s_context. If no device is selected or the s_context is not valid for that device, an error is reported. A SYCL queue corresponds to one or more OpenCL queues. An OpenCL command queue can be retrieved and the SYCL application can interact with it with the guarantee that all internal synchronization is handled by the SYCL runtime. Returns synchronous errors regarding the creation of the queue. If and only if there is an *async_handler* defined for the `s_context` object and wait methods used, it reports asynchronous errors via the *async_handler* 5.11.2 callback function class instance of the *context* 3.5.
`queue (const context &s_context, const device &d_selector)`	Creates a SYCL queue from the SYCL context 3.5 and device 3.5 provided. Returns synchronous errors regarding the creation of the queue. If and only if there is an *async_handler* defined for the `s_context` object and wait methods used, it reports asynchronous errors via the *async_handler* 5.11.2 callback function class instance of the *context* 3.5.

Continued on next page

Table 3.13: Constructors of the `queue` class.

Constructors	Description
`queue (const context &s_context, const device &syclDevice, info::queue_profiling flag)`	Creates a SYCL queue from the SYCL context 3.5 and device 3.5 given. It enables profiling on the queue if the `flag` is set to true. Returns synchronous errors regarding the creation of the queue. If and only if there is an *async_handler* defined for the `s_context` object and wait methods used, it reports asynchronous errors via the *async_handler* 5.11.2 callback function class instance of the *context* 3.5.
`queue (cl_command_queue clqueue, async_handler a_handler = )`	Creates a SYCL queue that corresponds to the OpenCL command queue provided, using the device and context information of the OpenCL queue. In this case, the SYCL queue is only going to have one corresponding underlying object and mirror the capabilities of the underlying OpenCL queue. At construction it does a retain on the queue memory object. Returns synchronous errors regarding the creation of the queue. If and only if there is an `a_handler` provided and throw methods are used, it reports asynchronous errors via the *async_handler* 5.11.2 callback function class instance.
`queue (queue &syclQueue)`	Copy constructor
	End of table

Table 3.13: Constructors of the queue class.

Methods	Description
`cl_command_queue get()`	Returns an OpenCL command queue for synchronization purposes after doing a retain. This memory object is expected to be released by the developer. Retains a reference to the returned `cl_command_queue` object. Caller should release it when finished. If the queue is a SYCL host queue then a synchronous *invalid_object_error* exception will be triggered of type *device_error*.
`context get_context ()const`	Returns the SYCL queue's context. Reports errors using SYCL exception classes.
`device get_device ()const`	Returns the SYCL device the queue is associated with. Reports errors using SYCL exception classes.
	Continued on next page

Table 3.14: Methods for class queue

Methods	Description
`bool is_host ()const`	Returns whether the queue is executing on a SYCL host device.
`void wait()`	Performs a blocking wait for the completion all enqueued tasks in the queue. Synchronous errors will be reported through SYCL exceptions.
`void wait_and_throw ()`	Performs a blocking wait for the completion all enqueued tasks in the queue. Synchronous errors will be reported via SYCL exceptions. Asynchronous errors will be passed to the `async_handler` passed to the queue on construction. If no `async_handler` was provided then asynchronous exceptions will be lost.
`void throw_asynchronous ()`	Checks to see if any asynchronous errors have been produced by the queue and if so reports them by passing them to the `async_handler` passed to the queue on construction. If no `async_handler` was provided then asynchronous exceptions will be lost.

`template<info::queue param>`
`    typename info::param_traits`
`    <info::queue, param>::type`
`    get_info ()const`
	Queries the platform for *cl_command_queue_info*

`template <typename T>`
`    handler_event submit(T cgf)`
	Submit a command group functor to the queue, in order to be scheduled for execution on the device.

`template <typename T>`
`    handler_event submit(`
`    T cgf, queue & secondaryQueue)`
	Submit a command group functor to the queue, in order to be scheduled for execution on the device. On kernel error, this command group functor, then it is scheduled for execution on the secondary queue. Returns a command group functor event, which is corresponds to the queue the command group functor is being enqueued on.

`device_queue get_default_device_queue()`	The default device_queue which is created by the OpenCL runtime is retrieved in order to enable scheduling work on the device, without host intervention. All of the errors for any subsequent kernels or events scheduled using device queues based on the host queue are returned using the async_handler provided in the constructor. If no `async_handler` was provided then asynchronous exceptions will be lost.
	End of table

Table 3.14: Methods for class queue

3.7.2 Queue information descriptors

A SYCL queue can be queried for all of the following information using the `get_info` function. All SYCL queues have valid queries for them, including the SYCL host queue. The available information is in table 3.15. The interface of all available device descriptors in the appendix C.4.

Queue Descriptors	Return type	Description
`info::queue::context`	`cl_context`	Return the context specified when the command-queue is created.
`info::queue::device`	`cl_device_id`	Return the device specified when the command-queue is created.
`info::queue::reference_count`	`cl_uint`	Return the command-queue reference count.
`info::queue::properties`	`info:: queue_profiling`	Return the currently specified properties for the command-queue. These properties are specified by the properties argument in the queue constructor.
		End of table

Table 3.15: Queue information descriptors

3.7.2.1 Queue error handling

Queue errors come in two forms:

- **Synchronous Errors** are those that are reported directly at the point of waiting on an event or waiting for a queue to complete, as well as any immediate errors reported by enqueuing work onto a queue. Such errors are returned via C++ exceptions.

- **Asynchronous errors** are those that are produced via callback functions only. These will be stored within the queue's context until they are dispatched to the context's asynchronous error handler. If a queue is constructed with a user-supplied context, then it is this context's asynchronous error handler to which asynchronous errors are reported. If a queue is constructed without a user-supplied context, then the queue's constructor can be supplied with a queue-specific asynchronous error handler which will be used to construct the queue's context. To ensure that such errors are processed predictably in a known host thread these errors are only passed to the asynchronous error handler on request when either `wait_and_throw` is called or when `throw_asynchronous` is called. If no asynchronous error handler is passed to the queue or its context on construction, then such errors go unhandled, much as they would if no callback were passed to an OpenCL context.

4. Data access and storage in SYCL

In SYCL, data storage and access are handled by separate classes. *Buffers*, *SVM allocations* and *images* handle storage and ownership of the data, whereas *accessors* handle access to the data. Buffers and images in SYCL are different to OpenCL buffers and images in that they can be bound to more than one device or context and they get destroyed when they go out-of-scope. They also handle ownership of the data, while allowing exception handling for blocking and non-blocking data transfers. Accessors manage data transfers between host and all the devices in the system, as well as tracking data dependencies.

In SYCL 2.2, shared virtual memory between the host and device is exposed in all the different modes that are core or optional features of OpenCL systems. The core functionality of shared virtual memory is to allow the underlying pointers which are managed by the SYCL runtime to be the same on host and device. The synchronisation rules for the core feature, which is coarse grained buffers, are very similar to the ones for non-svm buffers, where the host and the device need to request access to the pointer. In this case, pointer structs can be allowed for buffers and they will work for host and device buffers. The optional cases, where shared virtual memory synchronisation is mainly done through atomics and not from enqueuing reads and writes, do not use the `buffer` and `accessor` classes, as their allocation and synchronisation rules are different. The description of the different shared virtual memory modes is in chapter 6. `svm_allocator` is the C++ allocator, used for shared allocations and is compatible with buffer and accessor classes, is described in full in chapter 6.

4.1 Host allocation

A SYCL runtime may need to allocate temporary objects on the host to handle some operations (such as copying data from one context to the other). Allocation on the host is managed using an allocator object, following the standard C++ allocator class definition. The default allocator for memory objects is implementation defined, but the user can supply their own allocator class.

```
{
  buffer<int, 1, UserDefinedAllocator<int> > b(d);
}
```

When an allocator returns a `nullptr`, the runtime cannot create data in host memory. Note that in this case the runtime will raise an error if it requires host memory but it is not available (e.g when moving data across OpenCL contexts).

The user can implement an allocator that returns the same address as the one passed in the buffer constructor.

The definition of allocators extends the current functionality of SYCL, ensuring that users can define allocator functions for specific hardware or certain complex shared memory mechanism (e.g. NUMA) and improves interoperability with STL-based libraries.

In the case of shared virtual memory (SVM), allocations on host are also valid on device and the same virtual pointers can be used on both. Depending on the mode of SVM, the allocators that are compatible for the allocations differ. If *System Sharing Virtual Memory* is supported by the OpenCL system, then any C++ allocator can be used on host to allocate shared virtual memory addresses, which can be used in any available device kernel.

4.1.1 Default Allocators

A default allocator is always defined by the implementation, and it is guaranteed to return non `nullptr` and new memory positions every time. The default allocator for `const` buffers will remove the const-ness of the type (therefore, the default allocator for a buffer of type ”`const int`” will be an `Allocator<int>`). This implies that host accessors will not synchronize with the pointer given by the user in the buffer/image constructor, but will use the memory returned by the `Allocator` itself for that purpose. The user can implement an allocator that returns the same address as the one passed in the buffer constructor, but it is the responsibility of the user to handle the potential race conditions.

Allocators	**Description**
`buffer_allocator`	This is the default buffer allocator used by the runtime, when no allocator is defined by the user.
`image_allocator`	This is the default image allocator used by the runtime, when no allocator is defined by the user. The image allocator is required to be a byte-sized allocator, so the default type this allocator is typed to a type of size 1.
`svm_allocator`	It is the only allocator supported by the system for allocating buffer sharing SVM pointers, either for coarse or fine grained shared virtual address space. The pointer allocations for the above two modes need, can only allocate buffers and can be used within a *context*. The structures allocated may include pointer structures, which would be pre-allocated. The *svm_allocator* is compatible with the buffer class, and its default mode is `svm_coarse_grain`. The other alternative is *fine grained buffer sharing SVM* with or without atomics, which can be used with any container class or directly for allocating and deallocating raw pointers. In the case of system sharing SVM, the *svm_allocator* is not necessary, as any C++ allocation on the host is a valid SVM system allocation. For more detailed information, please see 6.
	End of table

Table 4.1: SYCL Default Allocators

See later 4.6 for detail on manual host-device synchronization.

4.1.2 Map Allocator

The `map_allocator` is a class provided by the SYCL interface that can be matched with a specialized constructor of the buffer or image in order to provide the capability of 'mapping' the host data on any devices that the buffer uses.

This allocator always uses the same host address in order to create or map any device buffers, avoiding any copies from host data to host buffer and also uses the same host memory for all host accesses. A side effect of that is that the host accessors will synchronize with the host memory and the runtime will handle that synchronization.

The host data address used for mapping the host data to the device data cannot be `const`, as the `map_allocator` will always use that address even for transferring data across devices of different contexts.

4.2 Buffers

The `buffer` class defines a shared array data of one, two or three dimensions that can be used by kernels in queues and has to be accessed using `accessor` classes. Buffers are templated on both the type of their data, and the number of dimensions the data is stored and accessed through.

A `cl::sycl::buffer` does not map to only one OpenCL buffer object and all OpenCL buffer memory objects are temporary for the use within a command group on a specific device. The only exception to this rule is when a buffer is constructed from a `cl_mem` object to interoperate with OpenCL. In the interop case the buffer will constitute a single `cl_mem` object and the ownership of the buffer `cl_mem` memory object remains at the OpenCL side. The SYCL buffer constructor and destructor use the existing retain and release mechanism available in OpenCL. Use of an interoperability buffer on a queue mapping to a context other than that in which the `cl_mem` was created is an error.

In the case of coarse grained buffer shared virtual memory, SVM buffers can be used. These are only mapped to one virtual memory buffer in one context, the context of the allocation, and are able to contain complex pointer structures allocated on host and used on the context's devices. This is a special case of a device buffer, where the allocation happens on host using a context, and the underlying raw pointers can be used on host and device. The `svm_allocator` class manages all the allocations and by default it allocates coarse grained buffers. The `accessor` class requests access on host and device, managing all the dependencies and guaranteed data consistency across host and device. More details of this mode can be found in 6.

4.2.1 Buffer Interface

Buffer constructors are listed in Table 4.2 and methods in Table 4.3.

```
namespace cl {
namespace sycl {
template <typename T, int dimensions,
          typename AllocatorT = cl::sycl::buffer_allocator>
```

```
class buffer {
 public:
  using value_type = T;
  using reference = value_type &;
  using const_reference = const value_type &;

  buffer(const range<dimensions> &bufferRange, AllocatorT allocator = nullptr);

  buffer(const T *hostData, const range<dimensions> &bufferRange,
         AllocatorT allocator = nullptr);

  buffer(T *hostData, const range<dimensions> &bufferRange,
         AllocatorT allocator = nullptr);

  buffer(shared_ptr_class<T> &hostData, const range<dimensions> &bufferRange,
         AllocatorT allocator = nullptr);

  buffer(shared_ptr_class<T> &hostData, const range<dimensions> &bufferRange,
         cl::sycl::mutex_class *m, AllocatorT allocator = nullptr);

  buffer(unique_ptr_class<void> &&hostData,
         const range<dimensions> &bufferRange, AllocatorT allocator = nullptr);

  buffer(buffer<T, dimensions, AllocatorT> b, const id<dimensions> &baseIndex,
         const range<dimensions> &subRange, AllocatorT allocator = nullptr);

  template <class InputIterator>
  buffer<T, 1>(InputIterator first, InputIterator last,
               AllocatorT allocator = nullptr);

  buffer(cl_mem memObject, queue &fromQueue, event availableEvent = nullptr);

  buffer(const buffer<T, dimensions, AllocatorT> &rhs);

  buffer<T, dimensions, AllocatorT> &operator=(
      const buffer<T, dimensions, AllocatorT> &rhs);

  ~buffer();

  const range<dimensions> get_range();

  size_t get_count() const;

  size_t get_size() const;

  allocatorT get_allocator() const;

  template <access::mode mode, access::target target = access::global_buffer>
  accessor<T, dimensions, mode, target> get_access(
      handler &command_group_handler);

  template <access::mode mode, access::target target = access::host_buffer>
  accessor<T, dimensions, mode, target> get_access();

  void set_final_data(weak_ptr_class<T> &finalData);
};
} // namespace sycl
} // namespace cl
```

Constructors	Description
`template<typename T, int dimensions, typename AllocatorT = cl::sycl::buffer_allocator> buffer (const range<dimensions> & bufferRange, AllocatorT allocator = nullptr)`	Create a new buffer of the given size with storage managed by the SYCL runtime. The default behavior is to use the default host buffer allocator, in order to allow for host accesses. If the type of the buffer, has the const qualifier, then the default allocator will remove the qualifier to allow host access to the data.
`template<typename T, int dimensions, typename AllocatorT = cl::sycl::buffer_allocator> buffer(const T* hostData, const range<dimensions> & bufferRange, AllocatorT allocator = nullptr)`	Create a new buffer with associated host memory. `hostData` points to the storage and values used by the buffer and `range`<dimensions> defines the size. The host address is const T, so the host accesses can be read-only. However, the `typename` T is not const so the device accesses can be both read and write accesses. Since, the `hostData` is const, this buffer is only initialized with this memory and there is no write after its destruction, unless there is another final data address given after construction of the buffer. The default value of the allocator is going to be the `cl::sycl::buffer_allocator` which will be of type T.
`template<typename T, int dimensions, typename AllocatorT= cl::sycl::buffer_allocator> buffer(T* hostData, const range<dimensions> & bufferRange, AllocatorT allocator = nullptr)`	Create a new buffer with associated host memory. The memory is owned by the runtime during the lifetime of the object. Data is copied back to the host unless the user overrides the behavior using the `set_final_data` method. `hostData` points to the storage and values used by the buffer and `range`<dimensions> defines the size.
`template<typename T, bool Depth, int dimensions, typename AllocatorT= cl::sycl::buffer_allocator> buffer(shared_ptr_class<T>& hostData, const range<dimensions> & bufferRange, cl::sycl::mutex_class * m, AllocatorT allocator = nullptr)`	Create a new buffer with associated memory, using the data in `hostData`. The ownership of the `hostData` is shared between the runtime and the user. In order to enable both the user application and the SYCL runtime to use the same pointer, a `cl::sycl::mutex_class` is used. The mutex m is locked by the runtime whenever the data is in use and unlocked otherwise. Data is synchronized with `hostData`, when the mutex is unlocked by the runtime.

Continued on next page

Table 4.2: Constructors for the `buffer` class.

Constructors	Description
`template<typename T, bool Depth, int dimensions, typename AllocatorT= cl::sycl::buffer_allocator> buffer(unique_ptr_class<void> && hostData, const range<dimensions> & bufferRange, AllocatorT allocator = nullptr)`	Create a new buffer which is initialized by `hostData`. The SYCL runtime receives full ownership of the `hostData` `unique_ptr` and in effect there is no synchronization with the application code using `hostData`.
`template<typename T, int dimensions =1, typename AllocatorT= cl::sycl::buffer_allocator> buffer(Iterator first, Iterator last, AllocatorT allocator = nullptr)`	Create a new allocated 1D buffer initialized from the given elements ranging from `first` up to one before `last`. The data is copied to an intermediate memory position by the runtime. Data is written back to the same iterator set if the iterator is not a const iterator.
`template<typename T, int dimensions, typename AllocatorT= cl::sycl::buffer_allocator> buffer(const buffer<T,dimensions,AllocatorT> & b )`	Create a new buffer copy that shares the data with the original buffer. The system uses reference counting to deal with data lifetime. The destruction of a copy of a buffer does not trigger a copy back from the device.
`template<typename T, int dimensions, typename AllocatorT= cl::sycl::buffer_allocator> buffer( buffer<T, dimensions, AllocatorT> &b, const index<dimensions> & baseIndex, const range<dimensions> & subRange)`	Create a new sub-buffer without allocation to have separate accessors later. `b` is the buffer with the real data. `baseIndex` specifies the origin of the sub-buffer inside the buffer `b`. `subRange` specifies the size of the sub-buffer.
`template<typename T, int dimensions, typename AllocatorT= cl::sycl::buffer_allocator> buffer (cl_mem memObject, queue & fromQueue, event availableEvent = nullptr)`	Create a buffer from an existing OpenCL memory object associated with a context after waiting for an event signaling the availability of the OpenCL data. `memObject` is the OpenCL memory object to use. `fromQueue` is the queue associated to the memory object. `availableEvent` specifies the event to wait for if non-null. Note that a buffer created from a `cl_mem` object will only have one underlying `cl_mem` for the lifetime of the buffer and use on an incompatible queue constitutes an error.
`buffer(const buffer<T,dimensions,AllocatorT> & rhs)`	Copy constructor

Continued on next page

Table 4.2: Constructors for the `buffer` class.

Constructors	Description
`template<typename T, bool Depth, bool MultiSample typename AllocatorT = cl::sycl::svm_allocator< svm_coarse_grain>> buffer<T,1,AllocatorT> (const range<1> & bufferRange , context & bufferContext AllocatorT allocator)`	Create an SVM buffer of size bufferRange, in bufferContext using an instance of `svm_allocator<svm_coarse_grain>`. The buffer allocation uses the *svm_allocator* in order to allocate and deallocate the buffer from the shared virtual address space defined for the given context. The buffer may contain complex pointer structures, which can be used both on host and device, using accessors. The underlying virtual pointers are valid on host and device, however, they need explicit synchronisation rules that need to explicitly make the pointers available on host and on device, in a non-overlapping manner. Synchronisation is managed by buffers and accessor lifetime scopes. This buffer cannot be copied or moved to a differnt context without invalidating any virtual memory pointer allocations. This buffer also cannot be associated with a pre- allocated data structure, as the allocation needs to use the *svm_allocator*. There is no sychronization at buffer destruction on host, as there is no associated host pointer with the allocation. On buffer destruction, the SVM buffer allocation will deallocated using the *svm_allocator*.
	End of table

Table 4.2: Constructors for the `buffer` class.

Methods	Description
`range<dimensions> get_range()const`	Return a range object representing the size of the buffer in terms of number of elements in each dimension as passed to the constructor.
`size_t get_count()const`	Returns the total number of elements in the buffer. Equal to `get_range()[0] * ... * get_range()[dimensions-1]`.
`size_t get_size()const`	Returns the size of the buffer storage in bytes. Equal to `get_count()*sizeof(T)`.
`allocatorT get_allocator()const`	Returns the allocator provided to the buffer.
	Continued on next page

Table 4.3: Methods for the `buffer` class.

Methods	Description
`template<access::mode mode, access::target target= access::global_buffer>` `accessor<T, dimensions, mode, target>` `get_access(handler &command_group_handler)`	Returns a valid accessor to the buffer with the specified access mode and target in the command group buffer. The value of target can be `access::global_buffer`, `access::constant_buffer` or `access::host_buffer`. This accessor may provide access to an SVM buffer, and in this case the underlying pointer structure is guaranteed to have the same virtual address as on the host. In the case of an SVM buffer the command group needs to be enqueued on a queue from the same context as the buffer allocation, otherwise an asynchronous exception will be thrown.
`template<access::mode mode, access::target target= access::host_buffer>` `accessor<T, dimensions, mode, target>` `get_access()`	Returns a valid host accessor to the buffer with the specified access mode and target. The value of target can only be `access::host_buffer`. This accessor may be providing access to an SVM buffer, and in this case the underlying pointer structure is guaranteed to have the same virtual address as on the host. In the case of an SVM buffer, the command group needs to be enqueued on a queue from the same context as the buffer allocation, otherwise an asynchronous exception will be thrown.
`void set_final_data(weak_ptr_class<T> & finalData))`	The `finalData` points to the host memory to which the outcome of all the buffer processing is going to be copied to. This is the final pointer, which is going to be accessible after the destruction of the buffer and in the case where this is a valid pointer, the data will be copied to this host address. `finalData` is different from the original host address, if the buffer was created associated with one. This is mainly to be used when a *shared_ptr* is given in the constructor and the output data will reside in a different location from the initialization data. It is defined as a `weak_ptr` referring to a `shared_ptr` that is not associated with the `cl::sycl::buffer`, and so the `cl::sycl::buffer` will have no ownership of `finalData` .
	End of table

Table 4.3: Methods for the `buffer` class.

4.2.2 Buffer Synchronization Rules

Buffers are reference-counted. When a buffer value is constructed from another buffer, the two values reference the same buffer and a reference count is incremented. When a buffer value is destroyed, the reference count is decremented. Only when there are no more buffer values that reference a specific buffer is the actual buffer destroyed and the buffer destruction behavior defined below is followed.

If any error occurs on buffer destruction, it is reported via the associated queue's asynchronous error handling mechanism

1. A buffer can be constructed with just a size and using the default buffer allocator. The memory management for this type of buffer is entirely handled by the SYCL system. The destructor for this type of buffer never blocks, even if work on the buffer has not completed. Instead, the SYCL system frees any storage required for the buffer asynchronously when it is no longer in use in queues. The initial contents of the buffer are undefined.

2. A buffer can be constructed with associated host memory and a default buffer allocator. The buffer will use this host memory for its full lifetime, but the contents of this host memory are undefined for the lifetime of the buffer. If the host memory is modified by the host, or mapped to another buffer or image during the lifetime of this buffer, then the results are undefined. The initial contents of the buffer will be the contents of the host memory at the time of construction.

   When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed, then copy the contents of the buffer back to the host memory (if required) and then return.

3. If the type of the host data is `const`, then the buffer is read-only; only read accessors are allowed on the buffer and no-copy-back to host memory is performed (although the host memory must still be kept available for use by SYCL). When using the default buffer allocator, the const-ness of the type will be removed in order to allow host allocation of memory, which will allow temporary host copies of the data by the SYCL runtime, for example for speeding up host accesses.

   When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed and then return, as there is no copy of data back to host.

4. If the type of the host data is not *const* but the pointer to host data is *const*, then the read-only restriction applies only on host and not on device accessed.

   When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed and then return, as there is no copy of data back to host.

   However, if the `set_final_data()` function is used and host data pointer is valid for copying data back, then the buffer will block on destruction and copy data using the `weak_ptr` provided.

5. A buffer can be constructed using a `unique_ptr` to host data, which serve as an initialization point for the buffer. The ownership of the host data pointer is moved to the SYCL runtime, and those data will not be available after the destruction of the buffer.

   When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed and then return, as there is no copy of data back to host, since the original pointer is invalid.

   However, if the `set_final_data()` function is used and host data pointer is valid for copying data back, then the buffer will block on destruction and copy data using the `weak_ptr` provided.

6. A buffer can be constructed using a `shared_ptr` to host data. This pointer is shared between the SYCL application and runtime. In order to allow synchronization between the the application and the runtime an `mutex` is used which will be locked by the runtime whenever the data are in use and unlock it when it no longer needs them.

   The `shared_ptr` reference counting is used in order to prevent either from destroying the buffer host data prematurely. If the `shared_ptr` is deleted from the user application before the buffer destruction, the buffer can continue securely due to the fact that the pointer hasn't be destroyed yet, but will not copy data back to the host before destruction, as the application side has already deleted its copy.

   There is no need to use the `set_final_data()` function in order to set the final data pointer if its is going to be the same as the `shared_ptr` used in construction of the buffer.

   In the case where `set_final_data()` is used and a `weak_ptr` referencing to a valid `shared_ptr` on the SYCL application side, the buffer will block and copy data back to that host memory.

7. A buffer can be constructed from a pair of iterator values. In this case, the buffer construction will copy the data from the data range defined by the iterator pair. The destructor will not copy back any data and will not block.

8. A buffer constructed from a `cl_mem` object creates a SYCL buffer that is initialized from a `cl_mem` object and may use the `cl_mem` object for the lifetime of the buffer. The destructor for this type of buffer will block until all operations on the buffer have completed and then will (if necessary) copy all modified data back into the associated `cl_mem` object. The buffer will have a single `cl_mem` object and all operations will be performed on this underlying storage.

9. An SVM buffer is created for a specific context and size of allocation. There is no host pointer parameter for this allocation, since the contents of the buffer need to be allocated for the host and device shared virtual memory. An `svm_allocator` is used for allocating and deallocation shared virtual memory for coarse grained buffers. On buffer destruction, the memory is deallocated after all the kernels that have been enqueued have completed.

As a convenience for the user, any constructor that takes a range argument can instead be passed range values as 1, 2 or 3 arguments of type `size_t`.

A buffer object can also be copied, which just copies a reference to the buffer. The buffer objects use reference counting, so copying a buffer object increments a reference count on the underlying buffer. If after destruction, the reference count for the buffer is non-zero, then no further action is taken.

A sub-buffer object can be created which is a sub-range reference to a base buffer. This sub-buffer can be used to create accessors to the base buffer, but which only have access to the range specified at time of construction of the sub-buffer.

If a buffer object is constructed from a `cl_mem` object, then the buffer is created and initialized from the OpenCL memory object. The SYCL system may copy the data to another device and/or context, but must copy it back (if modified) at the point of destruction of the buffer. The user must provide a `queue` and `event`. The memory object is assumed to only be available to the SYCL scheduler after the event has signaled and is assumed to be currently resident on the context and device signified by the `queue`.

4.3 Images

The class `image<int Dimensions, bool Depth, bool MultiSample` (Table 4.4) defines shared image data of one, two or three dimensions, that can be used by kernels in queues and has to be accessed using `accessor` classes with image accessor modes.

Image constructors are listed in Table 4.4 and methods in Table 4.5. Where relevant, it is the responsibility of the user to ensure that the format of the data matches the format described by `order` and `type`. Custom image allocators can be defined, but they need to be byte-sized allocators.

If an image object is constructed from a `cl_mem` object, then the image is created and initialized from the OpenCL memory object. The SYCL system may copy the data to the host, but must copy it back (if modified) at the point of destruction of the image. The user must provide a `queue` and `event`. The memory object is assumed to only be available to the SYCL scheduler after the event has signaled and is assumed to be currently resident on the context and device signified by the `queue`.

4.3.1 Image Interface

```
namespace cl {
namespace sycl {
enum class image_channel_order : unsigned int {
  r,
  rx,
  a,
  intensity,
  luminance,
  rg,
  rgx,
  ra,
  rgb,
  rgbx,
  rgba,
  argb,
  bgra,
  depth,
  srgb,
  srgbx,
  srgba,
  sbgra,
  abgr
};

enum class image_channel_type : unsigned int {
  snorm_int8,
  snorm_int16,
  unorm_int8,
  unorm_int16,
  unorm_short_565,
  unorm_short_555,
  unorm_int_101010,
  signed_int8,
  signed_int16,
  signed_int32,
  unsigned_int8,
  unsigned_int16,
  unsigned_int32,
  half_float,
  float,
  unorm_int_101010_2
};

template <int dimensions, bool Depth, bool MultiSample, typename AllocatorT = cl::sycl::
    image_allocator>
class image {
 public:
  image(void *hostPointer, image_format::channel_order order,
        image_format::channel_type type, const range<dimensions> &range,
        AllocatorT allocator = nullptr);

  image(void *hostPointer, image_format::channel_order order,
        image_format::channel_type type, const range<dimensions> &range,
        range<dimensions - 1> &pitch, AllocatorT allocator = nullptr);

  image(shared_ptr_class<void> &hostPointer, image_format::channel_order order,
        image_format::channel_type type, const range<dimensions> &range,
        AllocatorT allocator = nullptr);

  image(shared_ptr_class<void> &hostPointer, image_format::channel_order order,
        image_format::channel_type type, const range<dimensions> &range,
        const range<dimensions - 1> &pitch, AllocatorT allocator = nullptr);

  image(shared_ptr_class<void> &hostPointer, image_format::channel_order order,
        image_format::channel_type type, const range<dimensions> &range,
        mutex_class *m, AllocatorT allocator = nullptr);

  image(shared_ptr_class<void> &hostPointer, image_format::channel_order order,
        image_format::channel_type type, const range<dimensions> &range,
        const range<dimensions - 1> &pitch, mutex_class *m,
        AllocatorT allocator = nullptr);

  image(unique_ptr_class<void> &hostPointer, image_format::channel_order order,
        image_format::channel_type type, const range<dimensions> &range,
        AllocatorT allocator = nullptr);

  image(unique_ptr_class<void> &hostPointer, image_format::channel_order order,
        image_format::channel_type type, const range<dimensions> &range,
        const range<dimensions - 1> &pitch, AllocatorT allocator = nullptr);

  image(image_format::channel_order order, image_format::channel_type type,
        const range<dimensions> &range, AllocatorT allocator = nullptr);

  image(image_format::channel_order order, image_format::channel_type type,
        const range<dimensions> &range, const range<dimensions - 1> &pitch,
        AllocatorT allocator = nullptr);

  image(const image<dimensions, AllocatorT> &rhs);

  image<dimensions, AllocatorT> &operator=(
      const image<dimensions, AllocatorT> &rhs);

  image(cl_mem memObject, const queue &fromQueue, event availableEvent = nullptr);

  image(const image &rhs);

  ~image();

  range<dimensions> get_range();

  range<dimensions - 1> get_pitch();

  size_t get_size();

  size_t get_count();

  allocatorT get_allocator() const;

  template <access::mode accessMode,
            access::target accessTarget = access::image>
  accessor<T, dimensions, accessMode, accessTarget> get_access();

  template<T>
  set_final_data(weak_ptr_class<T> & finalPointer));
};
}  // namespace sycl
}  // namespace cl
```

Constructors	Description

`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl::image_allocator>`
`image (void * hostPointer,`
`    image_channel_order order,`
`    image_channel_type type,`
`    const range<dimensions> & range,`
`    AllocatorT allocator = nullptr)`
	Construct an image using the data from the host pointer. The type of the image data is defined by `order` and `type`.The size of the image in pixels is defined by `size`.
	During the lifetime of the image, the `hostPointer` ownership is passed to the SYCL runtime.
	An allocator instance, if provided, manages the image memory allocation on the host. If no allocator is provided, the default allocator is `cl::sycl::image_allocator` 4.1.1 will be used.

	Continued on next page

Table 4.4: Constructors for the `image` class.

Constructors	Description
`template<int dimensions,bool Depth, bool MultiSample , typename AllocatorT= cl::sycl::image_allocator> image<dimensions>(void *hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, const range<dimensions-1> &pitch, AllocatorT allocator = nullptr)`	Construct an image using the data from the host pointer `hostPointer`. The type of the image data is defined by `order` and `type` and whether it is a depth image or it supports mult-sampling. The size of the image in pixels is defined by `size` and the pitch is defined by `pitch`. During the lifetime of the image, the `hostPointer` ownership is passed to the SYCL runtime. An allocator instance, if provided, manages the image memory allocation on the host. If no allocator is provided, the default allocator is `cl::sycl::image_allocator` 4.1.1 will be used.
`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl:: image_allocator> image(shared_ptr_class<void>& hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, AllocatorT allocator = nullptr)`	Construct an image from shared host memory between the SYCL application and runtime. The type of the image data is defined by `order` and `type`. The size of the image in pixels is defined by `size`. During the lifetime of the image, the `hostPointer` ownership is passed to the SYCL runtime. An allocator instance, if provided, manages the image memory allocation on the host. If no allocator is provided, the default allocator is `cl::sycl::image_allocator` 4.1.1 will be used.
`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl:: image_allocator> image(shared_ptr_class<void>& hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, const range<dimensions-1> & pitch, AllocatorT allocator = {}))`	Construct an image using the data in host pointer. The type of the image data is defined by `order` and `type`.The size of the image in pixels is defined by `size`. The pitch of the image data, in bytes, is defined by `pitch`. The ownership is shared between the SYCL runtime and application, using the `mutex` provided and managing the synchronization from the SYCL application side. The mutex is locked by the runtime whenever the data is in use and unlocked otherwise. Data is synchronized with host pointer when the mutex is unlocked by the runtime. An allocator instance, if provided, manages the image memory allocation on the host. If no allocator is provided, the default allocator is `cl::sycl::image_allocator` 4.1.1 will be used.

Continued on next page

Table 4.4: Constructors for the image class.

Constructors	Description

`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl:: image_allocator>`
`image(shared_ptr_class<void>& hostPointer,`
`    image_channel_order order,`
`    image_channel_type type,`
`    const range<dimensions> & range,`
`    mutex_class * mutex,`
`    AllocatorT allocator = nullptr)`
	Construct an image from shared host memory between the SYCL application and runtime. The type of the image data is defined by `order` and `type`. The size of the image in pixels is defined by `size`.
	The ownership is shared between the SYCL runtime and application, using the `mutex` provided and managing the synchronization from the SYCL application side. The mutex is locked by the runtime whenever the data is in use and unlocked otherwise. Data is synchronized with host pointer when the mutex is unlocked by the runtime.
	An allocator instance, if provided, manages the image memory allocation on the host. If no allocator is provided, the default allocator is `cl::sycl::image_allocator` 4.1.1 will be used.

`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl:: image_allocator>`
`image(shared_ptr_class<void>& hostPointer,`
`    image_channel_order order,`
`    image_channel_type type,`
`    const range<dimensions> & range,`
`    const range<dimensions-1> & pitch,`
`    mutex_class * mutex,`
`    AllocatorT allocator = nullptr)`
	Construct an image using the data in host pointer. The type of the image data is defined by `order` and `type`.The size of the image in pixels is defined by `size`. The pitch of the image data, in bytes, is defined by `pitch`.
	The ownership is shared between the SYCL runtime and application, using the `mutex` provided and managing the synchronization from the SYCL application side. The `mutex` is locked by the runtime whenever the data is in use and unlocked otherwise. When the mutex is unlocked by the SYCL runtime the data is synchronized with `hostPointer`.
	An allocator instance, if provided, manages the image memory allocation on the host. If no allocator is provided, the default allocator is `cl::sycl::image_allocator` 4.1.1 will be used.

Continued on next page

Table 4.4: Constructors for the `image` class.

Constructors	Description

`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl::image_allocator>`
`image(unique_ptr_class<void>& hostPointer,`
`image_channel_order order,`
`image_channel_type type,`
`const range<dimensions> & range,`
`AllocatorT allocator = {})`
	Construct an image from a unique pointer class instance `hostPointer`. The type of the image data is defined by `order` and `type`. The size of the image in pixels is defined by `size`.
	The host memory ownership is moved to the SYCL image and unless the `get_final_data()` method is used with a host memory location that will be valid after the destruction of the image, no transfer back to host is done.
	An allocator instance, if provided, manages the image memory allocation on the host. If no allocator is provided, the default allocator is `cl::sycl::image_allocator` 4.1.1 will be used.

`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl::image_allocator>`
`image(unique_ptr_class<void>& hostPointer,`
`image_channel_order order,`
`image_channel_type type,`
`const range<dimensions> & range,`
`const range<dimensions-1> & pitch,`
`AllocatorT allocator = nullptr)`
	Construct an image from a unique pointer class instance `hostPointer`. The type of the image data is defined by `order` and `type`. The size of the image in pixels is defined by `size`. The pitch of the image data, in bytes, is defined by `pitch`.
	The host memory ownership is moved to the SYCL image and unless the `get_final_data()` method is used with a host memory location that will be valid after the destruction of the image, no transfer back to host is done.
	An allocator instance, if provided, manages the image memory allocation on the host. If no allocator is provided, the default allocator is `cl::sycl::image_allocator` 4.1.1 will be used.

`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl::image_allocator>`
`image (image_channel_order order,`
`image_channel_type type,`
`const range<dimensions> & range,`
`AllocatorT allocator = nullptr)`
	Create an image without any host data associated with it. This image will be created on the device side with no initial data. Unless the method `final_data()` is used, there will be no copy back to host on destruction. The type of the image data is defined by its `order` and `type`. The size of the image in pixels is defined by `size`.
	The default allocator is a byte allocator, it will be used for memory allocation management on host. If no allocator is provided, the default allocator is `cl::sycl::image_allocator` 4.1.1 will be used.

	Continued on next page

Table 4.4: Constructors for the `image` class.

Constructors	Description

`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl:: image_allocator>`
`image<dimensions>( image_channel_order order,`
`image_channel_type type,`
`const range<dimensions> & range,`
`const range<dimensions-1> &pitch,`
`AllocatorT allocator = nullptr)`
	Create an image without any host data associated with it. This image will be created on the device side with no initial data. Unless the method `final_data()` is used, there will be no copy back to host on destruction. The type of the image data is defined by `order` and `type`. The size of the image in pixels is defined by `size` and the pitch is defined by `pitch`.
	The default allocator is a byte allocator, it will be used for memory allocation management on host. If no allocator is provided, the default allocator is `cl::sycl:: image_allocator` 4.1.1 will be used.

`template<int dimensions, bool Depth, bool MultiSample, typename AllocatorT= cl::sycl:: image_allocator>`
`image(image<dimensions, AllocatorT> & rhs)`
	Copy construct an image as a reference to another image. Images are reference counted, so that they all point to the same underlying memory.

`template<int dimensions, bool Depth, bool MultiSample,`
`typename AllocatorT= cl::sycl::image_allocator>`
`image<dimensions>( cl_mem memObject,`
`queue fromQueue,`
`event availableEvent = nullptr)`
	Create an image from an existing OpenCL memory object associated with a context after waiting for an event signaling the availability of the OpenCL data. `memObject` is the OpenCL memory object to use. `fromQueue` is the queue associated to the memory object. Depending on whether it is a depth image or it supports multi-sampling the parameters should be set accordingly.
	`availableEvent` specifies the event to wait for if non-null. Retains a reference to the `cl_mem` object. Caller should release the passed `cl_mem` object when it is no longer needed. Note that an image created from a `cl_mem` object will only have one underlying `cl_mem` for the lifetime of the buffer and use on an incompatible queue constitues an error.

`image(const image<dimensions, AllocatorT> & rhs)`	Copy Constructor

`image<dimensions,AllocatorT> &operator=(`
`const image<dimensions,AllocatorT> &rhs)`
	Assignment operator which will be sharing the same underlying object and will be reference counted internally by the SYCL runtime.

	End of table

Table 4.4: Constructors for the `image` class.

Methods	Description
`const range<dimensions> get_range()`	Return a range object representing the size of the image in terms of the number of elements in each dimension as passed to the constructor.
`const range<dimensions-1> get_pitch()`	Return a range object representing the pitch of the image in bytes.
`size_t get_count()const`	Returns the total number of elements in the image. Equal to `get_range()[0] * ... * get_range()[dimensions-1]`.
`size_t get_size()const`	Returns the size of the image storage in bytes. The number of bytes may be greater than `get_count()*element size` due to padding of elements, rows and slices of the image for efficient access.
`allocatorT get_allocator()const`	Returns the allocator provided to the buffer.
`template<access::mode mode, access::target target= access::image>` `accessor<T, dimensions, mode, target>` `get_access(handler & command_group_handler)`	Returns a valid accessor to the image with the specified access mode and target. The value of target can be `access::image`
`template<access::mode mode, access::target target= access::host_image>` `accessor<T, dimensions, mode, target>` `get_access()`	Returns a valid accessor to the image with the specified access mode and target. The value of target can be `access::host_image`.
`void set_final_data(weak_ptr_class<T>& finalPointer)`	Function that sets the final data pointer, to be different than the original pointer given. This is mainly to be used when a *shared_-ptr* is given in the constructor and the output data will reside in a different location than the initialization data.
	End of table

Table 4.5: Methods of the `image` class.

4.3.2 Image Synchronization Rules

For the lifetime of the image object, the associated host memory must be left available to the SYCL runtime and the contents of the associated host memory is undefined until the image object is destroyed. If an image object value is copied, then only a reference to the underlying image object is copied. The underlying image object is reference-counted. Only after all image value references to the underlying image object have been destroyed is the actual image object itself destroyed.

If an image object is constructed with associated host memory, then its destructor blocks until all operations in all SYCL queues on that image object have completed. Any modifications to the image data will be copied back, if necessary, to the associated host memory. Any errors occurring during destruction are reported to any associated context's asynchronous error handler. If an image object is constructed with a storage object, then the storage object defines what synchronization or copying behaviour occurs on image object destruction.

4.3.3 Samplers

Samplers use the `cl::sycl::sampler` type which is equivalent to the OpenCL C `cl_sampler` and `sampler_t` types. Constructors for the `sampler` class are listed in Table 4.9 and methods in Table 4.10.

```
namespace cl {
namespace sycl {

enum class addressing_mode: unsigned int {
  mirrored_repeat,
  repeat,
  clamp_to_edge,
  clamp,
  none
};

enum class normalized_coordinates : unsigned int{ normalized, denormalized };

enum class filtering_mode: unsigned int { nearest, linear };

#ifdef cl_khr_mipmap_image
enum class mipmap_filtering_mode { mipmap_none, mipmap_nearest, mipmap_linear };
#endif

class sampler {
 public:
  sampler(bool normalized_coords, addressing_mode addressing,
          filtering_mode filtering);

  sampler(cl_sampler);

  ~sampler() {}

  addressing_mode get_address() const;

  filtering_mode get_filter() const;

  cl_sampler get_opencl_sampler() const;
};
}  // namespace sycl
}  // namespace cl
```

sampler_addressing_mode	**Description**
`mirrored_repeat`	Out of range coordinates will be flipped at every integer junction. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined.

Continued on next page

Table 4.6: Addressing modes description

sampler_addressing_mode	**Description**
`repeat`	Out of range image coordinates are wrapped to the valid range. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined.
`clamp_to_edge`	Out of range image coordinates are clamped to the extent.
`clamp`	Out of range image coordinates will return a border color.
`none`	For this addressing mode the programmer guarantees that the image coordinates used to sample elements of the image refer to a location inside the image; otherwise the results are undefined.
	End of table

Table 4.6: Addressing modes description

sampler_filtering_mode	**Description**
`nearest`	Chooses a color of nearest pixel.
`linear`	Performs a linear sampling of adjacent pixels.
	End of table

Table 4.7: Filtering modes description

sampler_mipmap_filtering_mode	**Description**
`mipmap_none`	Non-integer mipmap coordinates result in undefined behaviour
`mipmap_nearest`	Chooses a color of nearest pixel from nearest mipmap.
`mipmap_linear`	Performs a sampling from adjacent pixels from the image and its mipmaps.
	End of table

Table 4.8: Mipmap filterting mode description

Constructors	Description
`sampler( bool normalized_coords, sampler_addressing_mode addressing_mode, sampler_filter_mode filter_mode)`	`normalized_coords` selects whether normalized or un-normalized coordinates are used for accessing image data. `addressing_mode` specifies how out-of-range image coordinates are handled. `filtering_mode` specifies the type of filter that must be applied when reading an image.
`sampler(cl_sampler)`	Construct a sampler from an OpenCL sampler object. Retains a reference to the `cl_sampler` object. Caller should release the passed `cl_sampler` object when it is no longer needed.
	End of table

Table 4.9: Constructors for the `sampler` class.

Methods	Description
`sampler_addressing_mode get_address()const`	Return the addressing mode used to construct the sampler.
`sampler_filtering_mode get_filter()const`	Return the filter mode used to construct the sampler.
`cl_sampler get_sampler()const`	Returns the underlying *cl_sampler* object. Retains a reference to the returned `cl_sampler` object. Caller should release it when finished.
	End of table

Table 4.10: Methods for the `sampler` class.

4.4 Sharing Host Memory With The SYCL Data Management Classes

In SYCL, in order to allow the SYCL runtime to do the memory management and allow for data dependencies, there are two classes defined: `buffer` and `image`. The default behavior for them is that a 'raw' pointer is given at the construction of the data management class with full ownership to use it until the destruction of the SYCL object.

In this section we go in greater detail into sharing or explicitly not sharing host memory with the SYCL data classes. We will use the `buffer` class as an example. However, the same rules apply to images, as well.

4.5 Default behavior

When using a SYCL buffer, the ownership of the pointer passed to the constructor of the class is, by default, passed to the SYCL runtime and that pointer cannot be used on the host side until the buffer or image is destroyed. A SYCL application can use memory managed by a SYCL buffer within the buffer scope by using a `host_accessor`, as defined in 4.7. However, there is no guarantee that the host_accessor uses the same memory as the original host address used in its constructor.

The pointer passed in is the one use to copy data back, if needed, before buffer destruction. This host pointer will not de-allocated by the runtime when the buffer is back on the host, and the data is always copied back from the device (if there was a need for it).

4.5.1 SYCL ownership of the host memory

In the case where there is host memory to be used for initialization of data but there is no intention of using that host memory after the buffer is destroyed, then the buffer can take full ownership of that host memory.

When a buffer owns the host pointer there is no copy back, by default. In this situation, the SYCL application may pass a unique pointer to the host data, which will be then used by the runtime internally to initialize the data in the device.

If the pointer contained in the `unique_ptr` is null, pointer is initialized internally in the runtime but no data is copied in. This will be the generic case of a buffer constructor that takes no host pointer.

In this case the buffer and image constructors used will be the following:

- `template<typename T, int dimensions, typename U=default_allocator> buffer(cl::sycl::unique_ptr_class<T> hostData,range<dimensions> myRange);`
- `template<int dimensions, typename U=image_allocator> image(cl::sycl::unique_ptr_class<void> hostData, cl_channel_order order, cl_channel_type type, range<dimensions> size)`

For example, the following could be used:

```
{
 cl::sycl::unique_ptr_class<int> ptr(data);
  buffer<int, 1> b(std::move(ptr));
  // ptr is not valid anymore
  // There is nowhere to copy data back
}
```

However, optionally, the buffer::set_final_data() can be set to a `cl::sycl::weak_ptr_class` to enable copying data back, to another host memory address that is going to be valid after buffer construction.

```
{
  cl::sycl::unique_ptr_class<int> ptr(data);
  buffer<int, 1> b(std::move(ptr));
  // ptr is not valid anymore
  // There is nowhere to copy data back
  b.set_final_data(cl::sycl::weak_ptr_class<int>(....))
}
```

4.5.2 Shared SYCL ownership of the host memory

When a `shared_ptr` is passed to the buffer constructor, then the buffer object and the developer's application data is shared. If the shared pointer is still used in the application's code then the data will be copied back from the buffer or image and be available to the application after the buffer or image is destroyed.

If the data pointed by the shared object is initialized to some data, then that data is used to initialize the buffer. If the shared pointer is null, the pointer is initialized by the runtime internally (and, therefore, the user can use it afterwards on the host).

When the buffer is destroyed, if the number of copies of the shared pointer outside the runtime is 0, there is no user-side shared pointer to read the data and therefore the data is not copied out, and the buffer destructor does not need to wait for the data processes to be finished from OpenCL, as the outcome is not needed on the application's side.

This behavior can be overridden using the `set_final_data()` method of the buffer class, which will force the buffer destructor to wait until the data is copied to wherever the `set_final_data()` method has requested the data (or not wait nor copy if set final data is `nullptr`).

```
{
  cl::sycl::shared_ptr_class<int> ptr(data);
  {
    buffer<int, 1> b(ptr, range<2>(10, 10));
  }  // Data is copied back because there is an user side shared ptr
}
```

```
{
  cl::sycl::shared_ptr_class<int> ptr(data);
  {
    buffer<int, 1> b(ptr, range<2>(10, 10));
    ptr.release();
  } // Data is not copied back, there is no user side shared ptr.
}
```

4.6 Synchronisation Primitives

When the user wants to use the buffer simultaneously in the SYCL runtime and its own code (e.g. a multi-threaded mechanism) and wants to use manual synchonization without host accessors, a pointer to a `cl::sycl::mutex` can be passed to the buffer constructor.

The runtime promises to lock the mutex whenever the data is in use and unlock it when it no longer needs it.

```
{
  cl::sycl::mutex_class * m;
  auto shD = std::make_shared<int>(42)
  {
  buffer<int, 1> b(shD, m);

    m.lock();
    // User accesses the data
    do_something(shD);
    m.unlock();

  }
}
```

When the runtime releases the mutex the user is guaranteed to have the data copied back on the shared pointer - unless the final data destination has been changed using the method `set_final_data`.

4.7 Accessors

Accessors manage the access to data in buffers, coarse-grained SVM allocations and images. The user specifies the type of access to the data and the SYCL implementation ensures that the data is accessible in the right way on the right device in a queue. This separation allows a SYCL implementation to choose an efficient way to provide access to the data within an execution schedule. Common ways of allowing data access to shared data in a heterogeneous system include: copying between different memory systems; mapping memory into different device address spaces, or direct sharing of data in memory. The buffers and images are SYCL runtime classes that provide the management of the data.

Accessors are *device accessors* by default, but can optionally be specified as being host accessors. Device accessors can only be constructed within command groups and provide access to the underlying data in a queue. Only a kernel can access data using a device accessor. Constructing a device accessor is a non-blocking operation: the synchronization is added to the queue, not the host.

Host accessors can be created outside command groups and give immediate access to data on the host. Construction of host accessors is blocking, waiting for all previous operations on the underlying buffer or image to complete, including copying from device memory to host memory. Any subsequent device accessors need to block until the processing of the host accessor is done and the data is copied to the device.

Accessors always have an *element data type*. When accessing a buffer, the accessor's element data type must match the same data type as the buffer. An image accessor may have an element data type of either an integer vector or a floating-point vector. The image accessor data type provides the number and type of components of the pixel read. The actual format of the underlying image data is not encoded in the accessor, but in the image object itself. The data types allowed by buffers can be SYCL data types or user defined types that are aligned with the restrictions on kernel parameter passing 5.9.

There are two enumeration types inside `namespace cl::sycl::access`, `access::mode` and `access::target`. These two enumerations define both the access mode and the data that the accessor is targeting.

4.7.0.1 Access modes

The `mode` enumeration, shown in Table 4.11, has a base value, which must be provided.

The user must provide the *access mode* when defining an accessor. This information is used by the scheduler to ensure that any data dependencies are resolved by enqueuing any data transfers before or after the execution of a kernel. If a command group contains only *discard write mode* accesses to a buffer, then the previous contents of the buffer (or sub-range of the buffer, if provided) are not preserved. If a user wants to modify only certain parts of a buffer, preserving other parts of the buffer, then the user should specify the exact sub-range of modification of the buffer. A command-group's access to a specific buffer is the union of all access modes to that buffer in the command group, regardless of construction order. Atomic access is only valid to `local`, `global_buffer` and `host_buffer` targets (see next section).

access::mode	**Description**
`read`	read-only access
`write`	write-only access. Previous contents not discarded.
`read_write`	read and write access
`discard_write`	write-only access. Previous contents discarded.
`discard_read_write`	read and write access. Previous contents discarded.
`atomic`	atomic access.

Table 4.11: Enumeration of access modes available to accessors.

4.7.0.2 Access targets

The `target` enumeration, shown in Table 4.12, describes the type of object to be accessed via the accessor. The different values of the `target` enumeration require different constructors for the accessors.

access::target	**Description**
`global_buffer`	Access `buffer` via global memory.
`constant_buffer`	Access `buffer` via constant memory.
`local`	Access work-group-local memory.
`image`	Access an `image`.
`host_buffer`	Access a `buffer` immediately in host code.
`host_image`	Access an `image` immediately in host code.
`image_array`	Access an array of `image`s on a device.

Table 4.12: Enumeration of access modes available to accessors.

4.7.0.3 Accessor class

The accessor makes data available to host code, or to a specific kernel. The accessor is parameterized with the type and number of dimensions of the data. An accessor also has a mode, which defines the operations possible on the underlying data (see Table 4.11) and a target (see Table 4.12, which defines the type of data object to be modified. The constructors and methods available on an accessor depend on the mode and target.

The generic methods for the accessor class are defined in Table 4.14. Available methods are limited by the access mode and target provided as template parameters to the accessor object.

```
namespace cl {
namespace sycl {
namespace access {
enum class mode {
  read = 1,
  write,
  read_write,
  discard_write,
  discard_read_write,
  atomic
};

enum class target {
  global_buffer = 2014,
  constant_buffer,
  local,
  image,
  host_buffer,
  host_image,
  image_array
};
}  // namespace access

template <
  typename elementType,
  int dimensions,
  access::mode accessMode,
  access::target accessTarget>
class accessor {
 public:
  using value_type = T;
  using reference = value_type&;
  using const_reference = const value_type&;

  // Available only for: access::global_buffer, access::host_buffer,
  // and access::constant_buffer
  accessor(buffer<elementType, dimensions> &bufferRef,
           handler &commandGroupHandler);

  accessor(buffer<elementType, dimensions> &bufferRef,
           handler &commandGroupHandler,
           range<dimensions> offset, range<dimensions> range);

  // Available only for: access::image and access::host_image
```

```
accessor(image<dimensions> &imageRef, handler &commandGroupHandler);

// Available only for: access::local
accessor(range<dimensions> allocationSize, handler &commandGroupHandler);

size_t get_size();

// Methods available to buffer targets
// Available when access_mode includes non-atomic write permissions
// and dimensions>0
elementType &operator[](id<dimensions>);
// Available when access_mode is read-only and dimensions>0
const elementType &operator[](id<dimensions>);
// Available when access_mode includes non-atomic write permissions
// and dimensions==0
elementType &operator*();

// Available when access_mode is read-only and dimensions==0
const elementType &operator[]();
elementType operator*();

// Available when dimensions==0 and access mode is non-atomic.
operator elementType();

// Methods available for image targets
__undefined__ &operator()(sampler sample);
__undefined__ &operator[](id<dimensions>);

// Available when the accessor is to an image array
// Returns an accessor to a particular slice
accessor<elementType, 2, mode, image> operator[](size_t index)

// Available when the access target is global_buffer, host_buffer,
// or local, and access mode is atomic
// and dimensions>0
atomic<elementType> &operator[](id<dimensions);

// Available when the access target is global_buffer, host_buffer,
// or local, and access mode is atomic
// and dimensions==0
atomic<elementType> operator()();
atomic<elementType> operator*();


//Available when access target is local and access mode is non-atomic
local_ptr<elementType> get_pointer();

//Available when access target is global_buffer and access mode is non-atomic
global_ptr<elementType> get_pointer();

//Available when access target is constant_buffer
constant_ptr<elementType> get_pointer();

//Available when access target is generic and access mode is non-atomic
generic_ptr<elementType> get_pointer();
```

```

  //Available when access target is host_buffer and access mode is non-atomic
  elementType * get_pointer();
};

}  // namespace sycl
}  // namespace cl
```

Constructors	Description
`accessor(buffer<elementType, dimensions> &bufferRef, handler &commandGroupHandler)`	Construct a buffer accessor from a buffer using a command group handler object from the command group functor. Constructor only available for access modes `global_buffer`, `host_buffer` `constant_buffer` see Table 4.11. `access_target` defines the form of access being obtained. See Table 4.12.
`accessor( buffer<elementType, dimensions> &bufferRef, handler &commandGroupHandler, range<dimensions> offset, range<dimensions> range)`	Construct a buffer accessor from a buffer given a specific range for access permissions and an offset that provides the starting point for the access range using an instance of the `execution_handle`. This accessor limits the processing of the buffer to the [offset, offset+range] for every dimension. Any other parts of the buffer will be unaffected. Constructor only available for access modes `global_buffer`, `host_buffer` or `constant_buffer` (see Table 4.11). `access_target` defines the form of access being obtained (see Table 4.12). This accessor is recommended for *discard_write* and *discard_read_write* access modes, when the unaffected parts of the processing should be retained.
`accessor(image<dimensions> &imageRef, handler &commandGroupHandler)`	Construct an image accessor from an image using a command group handler object in the command group functor. Constructor only available if `accessMode` is `image`, or `host_image`, see Table 4.11. `access_target` defines the form of access being obtained. See Table 4.12. The `elementType` for image accessors must be defined by the user and is the type returned by any sampler or accessor read operation, as well as the value accepted by any write operation. It must be an `int`, `unsigned int` or `float` vector of 4 dimensions.

Continued on next page

Table 4.13: Accessor constructors.

Constructors	Description
`accessor(range<dimensions> allocationSize, handler &commandGroupHandler)`	Construct an accessor of `dimensions` dimensions with elements of type `elementType` using the passed range to specify the size in each dimension. It needs as a parameter an instance of the `execution_handle`. Constructor only available if `accessMode` is `local`, see Table 4.11.

End of table

Table 4.13: Accessor constructors.

Methods	Description
`size_t get_size()`	Returns the size of the underlying buffer in number of elements.
`elementType &operator[](id<dimensions>)`	Return a writeable reference to an element in the buffer. Available when `mode` includes non-atomic write permissions and `dimensions` 0.
`elementType &operator[](int)`	Return a writeable reference to an element in the buffer. Available when `mode` includes non-atomic write permissions and `dimensions` == 1.
`const elementType &operator[](id<dimensions>)`	Return the value of an element in the buffer. Available when `mode` is read-only and `dimensions` 0.
`const elementType &operator[](int)`	Return the value of an element in the buffer. Available when `mode` is read-only and `dimensions` == 1.
`elementType &operator[]()`	Return a writeable reference to the element in the buffer. Available when `mode` includes non-atomic write permissions and `dimensions` == 0.
`const elementType &operator[]()`	Return the value of the element in the buffer. Available when `mode` is read-only and `dimensions` == 0.
`operator elementType()`	Return the value of the element in the buffer. Available when `mode` is non-atomic and `dimensions` == 0.
`accessor<elementType, 2, mode, image> operator[](size_t index)`	Returns an accessor to a particular plane of an image array. Available when accessor acts on an image array.
`__undefined__ <dimensions-1> &operator[](int)`	Return an intermediate type with an additional subscript operator for each subsequent dimension of buffer where (dimensions ¿ 0). Available when `mode` non-atomic and for access mode read only the return type is const.

Continued on next page

Table 4.14: Methods for the `accessor` class.

Methods	Description
`__undefined__ &operator()(sampler sample)`	Return the value of an element in the image given a sampler. Available only for the case of an image accessor type.
`__undefined__ &operator [ ] ( id < dimensions > )`	Return the value of an element in the image with a sampler-less read. Available only for the case of an image accessor type.
`atomic<elementType> &operator[](id<dimensions>)`	Returns a reference to an atomic object, when the accessor if of type `access::global_buffer`, `access::local_buffer`, `access::host_buffer`, the target mode is `access::mode::atomic` and `dimensions >0` .
`atomic<elementType> &operator()()`	Returns a reference to an atomic object, when the accessor if of type `access::global_buffer`, `access::local_buffer`, `access::host_buffer`, the target mode is `access::mode::atomic` and `dimensions` ==0.
`atomic<elementType> &operator*()`	Returns a reference to an atomic object, when the accessor if of type `access::global_buffer`, `access::local_buffer`, `access::host_buffer`, the target mode is `access::mode::atomic` and `dimensions` ==0.
`local_ptr<elementType> get_pointer()`	Returns the accessor pointer, when the accessor is of type `access::local` and `mode` is non-atomic.
`global_ptr<elementType> get_pointer()`	Returns the accessor pointer, when the accessor is of type `access::global_buffer` and `mode` is non-atomic.
`constant_ptr<elementType> get_pointer()`	Returns the accessor pointer, when the accessor is of type `access::constant_buffer`
`generic_ptr<elementType> get_pointer()`	Returns the accessor pointer, when the accessor is of type `access::global_ptr`
`elementType* get_pointer()`	Returns the accessor pointer, when the accessor is of type `access::host_buffer` and `mode` is non-atomic.
	End of table

Table 4.14: Methods for the `accessor` class.

4.7.0.4 Buffer accessors

Accessors to buffers are constructed from a buffer with the same element data type and dimensionality as the accessor. A buffer accessor uses *global* memory by default, but can optionally be set to use *constant* memory. Accessors that use constant memory are restricted by the underlying OpenCL restrictions on device constant memory, i.e. there is a maximum total constant memory usable by a kernel and that maximum is specified by the OpenCL device. Only certain methods and constructors are available for buffer accessors.

The array operator `[id<dimensions>]` provides access to the elements of the buffer. The user can provide an index as an `id` parameter of the same dimensionality of the buffer, or just like C++ arrays, can provide one array operator per dimension, with individual indices of type `size_t` (e.g. `myAccessor[i][j][k]`).

The address space for the index operator matches that of the accessor target. For an `access::global_buffer`, the address space is `global`. For an `access::constant_buffer`, the address space is `global`.

Accessors to buffers can be constructed to only access a *sub-range* of the buffer. The sub-range restricts access to just that range of the buffer, which the scheduler can use as extra information to extract more parallelism from queues as well as restrict the amount of information copied between devices.

4.7.0.5 Image accessors

Accessors that target images must be constructed from images of the same dimensionality as the accessor. The `target` parameter must be either `image` or `host_image`. The `dataType` parameter must be a 4 dimension vector of `unsigned int`, `int` or `float`. The array operator `[id<dimensions>]` provides samplerless reading and writing of the image. The user can provide an index as an `id` parameter of the same dimensionality of the image, or just like C++ arrays, can provide one array operator per dimension, with individual indices of type `size_t` (e.g. `myAccessor[i][j][k]`). The bracket operator takes a `sampler` (see 4.3.3) parameter, which then allows floating-point sampler-based reading using the array operator (e.g. `myAccessor(mySampler)[my2dFloatVector]`).

To enable the reading and writing of pixels with and without samplers, using standard C++ operators, there are two internal classes: `__image_ref` and `_ _sampler`. These classes only exist to ensure that assignment to pixels uses image write functions and reading the value of pixels uses image read functions.

There are restrictions that apply to `cl::sycl::access::mode` depending on the SYCL device they are used. For OpenCL 2.x devices all the access modes are availables for images. In OpenCL 1.2 devices, there are extensions that can allow `access::mode::read_write` on images, but this is not a core feature of OpenCL 1.2 devices. It is the developer's responsibility to check whether there is this capability on the target SYCL device. Core capabilities for OpenCL 1.2 are `access::mode::read, access::mode::write` and `access::discard_write`, whereas for OpenCL 2.2 they are `access::mode::read, access::mode::write, access::mode::read_write, access ::mode::discard_write` and `access::mode::discard_read_write`.

4.7.0.6 Local accessors

Accessors can also be created for *local* memory, to enable pre-allocation of local buffers used inside a kernel. These accessors are constructed using `cl::sycl::range`, which defines the size of the memory to be allocated on a per work-group basis and must be constructed with an access target of `local`. Local memory is only shared across a work-group. A local accessor can provide a `local_ptr` to the underlying data within a kernel and is only usable within a kernel. The host has no access to the data of the local buffer and cannot read or write to the data, so the accessor cannot read or write data back to the host. There can be no associated host pointer for a local buffer or data transfers.

Local accessors are not valid for single-task or basic `parallel_for` invocations.

4.7.0.7 Host accessors

Host accessors have a target of `access::host_buffer` or `access::host_image`. Unlike other accessors, host accessors should be constructed outside of any command group and they do not require an `execution_handler` . The constructor will block until the data is ready for the host to access, while the destructor will block any further operations on the data in any SYCL queue. There are no special constructor or method signatures for host accessors, so there are is no table for special host accessors here (see buffer and image accessors above).

Host accessors are constructed outside command groups and not associated with any queue, so any error reporting is synchronous. By default, error reporting is via C++ exceptions.

4.7.0.8 Accessor capabilities and restrictions

Accessors provide access on the device or on the host to a buffer or image. The access modes allowed depend on the accessor type and target. A device accessor grants access to a kernel inside a command group functor, and depending on the access target, there are different accesses allowed. A host accessor grants access to the host program to the access target. Tables 4.15, 4.16 and 4.17 show all the permitted access modes depending on target.

Accessor Type	Access Target	Access mode	Data Type	Description

Device	`global_buffer`	`read`
		`write`
		`read_write`
		
		`discard_write`
		`discard_read_write`
			All available data types supported in SYCL.	Access a buffer allocated in global memory on the device.

Device	`constant_buffer`	`read`	All available data types supported in SYCL.	Access a buffer allocated in constant memory on the device.

Host	`host_buffer`	`read`
		`write`
		`read_write`
		`discard_write`
		`discard_read_write`
			All available data types supported in SYCL.	Access a host allocated buffer on host.

Device	`local`	`read`
		`write`
		`read_write`
			All supported data types in local memory	Access work-group local buffer, which is not associated with a host buffer.
				This is only accessible on device.

Table 4.15: Description of all the `accessor` types and modes with their valid combinations for buffers and local memory

Rules for casting apply to the accessors, as there is only a specific set of permitted conversions.

Accessor Type	Access Target	Access mode	Data Type	Description

Device	`image`	`read`
		`write`
		`read_write`
		`discard_-`
		`writediscard_read_-`
		`write`
			`uint4, int4,`
			`float4, half4`
				Access an image on device.

Host	`host_image`	`read`
		`write`
		`read_write`
		`discard_-`
		`writediscard_read_-`
		`write`
			`uint4, int4,`
			`float4, half4`
				Access an image on the host.

Device	`image_array`	`read`
		`write`
		`read_write`
		`discard_-`
		`writediscard_read_-`
		`write`
			`uint4, int4,`
			`float4, half4`
				Access an array of images on device.

Table 4.16: Description of all the `accessor` types and modes with their valid combinations for images

Accessor Types	Original Accessor Target	Original Access Mode	Converted Accessor Target	Converted Access Mode

Device	`global_buffer`	`read_write`	`global_buffer`	`read`
				`write`
				`discard_read_write`

Device	`local_buffer`	`read_write`	`local_buffer`	`read` `write`

Host	`host_buffer`	`read_write`	`host_buffer`	`read`
				`write`
				`discard_read_write`

Table 4.17: Description of the `accessor` to `accessor` conversions allowed

4.8 Address space classes

In OpenCL, there are four different address spaces. These are: global, local, constant and private. In the OpenCL 2.2 C++ kernel language, these address spaces can be specified using the templated types `local<T>`, `global<T>`,`private<T>` and `constant<T>`. Or, if left unspecified, they default to *generic* pointer, which overlaps with *global,local* and *private* address spaces and is allocated in one of those address spaces at runtime. In OpenCL C 1.2, the address spaces are manually specified using OpenCL-specific keywords.

In SYCL, the device compiler is expected to auto-deduce the address space for pointers in common situations of pointer usage. However, there are situations where auto-deduction is not possible. If the target device is a OpenCL 2.x device and auto-deduction at compile time wasn't possible, then the pointer will be defaulted to *generic* pointer type. However, if the device is an OpenCL 1.2 device, then an error will occur.

Here are the most common situations where auto-deduction may fail:

- When linking SYCL kernels with OpenCL C 1.2 functions. In this case, it is necessary to specify the address space for any pointer parameters when declaring an `extern "C"` function.
- For OpenCL 1.2 devices, when declaring data structures with pointers inside, it is not possible for the SYCL compiler to deduce at the time of declaration of the data structure what address space pointer values assigned to members of the structure will be. So, in this case, the address spaces will have to be explicitly declared by the developer. For OpenCL 2.x devices, declaring the address spaces may prove more performant depending on the platform.
- When a pointer is declared as a variable, but not initialized, then address space deduction is not automatic. For OpenCL 1.2, an explicit pointer class should be used, or the pointer should be initialized at declaration.

4.8.1 Explicit pointer classes

Explicit pointer classes are just like pointers: they can be converted to and from pointers with compatible address spaces, qualifiers and types. Assignment between explicit pointer types of incompatible address spaces is illegal. In SYCL 1.2, all address spaces are incompatible with all other address spaces. In SYCL 2.2, the generic address space is compatible with the global, local and private (but not constant) address spaces. Conversion from an explicit pointer to a C++ pointer preserves the address space.

In order to facilitate SYCL/OpenCL C interoperability, the `pointer` type is provided within each explicit pointer class. It is an implementation-defined type which corresponds to the underlying OpenCL C pointer type and can be used in `extern "C"` function declarations for OpenCL C functions called from SYCL kernels.

Explicit Pointer Classes	OpenCL Address Space	Compatible Accessor Target
`global_ptr`	`global_ptr`	global_buffer
`constant_ptr`	`constant_ptr`	constant_buffer
`local_ptr`	`local_ptr`	local
`private_ptr`	`private_ptr`	none
`generic_ptr`	none	global_buffer and local
		End of table

Table 4.18: Description of the pointer classes

An overview of the interface provided for all the explicit pointer classes is the following. For the full interface, please refer to B.1.

```
namespace cl {
namespace sycl {

template <typename ElementType>
class global_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr global_ptr();
  global_ptr(pointer);
  template <access::mode Mode>
  global_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  global_ptr(const global_ptr&);
  global_ptr &operator=(global_ptr &&r);
  constexpr global_ptr(nullptr);
  ~global_ptr();

  global_ptr &operator=(pointer r);
  global_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(global_ptr& r);
  global_ptr &operator++();
  global_ptr operator++(int);
  global_ptr &operator--();
  global_ptr operator--(int);
  global_ptr &operator+=(difference_type r);
  global_ptr &operator-=(difference_type r);
  global_ptr operator+(difference_type r);
  global_ptr operator-(difference_type r);

  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};

template <typename ElementType>
class constant_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr constant_ptr();
  constant_ptr(pointer);
  template <access::mode Mode>
  constant_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  constant_ptr(const constant_ptr&);
  constant_ptr &operator=(constant_ptr &&r);
  constexpr constant_ptr(nullptr);
  ~constant_ptr();

  constant_ptr &operator=(pointer r);
  constant_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(constant_ptr& r);
  constant_ptr &operator++();
  constant_ptr operator++(int);
  constant_ptr &operator--();
  constant_ptr operator--(int);
  constant_ptr &operator+=(difference_type r);
  constant_ptr &operator-=(difference_type r);
  constant_ptr operator+(difference_type r);
  constant_ptr operator-(difference_type r);

  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};

template <typename ElementType>
class local_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr local_ptr();
  local_ptr(pointer);
  template <access::mode Mode>
  local_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  local_ptr(const local_ptr&);
  local_ptr &operator=(local_ptr &&r);
  constexpr local_ptr(nullptr);
  ~local_ptr();

  local_ptr &operator=(pointer r);
  local_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(local_ptr& r);
  local_ptr &operator++();
  local_ptr operator++(int);
  local_ptr &operator--();
  local_ptr operator--(int);
  local_ptr &operator+=(difference_type r);
  local_ptr &operator-=(difference_type r);
  local_ptr operator+(difference_type r);
  local_ptr operator-(difference_type r);
  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};

template <typename ElementType>
class private_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr private_ptr();
  private_ptr(pointer);
  template <access::mode Mode>
  private_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  private_ptr(const private_ptr&);
  private_ptr &operator=(private_ptr &&r);
  constexpr private_ptr(nullptr);
  ~private_ptr();

  private_ptr &operator=(pointer r);
  private_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(private_ptr& r);
  private_ptr &operator++();
  private_ptr operator++(int);
  private_ptr &operator--();
  private_ptr operator--(int);
  private_ptr &operator+=(difference_type r);
  private_ptr &operator-=(difference_type r);
  private_ptr operator+(difference_type r);
  private_ptr operator-(difference_type r);
  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};

template <typename ElementType>
class generic_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // default generic space and forces disabling the address-space deduction
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr generic_ptr();
  generic_ptr(pointer);
  template <access::mode Mode>
  generic_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  generic_ptr(const generic_ptr&);
  generic_ptr &operator=(generic_ptr &&r);
  constexpr generic_ptr(nullptr);
  ~generic_ptr();

  generic_ptr &operator=(pointer r);
  generic_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(generic_ptr& r);
  generic_ptr &operator++();
  generic_ptr operator++(int);
  generic_ptr &operator--();
  generic_ptr operator--(int);
  generic_ptr &operator+=(difference_type r);
  generic_ptr &operator-=(difference_type r);
  generic_ptr operator+(difference_type r);
  generic_ptr operator-(difference_type r);
  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};

}  // namespace sycl
}  // namespace cl
```

Constructors	Description

`template <typename ElementType>`
`    global_ptr(pointer)`
	Constructs a `global_ptr` from the underlying ElementType pointer.

`template <access::mode Mode>`
`    global_ptr(`
`    accessor<ElementType, 1, Mode, global_buffer>)`
	Constructs a `global_ptr` from an accessor of `access::target::global_buffer`.

`template <typename ElementType>`
`    global_ptr(const global_ptr &)`
	Copy constructor.

	End of table

Table 4.19: Constructors for `global_ptr` explicit pointer class.

Operators	Description

`template <typename ElementType>`
`    ElementType &operator*()`
	Returns the ElementType of the de-referenced pointer class.

`template <typename ElementType>`
`    ElementType &operator[](size_t i)`
	Subscript operator.

`template <typename ElementType>`
`    operator pointer()`
	Returns the underlying pointer type of the `global_ptr` class

	End of table

Table 4.20: Operators on the `global_ptr` explicit pointer class.

Constructors	Description

`template <typename ElementType>`
`    constant_ptr(pointer)`
	Constructs a `constant_ptr` from the underlying ElementType pointer.

`template <access::mode Mode>`
`    constant_ptr(`
`    accessor<ElementType, 1, Mode, constant_buffer>)`
	Constructs a `constant_ptr` from an accessor of `access::target::constant_buffer`.

`template <typename ElementType>`
`    constant_ptr(const constant_ptr &)`
	Copy constructor.

	End of table

Table 4.21: Constructors for `constant_ptr` explicit pointer class.

Operators	Description

`template <typename ElementType>`
`    ElementType &operator*()`
	Returns the ElementType of the de-referenced pointer class.

`template <typename ElementType>`
`    ElementType &operator[](size_t i)`
	Subscript operator.

`template <typename ElementType>`
`    operator pointer()`
	Returns the underlying pointer type of the `constant_ptr` class

	End of table

Table 4.22: Operators on the `constant_ptr` explicit pointer class.

Constructors	Description
`template <typename ElementType>` `local_ptr(pointer)`	Constructs a `local_ptr` from the underlying ElementType pointer.
`template <access::mode Mode>` `local_ptr(` `accessor<ElementType, 1, Mode, constant_buffer>)`	Constructs a `local_ptr` from an accessor of `access::target::local`.
`template <typename ElementType>` `local_ptr(const local_ptr &)`	Copy constructor.
	End of table

Table 4.23: Constructors for `local_ptr` explicit pointer class.

Operators	Description
`template <typename ElementType>` `ElementType &operator*()`	Returns the ElementType of the de-referenced pointer class.
`template <typename ElementType>` `ElementType &operator[](size_t i)`	Subscript operator.
`template <typename ElementType>` `operator pointer()`	Returns the underlying pointer type of the `local_ptr` class
	End of table

Table 4.24: Operators on the `local_ptr` explicit pointer class.

Constructors	Description
`template <typename ElementType>` `private_ptr(pointer)`	Constructs a `private_ptr` from an Element-Type pointer.
`template <typename ElementType>` `private_ptr(const private_ptr &)`	Copy constructor.
	End of table

Table 4.25: Constructors for `private_ptr` explicit pointer class.

Operators	Description
`template <typename ElementType>` `ElementType &operator*()`	Returns the ElementType of the de-referenced pointer class.
`template <typename ElementType>` `ElementType &operator[](size_t i)`	Subscript operator.
`template <typename ElementType>` `operator pointer()`	Returns the underlying pointer type of the `private_ptr` class
	End of table

Table 4.26: Operators on the `private_ptr` explicit pointer class.

Constructors	Description

`template <typename ElementType>`
`    generic_ptr(pointer)`
	Constructs a `generic_ptr` from the underlying ElementType pointer.

`template <access::mode Mode>`
`    generic_ptr(`
`    accessor<ElementType, 1, Mode, generic_buffer>)`
	Constructs a `generic_ptr` from an accessor of `access::target::generic_buffer`.

`template <typename ElementType>`
`    generic_ptr(const generic_ptr &)`
	Copy constructor.

	End of table

Table 4.27: Constructors for `generic_ptr` explicit pointer class, which will force disabling auto-deduction of address space pointers and let the pointers be deduced at runtime by defaulting them to the OpenCL generic address space.

Operators	Description

`template <typename ElementType>`
`    ElementType &operator*()`
	Returns the ElementType of the dereferenced pointer class.

`template <typename ElementType>`
`    ElementType &operator[](size_t i)`
	Subscript operator.

`template <typename ElementType>`
`    operator pointer()`
	Returns the underlying pointer type of the `generic_ptr` class

	End of table

Table 4.28: Operators on the `generic_ptr` explicit pointer class.

Non-member functions	Description

`template <typename ElementType>`
`    bool operator==(const global_ptr<ElementType>& lhs,`
`    const global_ptr<ElementType>& rhs)`
	Comparison operator == for `global_ptr` class.

`template <typename ElementType>`
`    bool operator!=(const global_ptr<ElementType>& lhs,`
`    const global_ptr<ElementType>& rhs)`
	Comparison operator != for `global_ptr` class.

`template <typename ElementType>`
`    bool operator<(const global_ptr<ElementType>& lhs,`
`    const global_ptr<ElementType>& rhs)`
	Comparison operator < for `global_ptr` class.

`template <typename ElementType>`
`    bool operator>(const global_ptr<ElementType>& lhs,`
`    const global_ptr<ElementType>& rhs)`
	Comparison operator > for `global_ptr` class.

`template <typename ElementType>`
`    bool operator>=(const global_ptr<ElementType>& lhs,`
`    const global_ptr<ElementType>& rhs)`
	Comparison operator >= for `global_ptr` class.

	Continued on next page

Table 4.29: Non-member functions of the explicit pointer classes.

Non-member functions	Description

`template <typename ElementType>`
`    bool operator<=(const global_ptr<ElementType>& lhs,`
`    const global_ptr<ElementType>& rhs)`
	Comparison operator <= for `global_ptr` class.

`template <typename ElementType>`
`    bool operator!=(const global_ptr<ElementType>& lhs, nullptr_t rhs)`
	Comparison operator != for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator!=(nullptr_t lhs, const global_ptr< ElementType>& rhs)`
	Comparison operator != for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator==(const global_ptr<ElementType>& lhs, nullptr_t rhs)`
	Comparison operator == for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator==(nullptr_t lhs, const global_ptr< ElementType>& rhs)`
	Comparison operator == for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator>(const global_ptr<ElementType>& lhs, nullptr_t rhs)`
	Comparison operator > for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator>(nullptr_t lhs, const global_ptr< ElementType>& rhs)`
	Comparison operator > for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator<(const global_ptr<ElementType>& lhs, nullptr_t rhs)`
	Comparison operator < for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator<(nullptr_t lhs, const global_ptr< ElementType>& rhs)`
	Comparison operator < for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator`
¿=$(constglobal_ptr < ElementType > \&lhs, nullptr_trhs)$
	Comparison operator >= for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator>=(nullptr_t lhs, const global_ptr< ElementType>& rhs)`
	Comparison operator >= for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator<=(const global_ptr<ElementType>& lhs, nullptr_t rhs)`
	Comparison operator <= for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator<=(nullptr_t lhs, const global_ptr< ElementType>& rhs)`
	Comparison operator <= for `global_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator==(const constant_ptr<ElementType>& lhs,`
`    const constant_ptr<ElementType>& rhs)`
	Comparison operator == for `constant_ptr` class.

	Continued on next page

Table 4.29: Non-member functions of the explicit pointer classes.

Non-member functions	Description
`template <typename ElementType> bool operator!=(const constant_ptr<ElementType>& lhs, const constant_ptr<ElementType>& rhs)`	Comparison operator != for `constant_ptr` class.
`template <typename ElementType> bool operator<(const constant_ptr<ElementType>& lhs, const constant_ptr<ElementType>& rhs)`	Comparison operator < for `constant_ptr` class.
`template <typename ElementType> bool operator>(const constant_ptr<ElementType>& lhs, const constant_ptr<ElementType>& rhs)`	Comparison operator > for `constant_ptr` class.
`template <typename ElementType> bool operator>=(const constant_ptr<ElementType>& lhs, const constant_ptr<ElementType>& rhs)`	Comparison operator >= for `constant_ptr` class.
`template <typename ElementType> bool operator<=(const constant_ptr<ElementType>& lhs, const constant_ptr<ElementType>& rhs)`	Comparison operator <= for `constant_ptr` class.
`template <typename ElementType> bool operator!=(const constant_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator <= for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator!=(nullptr_t lhs, const constant_ptr<ElementType>& rhs)`	Comparison operator ! = for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator==(const constant_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator == for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator==(nullptr_t lhs, const constant_ptr<ElementType>& rhs)`	Comparison operator == for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator>(const constant_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator > for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator>(nullptr_t lhs, const constant_ptr <ElementType>& rhs)`	Comparison operator > for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator<(const constant_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator < for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator<(nullptr_t lhs, const constant_ptr <ElementType>& rhs)`	Comparison operator < for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator>=(const constant_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator >= for `constant_ptr` class with a `nullptr_t`.

Continued on next page

Table 4.29: Non-member functions of the explicit pointer classes.

Non-member functions	Description
`template <typename ElementType> bool operator>=(nullptr_t lhs, const constant_ptr<ElementType>& rhs)`	Comparison operator >= for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator<=(const constant_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator <= for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator<=(nullptr_t lhs, const constant_ptr<ElementType>& rhs)`	Comparison operator <= for `constant_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator==(const local_ptr<ElementType>& lhs, const local_ptr<ElementType>& rhs)`	Comparison operator == for `local_ptr` class.
`template <typename ElementType> bool operator!=(const local_ptr<ElementType>& lhs, const local_ptr<ElementType>& rhs)`	Comparison operator != for `local_ptr` class.
`template <typename ElementType> bool operator<(const local_ptr<ElementType>& lhs , const local_ptr<ElementType>& rhs)`	Comparison operator < for `local_ptr` class.
`template <typename ElementType> bool operator>(const local_ptr<ElementType>& lhs , const local_ptr<ElementType>& rhs)`	Comparison operator > for `local_ptr` class.
`template <typename ElementType> bool operator>=(const local_ptr<ElementType>& lhs, const local_ptr<ElementType>& rhs)`	Comparison operator >= for `local_ptr` class.
`template <typename ElementType> bool operator<=(const local_ptr<ElementType>& lhs, const local_ptr<ElementType>& rhs)`	Comparison operator <= for `local_ptr` class.
`template <typename ElementType> bool operator!=(const local_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator ! = for `local_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator!=(nullptr_t lhs, const local_ptr< ElementType>& rhs)`	Comparison operator ! = for `local_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator==(const local_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator == for `local_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator==(nullptr_t lhs, const local_ptr< ElementType>& rhs)`	Comparison operator == for `local_ptr` class with a `nullptr_t`.

Continued on next page

Table 4.29: Non-member functions of the explicit pointer classes.

Non-member functions	Description

`template <typename ElementType>`
`    bool operator>(const local_ptr<ElementType>& lhs`
`, nullptr_t rhs)`
	Comparison operator > for `local_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator>(nullptr_t lhs, const local_ptr<`
`ElementType>& rhs)`
	Comparison operator > for `local_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator<(const local_ptr<ElementType>& lhs`
`, nullptr_t rhs)`
	Comparison operator < for `local_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator<(nullptr_t lhs, const local_ptr<`
`ElementType>& rhs)`
	Comparison operator < for `local_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator>=(const local_ptr<ElementType>&`
`lhs, nullptr_t rhs)`
	Comparison operator >= for `local_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator>=(nullptr_t lhs, const local_ptr<`
`ElementType>& rhs)`
	Comparison operator >= for `local_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator<=(const local_ptr<ElementType>&`
`lhs, nullptr_t rhs)`
	Comparison operator <= for `local_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator<=(nullptr_t lhs, const local_ptr<`
`ElementType>& rhs)`
	Comparison operator <= for `local_ptr` class with a `nullptr_t`.

`template <typename ElementType>`
`    bool operator==(const private_ptr<ElementType>&`
`lhs,`
`    const private_ptr<ElementType>& rhs)`
	Comparison operator == for `private_ptr` class.

`template <typename ElementType>`
`    bool operator!=(const private_ptr<ElementType>&`
`lhs,`
`    const private_ptr<ElementType>& rhs)`
	Comparison operator != for `private_ptr` class.

`template <typename ElementType>`
`    bool operator<(const private_ptr<ElementType>&`
`lhs,`
`    const private_ptr<ElementType>& rhs)`
	Comparison operator < for `private_ptr` class.

`template <typename ElementType>`
`    bool operator>(const private_ptr<ElementType>&`
`lhs,`
`    const private_ptr<ElementType>& rhs)`
	Comparison operator > for `private_ptr` class.

`template <typename ElementType>`
`    bool operator>=(const private_ptr<ElementType>&`
`lhs,`
`    const private_ptr<ElementType>& rhs)`
	Comparison operator >= for `private_ptr` class.

Continued on next page

Table 4.29: Non-member functions of the explicit pointer classes.

Non-member functions	Description
template <typename ElementType> bool operator<=(const private_ptr<ElementType>& lhs, const private_ptr<ElementType>& rhs)	Comparison operator <= for private_ptr class.
template <typename ElementType> bool operator!=(const private_ptr<ElementType>& lhs, nullptr_t rhs)	Comparison operator ! = for local_ptr class with a nullptr_t.
template <typename ElementType> bool operator!=(nullptr_t lhs, const private_ptr<ElementType>& rhs)	Comparison operator ! = for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator==(const private_ptr<ElementType>& lhs, nullptr_t rhs)	Comparison operator == for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator==(nullptr_t lhs, const private_ptr<ElementType>& rhs)	Comparison operator == for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator>(const private_ptr<ElementType>& lhs, nullptr_t rhs)	Comparison operator >= for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator>(nullptr_t lhs, const private_ptr<ElementType>& rhs)	Comparison operator >= for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator<(const private_ptr<ElementType>& lhs, nullptr_t rhs)	Comparison operator <= for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator<(nullptr_t lhs, const private_ptr<ElementType>& rhs)	Comparison operator < for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator>=(const private_ptr<ElementType>& lhs, nullptr_t rhs)	Comparison operator <= for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator>=(nullptr_t lhs, const private_ptr<ElementType>& rhs)	Comparison operator >= for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator<=(const private_ptr<ElementType>& lhs, nullptr_t rhs)	Comparison operator <= for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator<=(nullptr_t lhs, const private_ptr<ElementType>& rhs)	Comparison operator <= for private_ptr class with a nullptr_t.
template <typename ElementType> bool operator==(const generic_ptr<ElementType>& lhs, const generic_ptr<ElementType>& rhs)	Comparison operator == for generic_ptr class.

Continued on next page

Table 4.29: Non-member functions of the explicit pointer classes.

Non-member functions	Description
`template <typename ElementType>` `bool operator!=(const generic_ptr<ElementType>& lhs,` `const generic_ptr<ElementType>& rhs)`	Comparison operator != for `generic_ptr` class.
`template <typename ElementType>` `bool operator<(const generic_ptr<ElementType>& lhs,` `const generic_ptr<ElementType>& rhs)`	Comparison operator < for `generic_ptr` class.
`template <typename ElementType>` `bool operator>(const generic_ptr<ElementType>& lhs,` `const generic_ptr<ElementType>& rhs)`	Comparison operator > for `generic_ptr` class.
`template <typename ElementType>` `bool operator>=(const generic_ptr<ElementType>& lhs,` `const generic_ptr<ElementType>& rhs)`	Comparison operator >= for `generic_ptr` class.
`template <typename ElementType>` `bool operator<=(const generic_ptr<ElementType>& lhs,` `const generic_ptr<ElementType>& rhs)`	Comparison operator <= for `generic_ptr` class.
`template <typename ElementType>` `bool operator!=(const generic_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator ! = for `local_ptr` class with a `nullptr_t`.
`template <typename ElementType>` `bool operator!=(nullptr_t lhs, const generic_ptr <ElementType>& rhs)`	Comparison operator ! = for `generic_ptr` class with a `nullptr_t`.
`template <typename ElementType>` `bool operator==(const generic_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator == for `generic_ptr` class with a `nullptr_t`.
`template <typename ElementType>` `bool operator==(nullptr_t lhs, const generic_ptr <ElementType>& rhs)`	Comparison operator == for `generic_ptr` class with a `nullptr_t`.
`template <typename ElementType>` `bool operator>(const generic_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator >= for `generic_ptr` class with a `nullptr_t`.
`template <typename ElementType>` `bool operator>(nullptr_t lhs, const generic_ptr< ElementType>& rhs)`	Comparison operator >= for `generic_ptr` class with a `nullptr_t`.
`template <typename ElementType>` `bool operator<(const generic_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator <= for `generic_ptr` class with a `nullptr_t`.
`template <typename ElementType>` `bool operator<(nullptr_t lhs, const generic_ptr< ElementType>& rhs)`	Comparison operator < for `generic_ptr` class with a `nullptr_t`.
`template <typename ElementType>` `bool operator>=(const generic_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator <= for `generic_ptr` class with a `nullptr_t`.
	Continued on next page

Table 4.29: Non-member functions of the explicit pointer classes.

Non-member functions	Description
`template <typename ElementType> bool operator>=(nullptr_t lhs, const generic_ptr <ElementType>& rhs)`	Comparison operator >= for `generic_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator<=(const generic_ptr<ElementType>& lhs, nullptr_t rhs)`	Comparison operator <= for `generic_ptr` class with a `nullptr_t`.
`template <typename ElementType> bool operator<=(nullptr_t lhs, const generic_ptr <ElementType>& rhs)`	Comparison operator <= for `private_ptr` class with a `nullptr_t`.
	End of table

Table 4.29: Non-member functions of the explicit pointer classes.

4.8.1.1 Multi-pointer class

There are situations where a user may want to template a data structure by an address space. Or, a user may want to write templates that adapt to the address space of a pointer. An example might be wrapping a pointer inside a class, where a user may need to template the class according to the address space of the pointer the class is initialized with. In this case, the `multi_ptr` class enables users to do this. In order to facilitate SYCL/OpenCL C interoperability, the `pointer` type is provided. It is an implementation-defined type which corresponds to the underlying OpenCL pointer type and can be used in `extern "C"` function declarations for OpenCL C 1.2 functions used in SYCL kernels or OpenCL C++ kernels.

If OpenCL 2.x devices are used, then the user can mark the data structure as `generic_ptr` and the default generic space will be assigned to all of the pointers of that structure. The default address space of OpenCL 2.x pointers is the generic address space and at runtime the pointers will be allocated on one of the following address spaces: *local, global and private*. That will force disabling any compile-time address-space reduction and usage of that data structure with the constant address space will result in undefined behaviour.

An overview of the interface provided for the `multi_ptr` class is the following, for the full interface please refer to B.2.

```
namespace cl {
namespace sycl {
namespace access {
enum class address_space : int {
  global_space,
  local_space,
  constant_space,
  private_space
};
}  // namespace access

template <typename ElementType, access::address_space Space>
class multi_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ T& reference;
  typedef __undefined__ const T& const_reference;
  typedef __undefined__ T* pointer;
  typedef __undefined__ const T* const_pointer;

  const address_space space;
  constexpr multi_ptr();
  multi_ptr(pointer);
  multi_ptr(const multi_ptr&);
  multi_ptr(multi_ptr&& r);
  constexpr multi_ptr(nullptr_t);
  ~multi_ptr();

  reference operator*();

  // Only if Space == global_space
  operator global_ptr<ElementType>();
  global_ptr<ElementType> pointer();

  // Only if Space == local_space
  operator local_ptr<ElementType>();
  local_ptr<ElementType> pointer();

  // Only if Space == constant_space
  operator constant_ptr<ElementType>();
  constant_ptr<ElementType> pointer();

  // Only if Space == private_space
  operator private_ptr<ElementType>();
  private_ptr<ElementType> pointer();

  // Only if Space == generic
  operator generic_ptr<ElementType>();
  generic_ptr<ElementType> pointer();

  pointer release();
  void reset(pointer p = pointer());
  void swap(multi_ptr& r);

  multi_ptr& operator++();
  multi_ptr operator++(int);
  multi_ptr& operator--();
  multi_ptr operator--(int);
  multi_ptr& operator+=(difference_type r);
  multi_ptr& operator-=(difference_type r);
  multi_ptr operator+(difference_type r);
  multi_ptr operator-(difference_type r);
};

template <typename ElementType, access::address_space Space>
multi_ptr<ElementType, Space> make_ptr(pointer);
}  // namespace sycl
}  // namespace cl
```

Constructors	Description

`template <typename ElementType, enum address_space Space>`
`    explicit multi_ptr(pointer)`
	Constructor that takes as an argument a pointer of ElementType.

`template <typename ElementType, access::address_space Space>`
`    multi_ptr(const multi_ptr &)`
	Copy constructor.

`template <typename ElementType, access::address_space Space>`
`    multi_ptr<ElementType, Space> make_ptr(pointer)`
	Global function to create a `multi_ptr` instance depending on the address space of the `pointer` type.

	End of table

Table 4.30: Constructors for `multi_ptr` class

Methods	Description

`template <typename ElementType, access::address_space Space>`
`    ElementType &operator*()`
	Operator that returns a reference to the ElementType of the multi_ptr class.

`template <typename ElementType, access::address_space Space>`
`    ElementType &operator[](size_t i)`
	Subscript operator.

`template <typename ElementType,`
`    access::address_space Space = access::address_space::global_space>`
`    operator global_ptr<ElementType>()`
	Conversion operator from `multi_ptr <ElementType,address_space::global_space>` to `global_ptr< ElementType>`.

`template <typename ElementType,`
`    access::address_space Space = access::address_space::global_space>`
`    global_ptr<ElementType> pointer()`
	Pointer method that returns a `global_ptr <ElementType>` from a `multi_ptr <ElementType,address_space::global_space>`.

`template <typename ElementType,`
`    access::address_space Space = access::address_space::local_space>`
`    operator local_ptr<ElementType>()`
	Conversion operator from `multi_ptr <ElementType,address_space::local_space>` to `local_ptr<ElementType>`.

`template <typename ElementType,`
`    access::address_space Space = access::address_space::local_space>`
`    local_ptr<ElementType> pointer()`
	Pointer method that returns a `local_ptr <ElementType>` from a `multi_ptr <ElementType,address_space::local_space>`.

`template <typename ElementType,`
`    access::address_space Space = access::address_space::constant_space>`
`    operator constant_ptr<ElementType>()`
	Conversion operator from `multi_ptr <ElementType,address_space::constant_space>` to `constant_ptr< ElementType>`.

`template <typename ElementType,`
`    access::address_space Space = access::address_space::constant_space>`
`    constant_ptr<ElementType> pointer()`
	Pointer method that returns a `constant_ptr <ElementType>` from a `multi_ptr <ElementType,address_space::constant_space>`.

	Continued on next page

Table 4.31: Methods of `multi_ptr` class

Methods	Description
`template <typename ElementType, access::address_space Space = access::address_space::private_space> operator private_ptr<ElementType>()`	Conversion operator from `multi_ptr <ElementType,address_space::private_space>` to `private_ptr< ElementType>`.
`template <typename ElementType, access::address_space Space = access::address_space::private_space> private_ptr<ElementType> pointer()`	Pointer method that returns a `private_ptr <ElementType>` from a `multi_ptr <ElementType,address_space::private_space>`.
`template <typename ElementType, access::address_space Space = access::address_space::generic_space> operator private_ptr<ElementType>()`	Conversion operator from `multi_ptr <ElementType,address_space::generic_space>` to `generic_ptr< ElementType>`.
`template <typename ElementType, access::address_space Space = access::address_space::generic_space> generic_ptr<ElementType> pointer()`	Pointer method that returns a `private_ptr <ElementType>` from a `multi_ptr <ElementType,address_space::generic_space>`.
	End of table

Table 4.31: Methods of `multi_ptr` class

Non-member functions	Description
`template <typename ElementType, access::address_space Space> bool operator==(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator `==` for `multi_ptr` class.
`template <typename ElementType, access::address_space Space> bool operator!=(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator `!=` for `multi_ptr` class.
`template <typename ElementType, access::address_space Space> bool operator<(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator < for `multi_ptr` class.
`template <typename ElementType, access::address_space Space> bool operator>(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator > for `multi_ptr` class.
`template <typename ElementType, access::address_space Space> bool operator<=(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator `<=` for `multi_ptr` class.
	Continued on next page

Table 4.32: Non-member functions of the multi_ptr class.

Non-member functions	Description
`template <typename ElementType, access::address_space Space>` `bool operator>=(const multi_ptr<ElementType, Space>& lhs,` `const multi_ptr<ElementType, Space>& rhs)`	Comparison operator >= for `multi_ptr` class.
`template <typename ElementType, access::address_space Space>` `bool operator!=(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs)`	Comparison operator != for `multi_ptr` class with a `nullptr_t`.
`template <typename ElementType, access::address_space Space>` `bool operator!=(nullptr_t lhs, const multi_ptr< ElementType, Space>& rhs)`	Comparison operator != for `multi_ptr` class with a `nullptr_t`.
`template <typename ElementType, access::address_space Space>` `bool operator==(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs)`	Comparison operator == for `multi_ptr` class with a `nullptr_t`.
`template <typename ElementType, access::address_space Space>` `bool operator==(nullptr_t lhs, const multi_ptr< ElementType, Space>& rhs)`	Comparison operator == for `multi_ptr` class with a `nullptr_t`.
`template <typename ElementType, access::address_space Space>` `bool operator>(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs)`	Comparison operator > for `multi_ptr` class with a `nullptr_t`.
`template <typename ElementType, access::address_space Space>` `bool operator>(nullptr_t lhs, const multi_ptr< ElementType, Space>& rhs)`	Comparison operator > for `multi_ptr` class with a `nullptr_t`.
`template <typename ElementType, access::address_space Space>` `bool operator<(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs)`	Comparison operator < for `multi_ptr` class with a `nullptr_t`.
`template <typename ElementType, access::address_space Space>` `bool operator<(nullptr_t lhs, const multi_ptr< ElementType, Space>& rhs)`	Comparison operator < for `multi_ptr` class with a `nullptr_t`.
`template <typename ElementType, access::address_space Space>` `bool operator>=(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs)`	Comparison operator >= for `multi_ptr` class with a `nullptr_t`.
`template <typename ElementType, access::address_space Space>` `bool operator>=(nullptr_t lhs, const multi_ptr< ElementType, Space>& rhs)`	Comparison operator >= for `multi_ptr` class with a `nullptr_t`.

Continued on next page

Table 4.32: Non-member functions of the multi_ptr class.

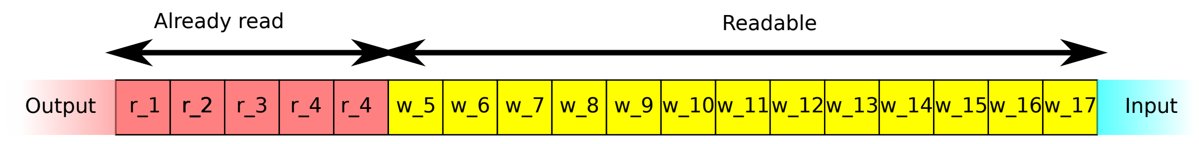

Figure 4.1: Pipe with elements already read and written elements to be read.

Non-member functions	**Description**

`template <typename ElementType, access::address_space Space>`
`bool operator<=(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs)`
	Comparison operator <= for `multi_ptr` class with a `nullptr_t`.

`template <typename ElementType, access::address_space Space>`
`bool operator<=(nullptr_t lhs, const multi_ptr< ElementType, Space>& rhs)`
	Comparison operator <= for `multi_ptr` class with a `nullptr_t`.

	End of table

Table 4.32: Non-member functions of the multi_ptr class.

4.9 Pipes

4.9.1 Overview

Pipes are communication channels between kernels. More conceptually, a pipe memory object is an ordered sequence of data items that stores data organized as a FIFO (first in, first out). A pipe has two endpoints: a write endpoint into which data items are inserted, and a read endpoint from which data items are removed. At any one time, only one kernel instance may write into a pipe, and only one kernel instance may read from a pipe. To support the producer-consumer design pattern, one kernel instance connects to the write endpoint (the producer) while another kernel instance connects to the reading endpoint (the consumer). Pipe data is not accessible from the host.

A pipe object is typed to store objects of a given type and has a capacity to store up to a given amount of objects set at the definition point of the pipe. An abstract view of a simple pipe is given on Figure 4.1.

Since access to pipes are done with read or write methods, but the pipe itself is an ordered storage of object(s), the read or write methods require some kind of mutual exclusion, which may not be possible when using a pipe from different work-items, for example. This is why the concept of reservation in a pipe has been introduced in OpenCL 2.0. A reservation station is a kind of array view that is reserved inside the pipe and each element of this reservation can be accessed independently from the other in a race-free way and in a parallel way. The reservation lasts up to the commit action that sends the written data to the pipe or frees the read reservation, according to whether write or read mode. There can be several reservations on-going in a pipe (typically from different work-groups or subgroups) and intermixed with normal read/write actions on the pipe, as shown of Figure 4.2. There is an auto-commit behaviour in the destructor of the reservation if it was not already committed, to ease the life of the programmer. See Section 4.9.7 for a use case.

Even if the reservation looks like an array view inside the pipe, the order in which a reservation element is read

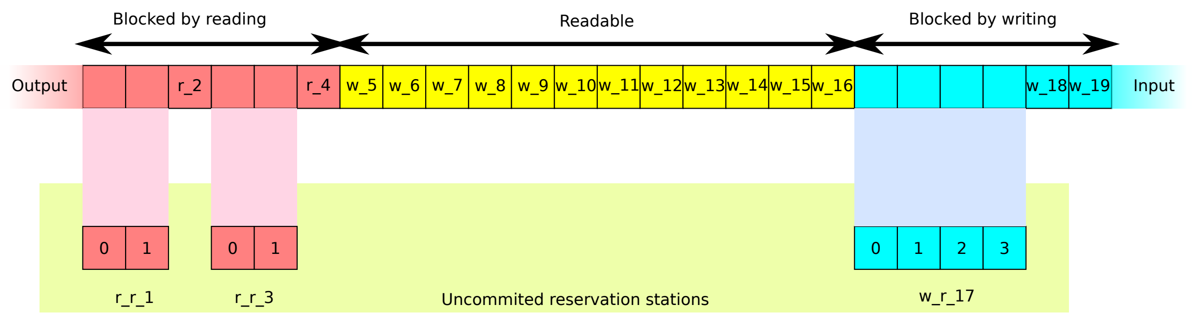

Figure 4.2: Pipe with read and write reservations intermixed with normal read and write operations. The uncommitted r_r_1 and r_r_3 read reservation prevent the elements in red from being reused for writing and the uncommitted w_r_1 write reservation prevents the elements in blue from being read yet.

from or written to the pipe is actually implementation-dependent. Note that committing a write reservation with some elements uninitialized is an undefined behavior.

A pipe application running with a SYCL implementation based on an OpenCL 2.x device should respect the limits retrieved through the device properties `CL_DEVICE_MAX_PIPE_ARGS`, `CL_DEVICE_PIPE_MAX_ACTIVE_RESERVATION` and `CL_DEVICE_PIPE_MAX_PACKET_SIZE`.

Note that pipes rely on the memory and execution model of the underlying OpenCL 2.x device. If a read, write or reserve action on a pipe fails, for example because a consumer has nothing to read, this action may fail forever because the producer may or not be able to run at the same time, since in OpenCL 2.x there is no independent forward progress guarantee in OpenCL 2.x. Furthermore, if an implementation chooses to use plain memory without atomic operations to implement the FIFO data structure of a pipe, the memory model of the OpenCL 2.x implementation is used. Since there is no synchronization of the memory view between the 2 kernels, even if they are running in parallel, a pipe read action may never see a pipe write action from a kernel before the completion of the writing kernel. It is up to the programmer to write kernels in such a way there is no deadlock.

The portable way to have an application using pipes is to create each pipe with a capacity sufficient to store all the data produced by the producer and have no cyclic dependency between kernels interacting through pipes.

This does not preclude some implementations using optimziations to alleviate these constraints.

4.9.2 pipe class

The `pipe` creates a pipe object to allow kernel communications of one kind of objects.

The `pipe` is parameterized with the object type and the maximum number of objects to be stored in the pipe at some point. Pipe constructors are listed in Table 4.33 and methods in Table 4.34.

Constructors	Description
`pipe(std::size_t capacity)`	Construct a pipe able to store up to `capacity` `T` objects.
	End of table

Table 4.33: Pipe constructors.

Methods	Description

`template <access::mode Mode, typename CGH>`
`    accessor<T, 1, Mode, access::pipe>`
`    get_access(CGH &command_group_handler)const`
	Get an accessor to the pipe with the required mode.
	`Mode` is the requested access mode, read or write.
	`command_group_handler` is the command group handler in which the kernel is to be executed.

```
namespace cl {
namespace sycl {
/** A SYCL pipe

    Implement a FIFO-style object that can be used through accessors
    to send some objects T from the input to the output
*/
template <typename T>
class pipe {
 public:
  /// The STL-like type
  using value_type = T;

  /// Construct a pipe able to store up to capacity T objects
  pipe(std::size_t capacity);

  /** Get an accessor to the pipe with the required mode

      \param Mode is the requested access mode

      \param[in] command_group_handler is the command group handler in
      which the kernel is to be executed
  */
  template <access::mode Mode, typename CommandGroupHandler>
  accessor<value_type, 1, Mode, access::pipe>
  get_access(CommandGroupHandler &command_group_handler) const;

  /// Return the maximum number of elements that can fit in the pipe
  std::size_t capacity() const;

};
}  // namespace sycl
}  // namespace cl
```

4.9.3 Pipe accessor class

The `accessor` makes pipe methods available to kernel code. The `accessor` is parameterized with the mode of access, either read or write (see Table 4.11).

Pipe accessor constructors are listed in Table 4.35. The generic methods for the `accessor` class are defined in Table 4.36.

The read methods should only exist for read accessors and the write methods for write accessors.

Implementation advice: after some experiments, to help programmers to detect errors when a method used does not match the access mode, we found it is less confusing to have the methods always defined but controlled with a `static_assert` to display a clear error message to the programmer about why the given method cannot be used or which one should be used instead.

Constructors	Description
`accessor(pipe<value_type> &p, CGH & command_group_handler)const`	Construct a pipe accessor from a pipe `p` using a `command_group_handler` object from the command group scope.
	End of table

Table 4.35: Pipe accessor constructors.

Methods	Description
`pipe_reservation<accessor> reserve(std::size_t size) const`	Make a reservation inside the pipe.
`std::size_t capacity()const`	Return the maximum number of elements that can fit in the pipe.
`std::size_t size()const`	Return the maximum number of elements that can fit in the pipe.
`bool empty()const`	Test if the pipe is empty. This is obviously a volatile value which is constrained by restricted relativity.
`bool full()const`	Test if the pipe is full. This is obviously a volatile value which is constrained by restricted relativity.
`explicit operator bool()const`	In an explicit bool context, the accessor gives the success status of the last access. It is not impacted by reservation success. The explicitness is related to avoid `some_pipe << some_value` to be interpreted as `some_bool << some_value` when the type of `some_value` is not the same type as the pipe type. Return true on success of the previous read or write operation.
`const accessor &write(const value_type &value)const`	Try to write a value to the pipe. `value` is what we want to write. Return `this` so we can apply a sequence of write for example. This function is const so it can work when the accessor is passed by copy in the `[=]` kernel lambda, which is not mutable by default.
`const accessor &operator<<(const value_type &value) const`	Some syntactic sugar to use `a << v` instead of `a.write(v)`.
`const accessor &read(value_type &value)const`	Try to read a value from the pipe. `value` is the reference to where to store what is read. Return `this` so we can apply a sequence of read for example. This function is const so it can work when the accessor is passed by copy in the [=] kernel lambda, which is not mutable by default.
	Continued on next page

Table 4.36: Methods for the pipe `accessor` class.

Methods	Description
`const accessor &operator>>(value_type &value)const`	Some syntactic sugar to use `a >> v` instead of `a.read(v)`.
	End of table

Table 4.36: Methods for the pipe `accessor` class.

```
namespace cl {
namespace sycl {
/** The pipe accessor abstracts the way pipe data are accessed inside
    a kernel

    This is a specialization of the plain accessor.
*/
template <typename T,
          access::mode AccessMode>
class accessor<T, 1, AccessMode, access::pipe> {
public:
  static constexpr auto rank = 1;
  static constexpr auto mode = AccessMode;
  static constexpr auto target = access::pipe;

  /// The STL-like types
  using value_type = T;
  using reference = value_type&;
  using const_reference = const value_type&;

  /** Construct a pipe accessor from a pipe using a command group
      handler object from the command group scope
  */
  accessor(pipe<value_type> &p, handler &command_group_handler);

  // To have the copy and move constructors working
  accessor() = default;

  /// Make a reservation inside the pipe
  pipe_reservation<accessor> reserve(std::size_t size) const;

  /// Return the maximum number of elements that can fit in the pipe
  std::size_t capacity() const;

  /** Get the current number of elements in the pipe

      This is obviously a volatile value which is constrained by
      restricted relativity.
  */
  std::size_t size() const;

  /** Test if the pipe is empty

      This is obviously a volatile value which is constrained by
      restricted relativity.
  */
```

```
    bool empty() const;

    /** Test if the pipe is full

        This is obviously a volatile value which is constrained by
        restricted relativity.
    */
    bool full() const;

    /** In an explicit bool context, the accessor gives the success
        status of the last access

        It is not impacted by reservation success.

        The explicitness is related to avoid \code some_pipe <<
        some_value \endcode to be interpreted as \code some_bool <<
        some_value \endcode when the type of \code some_value \endcode
        is not the same type as the pipe type.

        \return true on success of the previous read or write operation
    */
    explicit operator bool() const;

    /** Try to write a value to the pipe

        \param[in] value is what we want to write

        \return \code this \endcode so we can apply a sequence of write
        for example (but do not do this on a non blocking pipe...)

        \todo provide a && version

        This function is const so it can work when the accessor is
        passed by copy in the [=] kernel lambda, which is not mutable by
        default
    */
    const accessor &write(const value_type &value) const;

    /** Some syntactic sugar to use \code a << v \endcode instead of
        \code a.write(v) \endcode */
    const accessor &operator<<(const value_type &value) const;

    /** Try to read a value from the pipe

        \param[out] value is the reference to where to store what is
        read

        \return \code this \endcode so we can apply a sequence of read
        for example (but do not do this on a non blocking pipe...)

        This function is const so it can work when the accessor is
        passed by copy in the [=] kernel lambda, which is not mutable by
        default
    */
    const accessor &read(value_type &value) const;
```

```

  /// Some syntactic sugar to use a >> v instead of a.read(v)
  const accessor &operator>>(value_type &value) const;

};
}  // namespace sycl
}  // namespace cl
```

4.9.4 pipe_reservation class

The class pipe_reservation represents a reservation made inside a pipe object, either for reading or writing a given amount of objects, according to the accessor used. A successful reservation can be used by different work-item to access to different elements of the reservation in parallel.

A pipe reservation may fail at creation time if there is not enough room to write or not enough things to be read in the pipe. The success status of the pipe_reservation is tested by evaluating it in a boolean context.

A pipe_reservation is expected to be committed after use either by using the explicit commit or more easily by letting the destructor to do it. Only a successful reservation can be committed.

pipe_reservation constructors are listed in Table 4.37. The generic methods for the pipe_reservation class are defined in Table 4.38.

Constructors	Description
`pipe_reservation(accessor_type &accessor, std::size_t s)`	Create a pipe_reservation for an accessor and a number of elements.
	End of table

Table 4.37: Pipe reservation constructors.

Methods	Description
`operator bool()const`	Test if the pipe_reservation has been correctly allocated. Return true if the pipe_reservation can be used and committed.
`std::size_t size()const`	Get the number of reserved element(s).
`reference operator[](std::size_t index)const`	Access to a given element of the reservation.
`void commit()const`	Force a commit operation. Normally the commit is implicitly done in the destructor, but sometime it is useful to do it earlier.
`iterator begin()const`	Get an iterator on the first element of the reservation station.
`iterator end()const`	Get an iterator past the end of the reservation station.
`const_iterator cbegin()const`	Build a constant iterator on the first element of the reservation station.
	Continued on next page

Table 4.38: Methods for the pipe_reservation class.

Methods	Description
`const_iterator cend()const`	Build a constant iterator past the end of the reservation station.
`reverse_iterator rbegin()const`	Get a reverse iterator on the last element of the reservation station.
`reverse_iterator rend()const`	Get a reverse iterator on the first element past the end of the reservation station.
`const_reverse_iterator crbegin()const`	Get a constant reverse iterator on the last element of the reservation station.
`const_reverse_iterator crend()const`	Get a constant reverse iterator on the first element past the end of the reservation station.
	End of table

Table 4.38: Methods for the `pipe_reservation` class.

```
namespace cl {
namespace sycl {
/** The pipe reservation station allows to reserve an array-like view
    inside the pipe for ordered race-free access from various
    work-items for example
*/
template <typename PipeAccessor>
struct pipe_reservation {
  using accessor_type = PipeAccessor;
  /// The STL-like types
  using value_type = typename accessor_type::value_type;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = value_type*;
  using const_pointer = const value_type*;
  using size_type = std::size_t;
  using difference_type = ptrdiff_t;
  using iterator = implementation-defined;
  using const_iterator = implementation-defined;
  using reverse_iterator = implementation-defined;
  using const_reverse_iterator = implementation-defined;

  /// Create a pipe_reservation for an accessor and a number of elements
  pipe_reservation(accessor_type &accessor, std::size_t s);

  /** Use default constructors so that we can create a new buffer copy
      from another one, with either a l-value or a r-value (for
      std::move() for example).
  */
  pipe_reservation() = default;

  /** Test if the pipe_reservation has been correctly allocated

      \return true if the pipe_reservation can be used and committed
  */
  operator bool() const;

  /// Get the number of reserved element(s)
  std::size_t size() const;

  /// Access to a given element of the reservation
  reference operator[](std::size_t index) const;

  /** Force a commit operation

      Normally the commit is implicitly done in the destructor, but
      sometime it is useful to do it earlier.
  */
  void commit() const;

  /// Get an iterator on the first element of the reservation station
  iterator begin() const;

  /// Get an iterator past the end of the reservation station
  iterator end() const;

  /// Build a constant iterator on the first element of the reservation station
  const_iterator cbegin() const;

  /// Build a constant iterator past the end of the reservation station
  const_iterator cend() const;

  /// Get a reverse iterator on the last element of the reservation station
  reverse_iterator rbegin() const;

  /** Get a reverse iterator on the first element past the end of the
      reservation station */
  reverse_iterator rend() const;

  /** Get a constant reverse iterator on the last element of the
      reservation station */
  const_reverse_iterator crbegin() const;

  /** Get a constant reverse iterator on the first element past the
      end of the reservation station */
  const_reverse_iterator crend() const;

};

}  // namespace sycl
}  // namespace cl
```

4.9.5 static_pipe class

A SYCL static-scope pipe is a pipe with a constexpr capacity that can be analysed at either compile-time or run-time to allow an implementation to choose the best underlying implementation of the pipe.

Compared to a normal `pipe`, a `static_pipe` takes a `constexpr` capacity and is expected to be declared and used in a single C++ translation unit so the device compiler can analyse everything at compile time. This will enable a device compiler, or a device optimized runtime to layout the pipe, along with its producer and consumer in an efficient manner for the hardware being targeted.

This pipe object can only be used on the same device. It is the responsibility of the programmer to launch the kernels using this kind of pipe on the same device.

Pipe constructors are listed in Table 4.39 and methods in Table 4.40.

Constructors	Description
`static_pipe()`	Construct a static-scoped pipe able to store up to `Capacity` `T` objects.
	End of table

Table 4.39: Static pipe constructors.

Methods	Description

`template <access::mode Mode, typename CGH>`
`    accessor<T, 1, Mode, access::pipe>`
`    get_access(CGH &command_group_handler)const`
	Get an accessor to the pipe with the required mode.
	`Mode` is the requested access mode, read or write.
	`command_group_handler` is the command group handler in which the kernel is to be executed.

`std::size_t constexpr capacity()const`	Return the maximum number of elements that can fit in the pipe. This is a constexpr since the capacity is in the type.
	End of table

Table 4.40: Methods for the `static_pipe` class.

```
namespace cl {
namespace sycl {
/** A SYCL static-scope pipe is a pipe with a constexpr capacity that
    can be declared in a static context and is expected to be
    equivalent to an OpenCL program-scope pipe

    Implement a FIFO-style object that can be used through accessors
    to send some objects T from the input to the output.

    Compared to a normal pipe, a static_pipe takes a constexpr
    capacity and is expected to be declared in a compile-unit scope or
    class static scope so the device compiler can generate everything
    at compile time.

    This is useful to generate a fixed and optimized hardware
    implementation on FPGA for example, where the interconnection
    graph can be also inferred at compile time.

    This pipe object can only be used on the same device.It is up to
    the responsability of the programmer to launch the kernels using
    this kind of pipe on the same device.
*/
template <typename T, std::size_t Capacity>
struct static_pipe {
  /// The STL-like type
  using value_type = T;

  /// Construct a static-scope pipe able to store up to Capacity T objects
  static_pipe();

  /** Get an accessor to the pipe with the required mode

      \param Mode is the requested access mode, read or write.

      \param[in] command_group_handler is the command group handler in
      which the kernel is to be executed
  */
  template <access::mode Mode, typename CommandGroupHandler>
  accessor<value_type, 1, Mode, access::pipe>
  get_access(CommandGroupHandler &command_group_handler);

  /** Return the maximum number of elements that can fit in the pipe

      This is a constexpr since the capacity is in the type
  */
  std::size_t constexpr capacity() const;

};
}  // namespace sycl
}  // namespace cl
```

4.9.6 Pipe exceptions

4.9.7 Example of using pipes

Pipes are useful to implement data-flow applications with kernels consuming data produced by other kernels. According to the underlying devices, it is possible for example to implement full applications with data flowing through a graph of kernels without any further action from the host besides starting the kernel graph.

4.9.7.1 Producer-consumer example with pipe

In the following example, a producer streams some data from a buffer to a pipe and a consumer kernel reads data from the pipe to add them with another buffer.

```
#include <CL/sycl.hpp>
#include <iostream>
#include <iterator>

constexpr size_t N = 3;
```

```
using Vector = float[N];

int main() {
  Vector a = { 1, 2, 3 };
  Vector b = { 5, 6, 8 };
  Vector c;

  {
    // Create buffers from a & b vectors
    cl::sycl::buffer<float> A { std::begin(a), std::end(a) };
    cl::sycl::buffer<float> B { std::begin(b), std::end(b) };

    // A buffer of N float using the storage of c
    cl::sycl::buffer<float> C { c, N };

    // A pipe of 1 float elements
    cl::sycl::pipe<float> P { 1 };

    // Create a queue to launch the kernels
    cl::sycl::queue q;

    // Launch the producer to stream A to the pipe
    q.submit([&](cl::sycl::handler &cgh) {
      // Get write access to the pipe
      auto p = P.get_access<cl::sycl::access::write>(cgh);
      // Get read access to the data
      auto ka = A.get_access<cl::sycl::access::read>(cgh);

      cgh.single_task<class producer>([=] {
          for (int i = 0; i != N; i++)
            // Try to write to the pipe up to success
            while (!(p << ka[i]))
              ;
        });
      });

    // Launch the consumer that adds the pipe stream with B to C
    q.submit([&](cl::sycl::handler &cgh) {
      // Get read access to the pipe
      auto p = P.get_access<cl::sycl::access::read>(cgh);

      // Get access to the input/output buffers
      auto kb = B.get_access<cl::sycl::access::read>(cgh);
      auto kc = C.get_access<cl::sycl::access::write>(cgh);

      cgh.single_task<class consumer>([=] {
          for (int i = 0; i != N; i++) {
            /* Declare a variable of the same type as what the pipe
               can deal (a good example of single source advantage)
            */
            decltype(p)::value_type e;
            // Try to read from the pipe up to success
            while (!(p >> e))
              ;
            kc[i] = e + kb[i];
          }
        });
      });
  } //< End scope for the queue and the buffers, so wait for completion

  std::cout << std::endl << "Result:" << std::endl;
  for(auto e : c)
    std::cout << e << " ";
  std::cout << std::endl;
}
```

4.9.7.2 Producer-consumer example with pipe reservation

The following example shows a program with a producer and consumer, both using reservation, but one is using explicit commit.

```
#include <CL/sycl.hpp>
#include <iostream>
#include <iterator>
#include <numeric>

// Size of the buffers
constexpr size_t N = 200;
// Number of work-item per work-group
constexpr size_t WI = 20;
static_assert(N == WI*(N/WI), "N needs to be a multiple of WI");

using Type = int;

int main(int argc, char *argv[]) {
  // Initialize the input buffers to some easy-to-compute values
  cl::sycl::buffer<Type> a { N };
  {
    auto aa = a.get_access<cl::sycl::access::write>();
    // Initialize buffer a with increasing integer numbers starting at 0
    std::iota(aa.begin(), aa.end(), 0);
  }

  // A buffer of N Type to get the result
  cl::sycl::buffer<Type> c { N };

  // The plumbing with some weird size prime to WI to exercise the system
  cl::sycl::pipe<Type> pa { 2*WI + 7 };

  // Create a queue to launch the kernels
  cl::sycl::queue q;

  // Launch a producer for streaming va to the pipe pa
  q.submit([&] (cl::sycl::handler &cgh) {
      // Get write access to the pipe
      auto apa = pa.get_access<cl::sycl::access::write>(cgh);
      // Get read access to the data
      auto aa = a.get_access<cl::sycl::access::read>(cgh);
      /* Create a kernel with WI work-items executed by work-groups of
         size WI, that is only 1 work-group of WI work-items */
      cgh.parallel_for_work_group<class producer>(
        { WI, WI },
        [=] (auto group) {
          // Use a sequential loop in the work-group to stream chunks in order
          for (int start = 0; start != N; start += WI) {
            /* To keep the reservation status outside the scope of the
               reservation itself */
            bool ok;
            do {
              // Try to reserve a chunk of WI elements of the pipe for writing
              auto r = apa.reserve(WI);
              // Evaluating the reservation as a bool returns the status
              ok = r;
              if (ok) {
                /* There was enough room for the reservation, then
                   launch the work-items in this work-group to do the
                   writing in parallel */
                group.parallel_for_work_item([=] (cl::sycl::item<> i) {
                    r[i[0]] = aa[start + i[0]];
                  });
              }
              // Here the reservation object goes out of scope: commit
            }
            while (!ok);
          }
        });
    });

  // Launch the consumer to read stream from pipe pa to buffer c
  q.submit([&] (cl::sycl::handler &cgh) {
      // Get read access to the pipe
      auto apa = pa.get_access<cl::sycl::access::read>(cgh);
      // Get write access to the data
      auto ac = c.get_access<cl::sycl::access::write>(cgh);

      /* Create a kernel with WI work-items executed by work-groups of
         size WI, that is only 1 work-group of WI work-items */
      cgh.parallel_for_work_group<class consumer>(
        { WI, WI },
        [=] (auto group) {
          /* Use another approach different from the writing part to
             demonstrate the way to use an explicit commit as proposed
             by Alex Bourd */
          cl::sycl:: pipe_reservation<decltype(apa)> r;
          // Use a sequential loop in the work-group to stream chunks in order
          for (int start = 0; start != N; start += WI) {
            // Wait for the reservation to succeed
            while (!(r = apa.reserve(WI)))
              ;
            /* There was enough room for the reservation, then launch
               the work-items in this work-group to do the reading in
               parallel */
            group.parallel_for_work_item([=] (cl::sycl::item<> i) {
                ac[start + i[0]] = r[i[0]];
              });
            /** Explicit commit requested here. Note that in this
                simple example, since there is nothing useful after
                the commit, using the default destructor at the end of
                the work-group or inside the while would have been
                enough */
            r.commit();
          }
        });
    });

  // Display on the host the buffer content
  for (std::size_t i = 0; i != N; ++i)
    std::cout << c.get_access<cl::sycl::access::read>()[i] << std::endl;

  return 0;
}
```

4.9.7.3 Producer-consumer example with static-scoped pipe

In this example using a `static_pipe`, a direct harware implementation could be statically synthesized on a device, since the size of the pipe is known and also the interconnection of the kernels.

```
#include <CL/sycl.hpp>
#include <iostream>
#include <iterator>

constexpr size_t N = 3;
using Vector = float[N];

// A static-scoped pipe of 4 float elements
cl::sycl::static_pipe<float, 4> p;

int main() {
  Vector va = { 1, 2, 3 };
  Vector vb = { 5, 6, 8 };
  Vector vc;

  {
    // Create buffers from a & b vectors
    cl::sycl::buffer<float> ba { std::begin(va), std::end(va) };
    cl::sycl::buffer<float> bb { std::begin(vb), std::end(vb) };

    // A buffer of N float using the storage of vc
    cl::sycl::buffer<float> bc { vc, N };

    // Create a queue to launch the kernels
    cl::sycl::queue q;

    // Launch the producer to stream A to the pipe
    q.submit([&](cl::sycl::execution_handle &cgh) {
      // Get write access to the pipe
      auto kp = p.get_access<cl::sycl::access::write>(cgh);
      // Get read access to the data
      auto ka = ba.get_access<cl::sycl::access::read>(cgh);

      cgh.single_task<class producer>([=] {
          for (int i = 0; i != N; i++)
            // Try to write to the pipe up to success
            while (!(kp.write(ka[i])))
              ;
        });
      });

    // Launch the consumer that adds the pipe stream with B to C
    q.submit([&](cl::sycl::execution_handle &cgh) {
      // Get read access to the pipe
      auto kp = p.get_access<cl::sycl::access::read>(cgh);

      // Get access to the input/output buffers
      auto kb = bb.get_access<cl::sycl::access::read>(cgh);
      auto kc = bc.get_access<cl::sycl::access::write>(cgh);

      cgh.single_task<class consumer>([=] {
          for (int i = 0; i != N; i++) {
            /* Declare a variable of the same type as what the pipe
               can deal (a good example of single source advantage)
            */
            decltype(kp)::value_type e;
            // Try to read from the pipe up to success
            while (!(kp.read(e)))
              ;
            kc[i] = e + kb[i];
          }
        });
      });
  } //< End scope for the queue and the buffers, so wait for completion

  std::cout << std::endl << "Result:" << std::endl;
  for (auto e : vc)
    std::cout << e << " ";
  std::cout << std::endl;
}
```

4.10 Data types

SYCL as a C++11 programming model supports the C++11 ISO standard data types, and it also provides the ability for all SYCL applications to be executed on SYCL compatible devices, OpenCL and host devices. The scalar and vector data types that are supported by the SYCL system are defined below. More details about the SYCL device compiler support for fundamental and OpenCL interoperability types are found in 9.5

4.10.1 Scalar data types

SYCL follows the C++11 standard in terms of fundamental scalar data types. All SYCL applications match those data types and the size of those has to be matched in the SYCL application code for all available SYCL devices; host and OpenCL devices.

SYCL Integral Data Types	Description
`char`	a signed 8-bit integer, as defined by the C++11 ISO Standard
`unsigned char`	an unsigned 8-bit integer, as defined by the C++11 ISO Standard
`short int`	a signed integer of at least 16-bits, as defined by the C++11 ISO Standard
`unsigned short int`	an unsigned integer of at least 16-bits, as defined by the C++11 ISO Standard
`int`	a signed integer of at least 16-bits, as defined by the C++11 ISO Standard
`unsigned int`	an unsigned integer of at least 16-bits, as defined by the C++11 ISO Standard
`long int`	a signed integer of at least 32-bits, as defined by the C++11 ISO Standard
`unsigned long int`	an unsigned integer of at least 32-bits, as defined by the C++11 ISO Standard
`long long int`	an integer of at least 64-bits, as defined by the C++11 ISO Standard
`unsigned long long int`	an unsigned integer of at least 64-bits, as defined by the C++11 ISO Standard
`size_t`	the unsigned integer type of the result of the `sizeof` operator on host.
	End of table

Table 4.41: SYCL compiler fundamental integral datatypes

SYCL Floating Point Data Types	Description
`float`	a 32-bit IEEE 754 floating-point value, as defined by the C++11 ISO Standard
`double`	a 64-bit IEEE 754 floating-point value, as defined by the C++11 ISO Standard
`half`	a 16-bit IEEE 754-2008 half-precision floating-point value
	End of table

Table 4.42: SYCL compiler fundamental floating point datatypes

The OpenCL C language standardc̃ite[par. 6.11]opencl-1.2 defines its own built-in scalar types, which are supported for interoperability between SYCL and OpenCL C applications through the interoperability data types.

SYCL Scalar Datatypes	Description
`cl::sycl::cl_bool`	A conditional data type which is either true or false. The value true expands to the integer constant 1 and the value false expands to the integer constant 0.
`cl::sycl::cl_char`	a signed two's complement 8-bit integer
`cl::sycl::cl_uchar`	an unsigned 8-bit integer
`cl::sycl::cl_short`	a signed two's complement 16-bit integer
`cl::sycl::cl_ushort`	an unsigned 16-bit integer
`cl::sycl::cl_int`	a signed two's complement 32-bit integer
`cl::sycl::cl_uint`	an unsigned 32-bit integer
`cl::sycl::cl_long`	A signed two's complement 64-bit integer.
`cl::sycl::cl_ulong`	An unsigned 64-bit integer.
`cl::sycl::cl_float`	A 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format.
`cl::sycl::cl_double`	A 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format.
`cl::sycl::cl_half`	A 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format.
	End of table

Table 4.43: SYCL compiler OpenCL interoperability scalar datatypes

4.10.2 Vector types

SYCL provides a templated cross-platform vector type that works efficiently on SYCL devices as well as in host C++ code. This type allows sharing of vectors between the host and its SYCL devices. The vector supports methods that allow construction of a new vector from a swizzeled set of component elements. The vector are defined in Table 4.45 and Table 4.46

An overview of the interface provided for the `vec` class is the following, for the full interface please refer to B.5.

```
namespace cl {
namespace sycl {
template <typename dataT, int numElements>
class vec {
 public:
  typedef dataT element_type;
  //Underlying OpenCL type
  typedef __undefined__ vector_t;

  vec();

  explicit vec(const dataT &arg);

  vec(const T0 &arg0... args);

  vec(const vec<dataT, numElements> &rhs);

  size_t get_count();

  size_t get_size();

  template <typename asDataT, int width>
  vec<asDataT, width> as() const;

  // genvector is a generic typename for describing
  // all OpenCL/SYCL types.
  operator __genvector__() const;

  // Swizzle methods (see notes)
  swizzled_vec<T, out_dims> swizzle<elem s1, ...>();

#ifdef SYCL_SIMPLE_SWIZZLES
  swizzled_vec<T, 4> xyzw();
  ...
#endif  // #ifdef SYCL_SIMPLE_SWIZZLES
};
}  // namespace sycl
}  // namespace cl
```

`vec<typename` T, `int` dims> is a vector type that compiles down to the OpenCL built-in vector types on OpenCL devices where possible and provides compatible support on the host. The `vec` class is templated on its number of dimensions and its element type. The dimensions parameter, `dims`, can be one of: 1, 2, 3, 4, 8 or 16. Any other value should produce a compilation failure. The element type parameter, `T`, must be one of the basic scalar types supported in device code.

The SYCL library provides typedefs for: `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long long`, `unsigned long long`, `float` and `double` in all valid sizes. These vector typedefs are named `TypenameSize`, for example: `longlong2` is a vector of two long long integer elements, mapping to `vec<long long int, 2>`.

The SYCL library also provides the OpenCL interoperability types which are referred to in the document with the generic name `genvector`. The following table 4.44 is showing the the types that are available and are represented for brevity in the specification as a generic and not an actualy SYCL type `genvector`.

Generic type name	Description
genvector	cl::sycl::cl_float2, cl::sycl::cl_float3, cl::sycl::cl_float4, cl::sycl::cl_float8, cl::sycl::cl_float16 cl::sycl::cl_double2, cl::sycl::cl_double3, cl::sycl::cl_double4, cl::sycl::cl_double8, cl::sycl::cl_double16 cl::sycl::cl_char2, cl::sycl::cl_char3, cl::sycl::cl_char4, cl::sycl::cl_char8, cl::sycl::cl_char16 cl::sycl::cl_uchar2, cl::sycl::cl_uchar3, cl::sycl::cl_uchar4, cl::sycl::cl_uchar8, cl::sycl::cl_uchar16 cl::sycl::cl_short2, cl::sycl::cl_short3, cl::sycl::cl_short4, cl::sycl::cl_short8, cl::sycl::cl_short16 cl::sycl::cl_ushort2, cl::sycl::cl_ushort3, cl::sycl::cl_ushort4, cl::sycl::cl_ushort8, cl::sycl::cl_ushort16 cl::sycl::cl_uint2, cl::sycl::cl_uint3, cl::sycl::cl_uint4, cl::sycl::cl_uint8, cl::sycl::cl_uint16 cl::sycl::cl_int2, cl::sycl::cl_int3, cl::sycl::cl_int4, cl::sycl::cl_int8, cl::sycl::cl_int16 cl::sycl::cl_ulong2, cl::sycl::cl_ulong3, cl::sycl::cl_ulong4, cl::sycl::cl_ulong8, cl::sycl::cl_ulong16 cl::sycl::cl_long2, cl::sycl::cl_long3, cl::sycl::cl_long4, cl::sycl::cl_long8, cl::sycl::cl_long16
	End of table

Table 4.44: Generic type name description for genVector, which serves as a description for all valid types of OpenCL/SYCL interoperability vectors.

`swizzled_vec<T, out_dims> vec<T, in_dims>::swizzle<elem s1, elem s2...> ()` returns a temporary object representing a swizzled set of the original vector's member elements. The number of `s1`, `s2` parameters is the same as `out_dims`. All `s1`, `s2` parameters must be integer constants from zero to `in_dims-1`. The swizzled vector may be used as a source (r-value) and destination (l-value). In order to enable the r-value and l-value swizzling to work, this returns an intermediate swizzled-vector class, which can be implicitly converted to a vector (r-value evaluation) or assigned to.

If the user `#defines` the macro `SYCL_SIMPLE_SWIZZLES` before `#include <cl/sycl.hpp>`, then swizzle functions are defined for every combination of swizzles for 2D, 3D and 4D vectors only. The swizzle functions take the form:

```
swizzled_vec<T, out_dims> vec<T, in_dims>::xyzw();
swizzled_vec<T, out_dims> vec<T, in_dims>::rgba();
```

where, as above, the number of `elem::x`, `elem::y`, `elem::z`, `elem::w` or `elem::r`, `elem::g`, `elem::b`, `elem::a` letters is the same as `out_dims`. All `elem::x`, `elem::y`, `elem::z`, `elem::w` or `elem::r`, `elem::g`, `elem::b`, `elem::a` parameters must be letters from the sets first `in_dims` letters in "`xyzw`" or "`rgba`".

Swizzle letters may be repeated or re-ordered. For example, from a vector containins integers [0, 1, 2, 3], `vec.xxzy()` would return a vector of [0, 0, 2, 1].

Constructors	Description
`vec<T, dims>()`	Default construct a vector with element type `T` and with `dims` dimensions by default construction of each of its elements.
`explicit vec<T, dims>(const T &arg)`	Construct a vector of element type `T` and `dims` dimensions by setting each value to arg by assignment.
`vec<T, dims> ( const T &element_0, const T &element_1, . . . , const T &element_dims-1)`	Construct a vector with element type `T` and with `dims` dimensions out of `dims` initial values.
`vec<T, dims>(const &vec<T, dims>)`	Construct a vector of element type `T` and `dims` dimensions by copy from another similar vector.
	End of table

Table 4.45: Constructors for the `vec` class

Methods	Description
`size_t get_count()`	Returns the number of elements of the vector.
`size_t get_size()`	Returns the size of the vector.
`template<typename asDataT, int width> vec<asDataT,width> as()const`	Re-interpret the `vec<dataT,numElements>` to `vec<asDataT,width>` type.
`operator genvector()const`	Converts a `vec<dataT,numElements>` to the corresponding OpenCL vector, generically named openclVector[1] of the same type and width.
`operator vec<dataT,numElements>(genvector clVector) const`	Assignment operator that takes an OpenCL vector instance, which one of *openclVector*, and converts it to the corresponding SYCL `vec<dataT,numElements>` type.
`vec<dataT, numElements> operator+( const vec<dataT, numElements> &rhs)`	Construct vector from the sum of the respective elements of the current vector and rhs.
`vec<dataT, numElements> operator+( const dataT &rhs)`	Construct vector by adding rhs to each element of the current vector.
	Continued on next page

Table 4.46: Methods for the `vec` class

[1]This is not actual SYCL type, the genvector type is described in table 4.44

Methods	Description
`vec<dataT, numElements> operator-( const vec<dataT, numElements> &rhs)`	Construct vector by subtracting the elements of rhs from the respective elements of the current vector.
`vec<dataT, numElements> operator-( const dataT &rhs)`	Construct vector by subtracting rhs from each element of the current vector.
`vec<dataT, numElements> operator*( const vec<dataT, numElements> &rhs)`	Construct vector from the product of the elements of the current vector by the respective elements of rhs.
`vec<dataT, numElements> operator*( const dataT &rhs)`	Construct vector by multiplying each element of the current vector by rhs.
`vec<dataT, numElements> operator/( const vec<dataT, numElements> &rhs)`	Construct vector from the division of the elements of the current vector by the elements of rhs.
`vec<dataT, numElements> operator/( const dataT &rhs)`	Construct vector by dividing each element of the current vector by rhs.
`vec<dataT, numElements> operator%( const vec<dataT, numElements> &rhs)`	Construct vector from the modulo of the elements of the current vector and the elements of rhs.
`vec<dataT, numElements> operator%( const dataT &rhs)`	Construct vector by calculating the remainder of each element of the current vector and rhs.
`vec<dataT, numElements> operator++();`	Prefix increment by one for every element of the vector.
`vec<dataT, numElements> operator++(int);`	Post-fix increment by one for every element of the vector.
`vec<dataT, numElements> operator--();`	Prefix decrement by one for every element of the vector.
`vec<dataT, numElements> operator--(int);`	Post-fix decrement by one for every element of the vector.
`vec<dataT, numElements> operator|( const vec<dataT, numElements> &rhs)`	Construct vector from the bitwise OR of the respective elements of the current vector and rhs.
`vec<dataT, numElements> operator|( const dataT &rhs)`	Construct vector by applying the bitwise OR of each element of the current vector and rhs.
`vec<dataT, numElements> operator^( const vec<dataT, numElements> &rhs)`	Construct vector from the bitwise AND of the respective elements of the current vector and rhs.
`vec<dataT, numElements> operator^( const dataT &rhs)`	Construct vector by applying the bitwise AND of each element of the current vector and rhs.
`vec<dataT, numElements> operator&&( const vec<dataT, numElements> &rhs)`	Construct vector from the logical AND of the respective elements of the current vector and rhs.
`vec<dataT, numElements> operator&&( const dataT &rhs)`	Construct vector by applying the logical AND of each element of the current vector and rhs.
	Continued on next page

Table 4.46: Methods for the vec class

Methods	Description
`vec<dataT, numElements> operator||( const vec<dataT, numElements> &rhs)`	Construct vector from the logical OR of the respective elements of the current vector and rhs.
`vec<dataT, numElements> operator||( const dataT &rhs)`	Construct vector by applying the logical OR of each element of the current vector and rhs.
`vec<dataT, numElements> operator>>( const vec<dataT, numElements> &rhs)`	Construct vector from the outcome of shifting right the respective elements of the current vector by rhs.
`vec<dataT, numElements> operator>>( const dataT &rhs)`	Construct vector by shifting right each element of the current vector by rhs.
`vec<dataT, numElements> operator<<( const vec<dataT, numElements> &rhs)`	Construct vector from the outcome of shifting left the respective elements of the current vector by rhs.
`vec<dataT, numElements> operator<<( const dataT &rhs)`	Construct vector by shifting left each element of the current vector by rhs.
`vec<dataT, numElements> operator~()`	Construct vector from the outcome of applying bitwise not to the respective elements of the current vector.
`vec<dataT, numElements> operator!()`	Construct vector from the outcome of applying logical not to the respective elements of the current vector.
`vec<dataT, numElements> operator+=( const vec<dataT, numElements> &rhs)`	Add each element of rhs to the respective element of the current vector in-place.
`vec<dataT, numElements> operator+=( const dataT &rhs)`	Add rhs in-place to each element of the current vector.
`vec<dataT, numElements> operator-=( const vec<dataT, numElements> &rhs)`	Subtract each element of rhs from the respective element of the current vector in-place.
`vec<dataT, numElements> operator-=( const dataT &rhs)`	Subtract rhs in-place from each element of the current vector.
`vec<dataT, numElements> operator*=( const vec<dataT, numElements> &rhs)`	Multiple each element of the current vector by the respective element of rhs in-place.
`vec<dataT, numElements> operator*=( const dataT &rhs)`	Multiple in-place each element of the current vector by rhs.
`vec<dataT, numElements> operator/=( const vec<dataT, numElements> &rhs)`	Divide each element of the current vector in-place by the respective element of rhs.
`vec<dataT, numElements> operator/=( const dataT &rhs)`	Divide in-place each element of the current vector by rhs.
`vec<dataT, numElements> operator|=( const vec<dataT, numElements> &rhs)`	Bitwise OR of each element of the current vector in-place by the respective element of rhs.
`vec<dataT, numElements> operator|=( const dataT &rhs)`	Bitwise OR in-place each element of the current vector by rhs.
`vec<dataT, numElements> operator^=( const vec<dataT, numElements> &rhs)`	Bitwise and of each element of the current vector in-place by the respective element of rhs.
	Continued on next page

Table 4.46: Methods for the vec class

Methods	Description
`vec<dataT, numElements> operator^=( const dataT &rhs)`	Bitwise AND in-place each element of the current vector by rhs.
`vec<dataT, numElements> operator<<=( const vec<dataT, numElements> &rhs)`	Shift left each element of the current vector in-place by the respective element of rhs.
`vec<dataT, numElements> operator<<=( const dataT &rhs)`	Shift left in-place each element of the current vector by rhs.
`vec<dataT, numElements> operator>>=( const vec<dataT, numElements> &rhs)`	Shift right each element of the current vector in-place by the respective element of rhs.
`vec<dataT, numElements> operator>>=( const dataT &rhs)`	Shift right in-place each element of the current vector by rhs.
`vec<dataT, numElements> operator&=( const vec<dataT, numElements> &rhs)`	Bitwise and of each element of the current vector in-place by the respective element of rhs.
`vec<dataT, numElements> operator&=( const dataT &rhs)`	Bitwise AND in-place each element of the current vector by rhs.
`vec<dataT, numElements> operator%=( const vec<dataT, numElements> &rhs)`	Remainder of each element of the current vector in-place by the respective element of rhs.
`vec<dataT, numElements> operator%=( const dataT &rhs)`	Remainder in-place each element of the current vector by rhs.
`vec<dataT, numElements> &operator=( const vec<dataT, numElements> &rhs)`	Update each element of the current vector with the respective element of rhs and return a reference to the current vector.
`vec<dataT, numElements> &operator=( const dataT &rhs)`	Update each element of the current vector with rhs and return a reference to the current vector.
`bool operator==(const vec<dataT, numElements> &rhs) const`	Return true if all elements of rhs compare equal to the respective element of the current vector.
`bool operator!=(const vec<dataT, numElements> &rhs) const`	Return true if any one element of rhs does not compare equal to the respective element of the current vector.
	End of table

Table 4.46: Methods for the vec class

5. Expressing parallelism through kernels

5.1 Command group

A ***command group*** in SYCL as it is defined in Section 2.3.1 consists of a kernel and all the commands for queued data transfers in order for the kernel's execution to be successful. The SYCL runtime will enqueue all the OpenCL commands that are necessary for a kernel and all its data dependencies to be enqueued on the give queue. The commands that enqueue a kernel and issue all the relevant data transfers for it, form the *command group* and are defined as a C++ functor object.

The command group functor takes as a parameter an instance of the command group `execution_handle` 5.3 class, which encapsulates all the methods executed in the command group scope. This abstraction of the kernel execution unifies the data with its processing and consequently allows more abstraction and flexibility in the parallel programming models that can be implemented on top of SYCL.

The command group functor and the `execution_handle` class serves as an interface for the encapsulation of *command group scope*. For SYCL 2.2 the *execution_handle* class provides access to a broad feature set and is parameterized in order to provide access to different kinds of SVM sharing and OpenCL 2.x capabilities. A command group *handler* class instance is providing the execution capabilities of OpenCL 1.2 devices and features. The existence of a memory model in OpenCL 2.x means that synchronization and atomics may have different behavior depending the version and thus the execution mode is defined by these classes.

A kernel is defined in a command group either as a functor object or as a lambda function. All the device data accesses are defined inside this group and any transfers are managed by the system. The rules for the data transfers regarding device and host data accesses are better described in the data management section (4), where buffers (4.2) and accessor (4.7) classes are described. In the case of shared virtual memory, for coarse grained buffer sharing allocations, buffers using `svm_allocator` instances are used to make sure that all the allocations are happening in the same context and are deallocated correctly. If fine grained virtual address space is available in the system, the command group functor executed has the additional capabilities of either allowing raw pointers allocated in the system being passed to the kernel, if system sharing is supported, or *registering access* to SVM fine grained buffer allocations, in order to make them available to the kernel and tracking dependencies among the kernels, to insure data consistency.

The command group functor has no static state and the commands are explicitly issued by an instance of the `execution_handle` with the *highest* capabilities required for the correct execution of the functor. For example, if a command group functor uses images that need to follow the core capabilities of creating memory objects for OpenCL and SYCL allocations for the host devie, which are compatible with SYCL 1.2 functionality, but is also using system sharing virtual address spaces for buffers, then the `execution_handle` needs to have the *highest* or the most *flexible* capability available, the `svm_fine_grain<svm_sharing::system,svm_atomics::enabled>`.

It is possible to obtain events for the start of the command group functor, the kernel starting, and the command group completing. These events are most useful for profiling, because safe synchronization in SYCL requires synchronization on buffer availability, not on kernel completion. This is due to the fact that the it is not rigidly specified which memory data are stored on kernel completion if buffers are used. The events are provided at the submission of the command group functor at the queue to be executed on. The exception to the above rule is when the pointers used by the kernel are *mapped* to the device or are virtual shared pointer allocations between the host and device. The latter, allow explicit processing of the allocations by the SYCL kernels and the synchronization is happening on the pointers given. Effectively, the `map_allocator` and the `svm_allocator` are making the synchronization of the SYCL buffers and allocations explicit to the pointers give to them.

A command group functor may fail to be enqueued at a queue, or it may fail to execute correctly. A user can therefore supply a secondary queue when submitting a command group to the primary queue. If the SYCL runtime fails to enqueue or execute a command group on a primary queue, it can attempt to run the command group on the secondary queue. The circumstances in which it is, or is not, possible for a SYCL runtime to fall-back from primary to secondary queue are undefined in the specification. One of the most important restrictions for fall-back, is the availability of the requirements of the kernel being accessible from host, e.g. pipes are not accessible from the host and as a result if one of the kernels in a large sequence of kernels that use pipes, fails, there is no way for the SYCL runtime to return back to a consistent state of the pipe objects.

A device-side command group submits a device-side kernel as part of the *nested parallelism* support of OpenCL 2.x systems. The device command group functors do not enqueue memory commands, instead they encapsulate the kernel execution and its enqueue policy as well as the device-side events. The device-side command groups can only be submitted at device queues. An overview of *nested parallelism* is given at 5.5 and the full interface and description in 7.3.

5.2 Ranges and index space identifiers

The data parallelism of the OpenCL execution model and its exposure through SYCL requires instantiation of a parallel execution over a range of iteration space coordinates. To achieve this we expose types to define the range of execution and to identify a given execution instance's point in the iteration space.

To achieve this we expose six types: `range`, `nd_range`, `id`, `item`, `nd_item`, `group` and `sub_group`.

When constructing ids or ranges from integers, the elements are written in row-major format.

5.2.1 `range` class

`range<int dimensions>` is a 1D, 2D or 3D vector that defines the iteration domain of either a single work-group in a parallel dispatch, or the overall dimensions of the dispatch. It can be constructed from integers. Constructors for the range class are described in Table 5.1, methods in Table 5.2 and global operators on ranges in Table 5.3.

An overview of the interface provided for the `range` class is the following, for the full interface please refer toB.3.

```
namespace cl {
namespace sycl {
template <size_t dimensions>
struct range {
  range(const range<dimensions> &);

  range(size_t x);                     // When dimensions==1
  range(size_t x, size_t y);           // When dimensions==2
  range(size_t x, size_t y, size_t z); // When dimensions==3

  size_t get(int dimension) const;
  size_t &operator[](int dimension);

  range &operator=(const range &rhs);
  range &operator+=(const range &rhs);
  range &operator*=(const range &rhs);
  range &operator/=(const range &rhs);
  range &operator%=(const range &rhs);
  range &operator>>=(const range &rhs);
  range &operator<<=(const range &rhs);
  range &operator&=(const range &rhs);
  range &operator^=(const range &rhs);
  range &operator|=(const range &rhs);

  size_t size() const;
};
}  // sycl
}  // cl
```

Constructors	Description
`range(const range<dimensions>&)`	Construct a range by deep copy from another range.
`range(size_t x)`	Construct a 1D range with value x. Only valid when for one dimension.
`range(size_t x, size_t y)`	Construct a 2D range with value x, y. Only valid when `dimensions` is 2.
`range(size_t x, size_t y, size_t z)`	Construct a 3D range with value x, y, z. Only valid when `dimensions` is 3.
	End of table

Table 5.1: Constructors for the `range` class.

Methods	Description
`size_t get(int dimension)const`	Return the value of the specified dimension of the `range`.
`size_t &operator[](int dimension)`	Return the l-value of the specified dimension of the `range`.
`range &operator=(const range &rhs)`	Assign each element of range from its corresponding element of rhs.
`range &operator+=(const range &rhs)`	Elementwise addition of rhs to the current range. Returns reference to updated range.
`range &operator*=(const range &rhs)`	Elementwise multiplication of the current range by rhs. Returns reference to updated range.
`range &operator/=(const range &rhs)`	Elementwise division of the current range by rhs. Returns reference to updated range.
	Continued on next page

Table 5.2: Methods for the `range` class.

Methods	Description
`range &operator%=(const range &rhs)`	Elementwise division of the current range by rhs, updating with the division remainder. Returns reference to updated range.
`range &operator>>=(const range &rhs)`	Elementwise arithmetic right shift of the current range by rhs. Returns reference to updated range.
`range &operator<<=(const range &rhs)`	Elementwise logical left shift of the current range by rhs. Returns reference to updated range.
`range &operator&=(const range &rhs)`	Elementwise bitwise AND of the current range by rhs. Returns reference to updated range.
`range &operator^=(const range &rhs)`	Elementwise bitwise exclusive OR of the current range by rhs. Returns reference to updated range.
`range &operator|=(const range &rhs)`	Elementwise bitwise OR of the current range by rhs. Returns reference to updated range.
`size_t size()const;`	Return the size of the range computed as dimension0*...*dimensionN.
	End of table

Table 5.2: Methods for the range class.

Non-member functions	Description

`template <size_t dimensions>`
`    bool operator ==(`
`    const range<dimensions> &a,`
`    const range<dimensions> &b)`
	Compare two ranges for elementwise equality. If all dimensions are equal, the ranges are equal.

`template <size_t dimensions>`
`    bool operator !=(`
`    const range<dimensions> &a,`
`    const range<dimensions> &b)`
	Compare two ranges for elementwise inequality. If any dimension is not equal, the ranges are not equal.

`template <size_t dimensions>`
`    bool operator >(`
`    const range<dimensions> &a,`
`    const range<dimensions> &b)`
	Compare two ranges such that a is lexically greater than b. Returns true if for any dimension n, $a[n] > b[n]$ and for all $m > n$, $a[m] == b[m]$.

`template <size_t dimensions>`
`    bool operator <(`
`    const range<dimensions> &a,`
`    const range<dimensions> &b)`
	Compare two ranges such that a is lexically less than b. Returns true if for any dimension n, $a[n] < b[n]$ and for all $m > n$, $a[m] == b[m]$.

`template <size_t dimensions>`
`    bool operator >=(`
`    const range<dimensions> &a,`
`    const range<dimensions> &b)`
	Compare two ranges such that a is lexically less than or equal to b. Returns true if $a == b$ or $a > b$.

	Continued on next page

Table 5.3: Non-member functions for the range class.

Non-member functions	Description

`template <size_t dimensions>`
`bool operator <=(`
`const range<dimensions> &a,`
`const range<dimensions> &b)`
	Compare two ranges such that a is lexically less than or equal to b. Returns true if $a == b$ or $a < b$.

`template <int dimensions>`
`range<dimensions> operator *(`
`range<dimensions> a,`
`range<dimensions> b)`
	Multiply each element of `a` by its respective element of `b` and return a range constructed from the resulting values.

`template <int dimensions>`
`range<dimensions> operator /(`
`range<dimensions> dividend,`
`range<dimensions> divisor)`
	Divide each element of `dividend` by its respective element element in `divisor` and return a range constructed of the resulting value.

`template <int dimensions>`
`range<dimensions> operator +(`
`range<dimensions> a,`
`range<dimensions> b)`
	Add each element of `a` to its respective element of `b` and return a range constructed from the resulting values.

`template <int dimensions>`
`range<dimensions> operator -(`
`range<dimensions> a,`
`range<dimensions> b)`
	Subtract each element of `b` from its respective element of `a` and return a range constructed from the resulting values.

`template <size_t dimensions>`
`range<dimensions> operator %(`
`const range<dimensions> &a,`
`const range<dimensions> &b)`
	Divide each element of `b` from its respective element of `a` and return a range constructed from the remainders.

`template <size_t dimensions>`
`range<dimensions> operator <<(`
`const range<dimensions> &a,`
`const range<dimensions> &b)`
	Logically shift each element of `a` left by its matching element of `b` and return a range constructed from the shifted values.

`template <size_t dimensions>`
`range<dimensions> operator >>(`
`const range<dimensions> &a,`
`const range<dimensions> &b)`
	Arithmetically shift each element of `a` right by its matching element of `b` and return a range constructed from the shifted values.

`template <size_t dimensions>`
`range<dimensions> operator &(`
`const range<dimensions> &a,`
`const range<dimensions> &b)`
	Construct a range from the bitwise AND of each element of `a` with the equivalent element of `b`.

`template <size_t dimensions>`
`range<dimensions> operator |(`
`const range<dimensions> &a,`
`const range<dimensions> &b)`
	Construct a range from the bitwise OR of each element of `a` with the equivalent element of `b`.

`template <size_t dimensions>`
`range<dimensions> operator &&(`
`const range<dimensions> &a,`
`const range<dimensions> &b)`
	Construct a range from the logical AND of each element of `a` with the equivalent element of `b`.

Continued on next page

Table 5.3: Non-member functions for the `range` class.

Non-member functions	Description
`template <size_t dimensions> range<dimensions> operator ||( const range<dimensions> &a, const range<dimensions> &b)`	Construct a range from the logical OR of each element of a with the equivalent element of b.
`template <size_t dimensions> range<dimensions> operator ^( const range<dimensions> &a, const range<dimensions> &b)`	Construct a range from the bitwise XOR of each element of a with the equivalent element of b.
`template <size_t dimensions> range<dimensions> operator *( const size_t &a, const range<dimensions> &b)`	Construct a range from the multiplication of each element of b with a.
`template <size_t dimensions> range<dimensions> operator *( const range<dimensions> &a, const size_t &b)`	Construct a range from the multiplication of each element of a with b.
`template <size_t dimensions> range<dimensions> operator /( const size_t &a, const range<dimensions> &b)`	Construct a range from the division of each element of a by b.
`template <size_t dimensions> range<dimensions> operator /( const range<dimensions> &a, const size_t &b)`	Construct a range from the division of each element of a with b.
`template <size_t dimensions> range<dimensions> operator +( const size_t &a, const range<dimensions> &b)`	Construct a range from the addition of each element of b with a.
`template <size_t dimensions> range<dimensions> operator +( const range<dimensions> &a, const size_t &b)`	Construct a range from the addition of each element of a with b.
`template <size_t dimensions> range<dimensions> operator -( const size_t &a, const range<dimensions> &b)`	Construct a range from the subtraction of each element of b with a.
`template <size_t dimensions> range<dimensions> operator -( const range<dimensions> &a, const size_t &b)`	Construct a range from the subtraction of each element of a with b.
`template <size_t dimensions> range<dimensions> operator %( const size_t &a, const range<dimensions> &b)`	Construct a range from the modulo of each element of b with a.

Continued on next page

Table 5.3: Non-member functions for the `range` class.

Non-member functions	Description
`template <size_t dimensions> range<dimensions> operator %( const range<dimensions> &a, const size_t &b)`	Construct a range from the modulo of each element of a with b.
`template <size_t dimensions> range<dimensions> operator <<( const size_t &a, const range<dimensions> &b)`	Construct a range from shiffting left of each element of b with a.
`template <size_t dimensions> range<dimensions> operator <<( const range<dimensions> &a, const size_t &b)`	Construct a range from shiffting left of each element of a with b.
`template <size_t dimensions> range<dimensions> operator >>( const size_t &a, const range<dimensions> &b)`	Construct a range from shiffting right of each element of b with a.
`template <size_t dimensions> range<dimensions> operator >>( const range<dimensions> &a, const size_t &b)`	Construct a range from shiffting right of each element of a with b.
	End of table

Table 5.3: Non-member functions for the range class.

5.2.2 nd_range class

```
namespace cl {
namespace sycl {
template <int dimensions>
struct nd_range {
  nd_range(const nd_range<dimensions> &);

  nd_range(range<dims> globalSize, range<dims> localSize,
           id<dims> offset = id<dims>());

  range<dims> get_global() const;
  range<dims> get_local() const;
  range<dims> get_group() const;
  id<dims> get_offset() const;
};
}  // namespace sycl
}  // namespace cl
```

nd_range<int dimensions> defines the iteration domain of both the work-groups and the overall dispatch. To define this the nd_range comprises two ranges: the whole range over which the kernel is to be executed, and the range of each work group. Constructors for the nd_range class are described in Table 5.4 and methods in Table 5.5.

Constructors	Description
`nd_range(const nd_range<dimensions> &)`	Construct an `nd_range` by deep copy from another `ndrange`.
`nd_range<dimensions>( range<dimensions> globalSize, range<dimensions> localSize) id<dimensions> offset = id<dimensions>())`	Construct an `nd_range` from the local and global constituent ranges as well as an optional offset. If the offset is not provided it will default to no offset.
	End of table

Table 5.4: Constructors for the `nd_range` class.

Methods	Description
`range<dimensions> get_global()const`	Return the constituent global range.
`range<dimensions> get_local()const`	Return the constituent local range.
`range<dimensions> get_group()const`	Return a range representing the number of groups in each dimension. This range would result from `globalSize/localSize` as provided on construction.
`id<dimensions> get_offset()const`	Return the constituent offset.
	End of table

Table 5.5: Methods for the `nd_range` class.

5.2.3 id class

`id<int dimensions>` is a vector of dimensions that is used to represent an *index* into a global or local `range`. It can be used as an index in an accessor of the same rank. The `[n]` operator returns the component `n` as an `size_t`. Constructors for the id class are described in Table 5.6, methods in Table 5.7 and global operators on ids in Table 5.8

An overview of the interface provided for the `id` class is the following, for the full interface please refer to B.4.

```
namespace cl {
namespace sycl {
template <size_t dimensions>
struct id {
  id(size_t x);                        // When dimensions==1
  id(size_t x, size_t y);              // When dimensions==2
  id(size_t x, size_t y , size_t z);  // When dimensions==3
  id(const id<dimensions> & rhs);
  id(const range<dimensions> & rangeSize);
  id(const item<dimensions> & rhs);

  size_t get(int dimension) const;
  size_t &operator[](int dimension);
  operator size_t();          // When dimensions==1

  id &operator=(const id & rhs);
  id &operator+=(const id & rhs);
  id &operator*=(const id & rhs);
  id &operator/=(const id & rhs);
  id &operator%=(const id & rhs);
  id &operator>>=(const id & rhs);
  id &operator<<=(const id & rhs);
  id &operator&=(const id & rhs);
  id &operator^=(const id & rhs);
  id &operator|=(const id & rhs);
};
} // namespace sycl
} // namespace cl
```

Constructors	Description
`id(size_t x)`	Construct a 1D id with value x. Only valid when dimensions is 1.
`id(size_t x, size_t y)`	Construct a 1D id with value x, y. Only valid when dimensions is 2.
`id(size_t x, size_t y, size_t z)`	Construct a 1D id with value x, y, z. Only valid when dimensions is 3.
`id(const id &)`	Construct an id by deep copy.
`id(const range &r)`	Construct an id from the dimensions of a range.
`id(const item &it)`	Construct an id from it.get_global_id().
	End of table

Table 5.6: Constructors for the `id` class.

Methods	Description
`size_t get(int dimension)const`	Return the value of the `id` for dimension `dimension`.
`size_t &operator[](int dimension)const`	Return a reference to the requested dimension of the id object.
`operator size_t()`	Conversion operator so that a `id`<1> can be used as a plain `size_t`.
`id &operator=(const id &rhs)`	Elementwise assignment of id rhs to current id.
`id &operator+=(const id &rhs)`	Elementwise addition of rhs to the current id. Returns reference to updated id.
`id &operator*=(const id &rhs)`	Elementwise multiplication of the current id by rhs. Returns reference to updated id.
`id &operator/=(const id &rhs)`	Elementwise division of the current id by rhs. Returns reference to updated id.
`id &operator%=(const id &rhs)`	Elementwise division of the current id by rhs, updating with the division remainder. Returns reference to updated id.
`id &operator>>=(const id &rhs)`	Elementwise arithmetic right shift of the current id by rhs. Returns reference to updated id.
	Continued on next page

Table 5.7: Methods for the `id` class.

Methods	Description
`id &operator<<=(const id &rhs)`	Elementwise logical left shift of the current id by rhs. Returns reference to updated id.
`id &operator&=(const id &rhs)`	Elementwise bitwise AND of the current id by rhs. Returns reference to updated id.
`id &operatorˆ=(const id &rhs)`	Elementwise bitwise exclusive OR of the current id by rhs. Returns reference to updated id.
`id &operator|=(const id &rhs)`	Elementwise bitwise OR of the current id by rhs. Returns reference to updated id.
	End of table

Table 5.7: Methods for the id class.

Non-member functions	Description
`template <size_t dimensions> bool operator ==( const id<dimensions> &a, const id<dimensions> &b)`	Compare two ids for elementwise equality. If all dimensions are equal, the ids are equal.
`template <size_t dimensions> bool operator !=( const id<dimensions> &a, const id<dimensions> &b)`	Compare two ids for elementwise inequality. If any dimension is not equal, the ids are not equal.
`template <size_t dimensions> bool operator >( const id<dimensions> &a, const id<dimensions> &b)`	Compare two ids such that a is lexically greater than b. Returns true if for any dimension n, $a[n] > b[n]$ and for all $m > n$, $a[m] == b[m]$.
`template <size_t dimensions> bool operator <( const id<dimensions> &a, const id<dimensions> &b)`	Compare two ids such that a is lexically less than b. Returns true if for any dimension n, $a[n] < b[n]$ and for all $m > n$, $a[m] == b[m]$.
`template <size_t dimensions> bool operator >=( const id<dimensions> &a, const id<dimensions> &b)`	Compare two ids such that a is lexically less than or equal to b. Returns true if $a == b$ or $a > b$.
`template <size_t dimensions> bool operator <=( const id<dimensions> &a, const id<dimensions> &b)`	Compare two ids such that a is lexically less than or equal to b. Returns true if $a == b$ or $a < b$.
`template <size_t dimensions> id<dimensions> operator *( const id<dimensions> &a, const id<dimensions> &b)`	Multiply each element of `a` by its respective element of `b` and return an id constructed from the resulting values.
`template <size_t dimensions> id<dimensions> operator /( const id<dimensions> &dividend, const id<dimensions> &divisor)`	Divide each element of `dividend` by its respective element element in `divisor` and return an id constructed of the resulting value.
	Continued on next page

Table 5.8: Non-member functions for the id class.

Non-member functions	Description

`template <size_t dimensions>`
`id<dimensions> operator +(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Add each element of a to its respective element of b and return an id constructed from the resulting values.

`template <size_t dimensions>`
`id<dimensions> operator -(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Subtract each element of b from its respective element of a and return an id constructed from the resulting values.

`template <size_t dimensions>`
`id<dimensions> operator %(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Divide each element of b from its respective element of a and return an id constructed from the remainders.

`template <size_t dimensions>`
`id<dimensions> operator <<(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Logically shift each element of a left by its matching element of b and return an id constructed from the shifted values.

`template <size_t dimensions>`
`id<dimensions> operator >>(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Arithmetically shift each element of a right by its matching element of b and return an id constructed from the shifted values.

`template <size_t dimensions>`
`id<dimensions> operator &(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Construct an id from the bitwise AND of each element of a with the equivalent element of b.

`template <size_t dimensions>`
`id<dimensions> operator |(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Construct an id from the bitwise OR of each element of a with the equivalent element of b.

`template <size_t dimensions>`
`id<dimensions> operator ^(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Construct an id from the bitwise XOR of each element of a with the equivalent element of b.

`template <size_t dimensions>`
`id<dimensions> operator &&(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Construct an id from the logical AND of each element of a with the equivalent element of b.

`template <size_t dimensions>`
`id<dimensions> operator ||(`
`const id<dimensions> &a,`
`const id<dimensions> &b)`
	Construct an id from the logical OR of each element of a with the equivalent element of b.

`template <size_t dimensions>`
`id<dimensions> operator *(`
`const size_t &a,`
`const id<dimensions> &b)`
	Construct an id from the multiplication of a with each element of a.

Continued on next page

Table 5.8: Non-member functions for the id class.

Non-member functions	Description

`template <size_t dimensions>`
`    id<dimensions> operator *(`
`    const id<dimensions> &a,`
`    const size_t &b)`
	Construct an id from the multiplication of each element of a with b.

`template <size_t dimensions>`
`    id<dimensions> operator +(`
`    const size_t &a,`
`    const id<dimensions> &b)`
	Construct an id from the addition of a with each equivalent element of b.

`template <size_t dimensions>`
`    id<dimensions> operator +(`
`    const id<dimensions> &a,`
`    const size_t &b)`
	Construct an id from the addition of each element of a with b.

`template <size_t dimensions>`
`    id<dimensions> operator -(`
`    const size_t &a,`
`    const id<dimensions> &b)`
	Construct an id from the subtraction of a with each equivalent element of b.

`template <size_t dimensions>`
`    id<dimensions> operator -(`
`    const id<dimensions> &a,`
`    const size_t &b)`
	Construct an id from the subtraction of each element of a with b.

`template <size_t dimensions>`
`    id<dimensions> operator /(`
`    const size_t &a,`
`    const id<dimensions> &b)`
	Construct an id from the division of a with each equivalent element of b.

`template <size_t dimensions>`
`    id<dimensions> operator /(`
`    const id<dimensions> &a,`
`    const size_t &b)`
	Construct an id from the division of each element of a with b.

`template <size_t dimensions>`
`    id<dimensions> operator %(`
`    const size_t &a,`
`    const id<dimensions> &b)`
	Construct an id from the modulo of a with each equivalent element of b.

`template <size_t dimensions>`
`    id<dimensions> operator %(`
`    const id<dimensions> &a,`
`    const size_t &b)`
	Construct an id from the modulo of each element of a with b.

`template <size_t dimensions>`
`    id<dimensions> operator <<(`
`    const size_t &a,`
`    const id<dimensions> &b)`
	Construct an id from shifting left a with each equivalent element of b.

`template <size_t dimensions>`
`    id<dimensions> operator <<(`
`    const id<dimensions> &a,`
`    const size_t &b)`
	Construct an id from shifting left each element of a with b.

Continued on next page

Table 5.8: Non-member functions for the id class.

Non-member functions	Description
`template <size_t dimensions> id<dimensions> operator >>( const size_t &a, const id<dimensions> &b)`	Construct an id from shifting right a with each equivalent element of b.
`template <size_t dimensions> id<dimensions> operator >>( const id<dimensions> &a, const size_t &b)`	Construct an id from shifting right each element of a with b.
	End of table

Table 5.8: Non-member functions for the id class.

5.2.4 item class

item<int dimensions identifies an instance of the functor executing at each point in a range<> passed to a parallel_for call, or passed to a parallel_for_work_item call. It encapsulates enough information to identify the work-item's global or local ID, the range of possible values, and the offset of the range, if provided to the parallel_for. Instances of the item<> class are not user-constructible and are passed by the runtime to each instance of the functor. Methods for the item<> class are described in Table 5.9.

```
namespace cl {
namespace sycl {
template <int dimensions>
struct item {
  item() = delete;

  item(const item &rhs) = default;

  item(id &id);

  id<dimensions> get() const;

  size_t get(int dimension) const;

  size_t &operator[](int dimension);

  range<dimensions> get_range() const;

  id<dimensions> get_offset() const;

  size_t get_linear_id() const;

  range<dimensions> get_enqueued_local_range() const;

  size_t get_enqueued_local_range(int dimension) const;

  group<dimensions> get_group() const;

  size_t get_group(int) const;

  size_t get_group_linear_id() const;

  id<dimensions> get_num_groups() const;

  size_t get_num_groups(int) const;

  sub_group get_sub_group() const;

  size_t get_num_sub_groups() const;

  size_t get_max_sub_group_size() const;

  size_t get_enqueued_num_sub_groups() const;a



};
}  // namespace sycl
}  // namespace cl
```

Methods	Description
`id<dimensions> get()const`	Return the constituent local or global `id<>` representing the work-item's position in the iteration space.
`size_t get(int dimension)const`	Return the requested dimension of the constituent `id<>` representing the work-item's position in the iteration space.
`size_t &operator[](int dimension)`	Return the constituent `id<>` l-value representing the work-item's position in the iteration space in the given `dimension`.
`range<dimensions> get_range()const`	Returns a `range<>` representing the dimensions of the range of possible values of the `item`.
`id<dimensions> get_offset()const`	Returns an `id<>` representing the *n*-dimensional offset provided to the `parallel_for` and that is added by the runtime to the global-ID of each work-item, if this item represents a global range. For an item representing a local range of where no offset was passed this will always return an `id` of all 0 values.
`size_t get_linear_id()const`	Return the linearized ID in the `item`'s range. Computed as the flatted ID after the offset is subtracted.
`group<dimensions> get_group()const`	Return the constituent group `group` representing the work-group's position within the overall `nd_range`.

Continued on next page

Table 5.9: Methods for the `item` class.

Methods	Description
`size_t get_group(int dimension)const`	Return the constituent element of the group `id` representing the work-group's position within the overall `nd_range` in the given `dimension`.
`size_t get_group_linear_id()const`	Return the flattened id of the current work-group.
`id<dimensions> get_num_groups()const`	Returns the number of groups in the `nd_range`.
`size_t get_num_groups(int dimension)const`	Return the number of groups for `dimension` in the `nd_range`.
`sub_group<dimensions> get_sub_group()const`	Return the sub_group for the given kernel.
`size_t get_num_sub_groups()const`	Return the number of subgroups per dimension.
`range<dimensions> get_global_range()const`	Returns a `range<>` representing the dimensions of the `nd_range<>`
`range<dimensions> get_local_range()const`	Returns a `range<>` representing the dimensions of the current work-group.
`id<dimensions> get_offset()const`	Returns an `id<>` representing the n-dimensional offset provided to the constructor of the `nd_range<>` and that is added by the runtime to the global-ID of each work-item.
	End of table

Table 5.9: Methods for the `item` class.

5.2.5 `nd_item` class

`nd_item<int dimensions, opencl22>` identifies an instance of the functor executing at each point in an `nd_range<int dimensions>` passed to a `parallel_for_ndrange` call. It encapsulates enough information to identify the work-item's local and global IDs, the work-groups ID and also provides barrier functionality to synchronize work-items. Instances of the `nd_item<int dimensions>` class are not user-constructible and are passed by the runtime to each instance of the functor. Methods for the `nd_item<int dimensions>` class are described in Table 5.10.

The SYCL 2.2 version of the `nd_item<>` class has also the capability of retrieving the sub_group information from the kernel.

```cpp
namespace cl {
namespace sycl {
template <int dimensions>
struct nd_item {
  nd_item() = delete;

  nd_item(const nd_item &) = default;

  id<dimensions> get_global() const;

  size_t get_global(int) const;

  size_t get_global_linear_id() const;

  id<dimensions> get_local() const;

  size_t get_local(int) const;

  size_t get_local_linear_id() const;

  group<dimensions> get_group() const;

  size_t get_group(int) const;

  size_t get_group_linear_id() const;

  id<dimensions> get_num_groups() const;

  size_t get_num_groups(int) const;

  sub_group get_sub_group() const;

  size_t get_num_sub_groups() const;

  size_t get_max_sub_group_size() const;

  size_t get_enqueued_num_sub_groups() const;a

  range<dimensions> get_global_range() const;

  range<dimensions> get_local_range() const;

  id<dimensions> get_offset() const;

  nd_range<dimensions> get_nd_range() const;

  range<dimensions> get_enqueued_local_range() const;

  size_t get_enqueued_local_range(int dimension) const;

  void barrier(access::fence_space flag = access::fence_space::global_and_local) const;

  void barrier(group, access::fence_space flag, access::memory_scope = access::memory_scope::
      work_group ) const;

  void barrier(sub_group, access::fence_space flag, access::memory_scope = access::memory_scope::
      sub_group ) const;
};
}  // namespace sycl
}  // namespace cl
```

Methods	Description
size_t get_work_dim	Returns the number of dimensions in use.

Continued on next page

Table 5.10: Methods for the nd_item class.

Methods	Description
`id<dimensions> get_global()const`	Return the constituent global `id` representing the work-item's position in the global iteration space.
`size_t get_global(int dimension)const`	Return the constituent element of the global `id` representing the work-item's position in the global iteration space in the given `dimension`.
`size_t get_global_linear_id()const`	Return the flattened id of the current work-item after subtracting the offset.
`id<dimensions> get_local()const`	Return the constituent local `id` representing the work-item's position within the current work-group.
`size_t get_local(int dimension)const`	Return the constituent element of the local `id` representing the work-item's position within the current work-group in the given `dimension`.
`size_t get_local_linear_id()const`	Return the flattened id of the current work-item within the current work-group.
`group<dimensions> get_group()const`	Return the constituent group `group` representing the work-group's position within the overall `nd_range`.
`size_t get_group(int dimension)const`	Return the constituent element of the group `id` representing the work-group's position within the overall `nd_range` in the given `dimension`.
`size_t get_group_linear_id()const`	Return the flattened id of the current work-group.
`id<dimensions> get_num_groups()cons`	Returns the number of groups in the `nd_range`.
`size_t get_num_groups(int dimension)const`	Return the number of groups for `dimension` in the `nd_range`.
`sub_group<dimensions> get_sub_group()const`	Return the sub_group for the given kernel.
`size_t get_num_sub_groups()const`	Return the number of subgroups per dimension
`size_t get_max_sub_group_size()const`	Returns the maximum size of a subgroup within the dispatch. This value will be invariant for a given set of dispatch dimensions and a kernel object compiled for a given device.
`range<dimensions> get_global_range()const`	Returns a `range` representing the dimensions of the `nd_range`.
`range<dimensions> get_local_range()const`	Returns a range representing the dimensions of the current work-group.

Continued on next page

Table 5.10: Methods for the `nd_item` class.

Methods	Description
`id<dimensions> get_offset()const`	Returns an id<> representing the n-dimensional offset provided to the constructor of the nd_range<> and that is added by the runtime to the global-ID of each work-item.
`nd_range<dimensions> get_nd_range()const`	Returns the nd_range<> of the current execution.
`void barrier( access::fence_space flag= access::fence_space::global_and_local)const`	Executes a barrier with memory ordering on the local address space, global address space or both based on the value of flag. The current work-item will wait at the barrier until all work-items in the current work-group have reached the barrier. In addition the barrier performs a fence operation ensuring that all memory accesses in the specified address space issued before the barrier complete before those issued after the barrier.
	End of table

Table 5.10: Methods for the nd_item class.

5.2.6 group class

The group<int dimensions> is passed to each instance of the functor execution a parallel_for_work_group in a hierarchical parallel execution. The group encapsulates all functionality required to represent a particular group within a parallel execution. It is not user-constructable. Methods for the group<> class are described in Table 5.11.

The local range stored in the group class will be provided either by the programmer, when it is passed as an optional parameter to parallel_for_work_group, or by the runtime system when it selects the optimal work-group size. This allows the developer to always know how many concurrent work-items are active in each executing work-group, even through the abstracted dimensions of the parallel_for_work_item loops.

```
namespace cl {
namespace sycl {
  enum class work_group_op: unsigned int { add, min, max };


template <int dimensions>
struct group {
  group(const group &rhs) = default;

  id<dimensions> get() const;

  size_t get(int dimension) const;

  range<dimensions> get_global_range() const;

  size_t get_global_range(int) const;

  range<dimensions> get_group_range() const;

  size_t get_group_range(int) const;

  size_t operator[](int) const;

  size_t get_linear() const;

  cl_bool all(cl_bool);

  cl_bool any(cl_bool);

  template<typename T, int dimensions>
  T broadcast(T a, id<dimensions> local_id);

  template<typename T, work_group_op Op>
  T reduce(T x);

  template<typename T, work_group_op Op>
  T scan_exclusive(T x);

  template<typename T, work_group_op Op>
  T scan_inclusive(T x);

};
}  // sycl
}  // cl
```

Methods	Description
`id<dimensions> get()`	Return an `id` representing the index of the group within the `nd_range` for every dimension.
`size_t get(int dimension)const`	Return the index of the group in the given `dimension`.
`range<dimensions> get_global_range()`	Return the constituent global range.
`size_t get_global_range(int dimension)`	Return element `dimension` from the constituent global range.
`range<dimensions> get_group_range()`	Return a `range<>` representing the dimensions of the current group. This local range may have been provided by the programmer, or chosen by the runtime.
`size_t get_group_range(int dimension)`	Return element `dimension` from the constituent group range.
`size_t operator[](int dimension)const`	Return the index of the group in the given `dimension` within the `nd_range<>`.
`size_t get_linear()const`	Get a linearized version of the group ID.
`cl_bool all(cl_bool predicatel)`	Evaluates predicate for all work-items in the work-group and returns true if predicate evaluates to true for all work-items in the work-group.

Continued on next page

Table 5.11: Methods for the `group` class.

Methods	Description
`cl_bool any(cl_bool predicate)`	Evaluates predicate for all work-items in the work-group and returns true if predicate evaluates to true for any work-items in the work-group.

`template<typename T, int dimensions>`
`    T broadcast(T a, id<dimensions> local_id)`
	Broadcast the value of a for work-item identified by local_id to all work-items in the work-group. local_id must be the same value for all work-items in the work-group.

`template<typename T, work_group_op Op>`
`    T reduce(T x)`
	Return result of reduction operation specified by *Op* for all values of x specified by work-items in a work-group.

`template<typename T, work_group_op Op>`
`    T scan_exclusive(T x)`
	Do an exclusive scan operation specified by <op> of all values specified by work-items in the work-group. The scan results are returned for each work-item.
	The scan order is defined by increasing 1D linear global ID within the work-group.

`template<typename T, work_group_op Op>`
`    T scan_inclusive(T x)`
	Do an inclusive scan operation specified by *Op* of all values specified by work-items in the work-group. The scan results are returned for each work-item.
	The scan order is defined by increasing 1D linear global ID within the work-group.

	End of table

Table 5.11: Methods for the `group` class.

5.2.7 sub_group class

The `sub_group` is a vector of work-items in a nd_range parallel execution of a kernel. It is passed to each instance of the functor execution a `parallel_for_sub_group` in a hierarchical parallel execution. The sub_group encapsulates all functionality required to represent a particular sub_group within a parallel execution. It is not user-constructable. Methods for the `sub_group<>` class are described in Table5.12.

The number of sub-groups are defined by the template parameter provided to the `parallel_for_work_group` or the `parallel_for` variants that take an item or `nd_item`.

```
namespace cl {
namespace sycl {
struct sub_group {
  sub_group(const sub_group &rhs) = default;

  id<1> get() const;

  id<1> get_local() const;

  range<1> get_range() const;

  cl_bool all(cl_bool);

  cl_bool any(cl_bool);

  template<typename T>
  T broadcast(T a, id<1> local_id);

  template<typename T, work_group_op Operand>
  T reduce(T x);

  template<typename T, work_group_op Operand>
  T scan_exclusive(T x);

  template<typename T, work_group_op Operand>
  T scan_inclusive(T x);


};
}  // sycl
}  // cl
```

Methods	Description
`id<1> get()`	Returns the sub-group id, which can be from 0 to the number of groups.
`id<1> get_local()`	Returns the unique work-item id within the current sub-group. The mapping of the local id of the work group to the id of the sub_group will be invariant for the lifetime of the workgroup.
`range<1> get_range()`	Returns the number of work-items in the sub-group. This value is no more than the maximum sub-group size and is implementation-defined based on a combination of the compiled kernel and the dispatch dimensions. This will be a constant value for the lifetime of the sub-group.
`cl_bool all(cl_bool predicate)`	Evaluates predicate for all work-items in the sub-group and returns a non-zero value if predicate evaluates to non-zero for all work-items in the sub-group.
`cl_bool any(cl_bool predicate)`	Evaluates predicate for all work-items in the sub-group and returns a non-zero value if predicate evaluates to non-zero for any work-items in the sub-group.

`template<typename T>`
`T broadcast(T x, id<1> local_id)`
	Broadcast the value of x for work-item identified by `sub_group_local_id` (value returned by `get_sub_group_local_id`) to all work-items in the sub-group. `sub_group_local_id` must be the same value for all work-items in the sub-group.

Continued on next page

Table 5.12: Methods for the `sub_group` class.

Methods	Description

`template<typename T, work_group_op Op>`
`    T reduce(T x)`
	Return result of reduction operation specified by *Op* for all values of x specified by work-items in a sub-group.

`template<typename T, work_group_op Op>`
`    T scan_exclusive(T x)`
	Do an exclusive scan operation specified by *Op* of all values specified by work-items in a sub-group. The scan results are returned for each work-item.
	The scan order is defined by increasing 1D linear global id within the sub-group.

`template<typename T, work_group_op Op>`
`    T scan_inclusive(T x)`
	Do an inclusive scan operation specified by *Op* of all values specified by work-items in a sub-group. The scan results are returned for each work-item.
	The scan order is defined by increasing 1D linear global ID within the *sub-group*.
	The inclusive scan operation takes a binary operator *Op* with an identity I and n (where n is the size of the work-group) elements `[a0, a1, ... an-1]` and returns `[a0, (a0 Op a1), ... (a0 Op a1 Op ... Op an-1)]`. *Op* can be either add, min or max, the results for each case are:
	• If *Op* = *add*, the identity matrix *I* is 0.
	• If *Op* = *min*, the identity matrix *I* is
	  – INT_MAX, for int
	  – UINT_MAX, for unsigned int
	  – LONG_MAX, for long long int
	  – ULONG_MAX, for unsigned long long int
	  – +INF, for floating point types.
	• If *Op* = *max*, the identity matrix *I* is
	  – INT_MIN, for int
	  – 0, for unsigned int
	  – LONG_MIN, for long long int
	  – 0, for unsigned long long int
	  – -INF, for floating point types.

	End of table

Table 5.12: Methods for the `sub_group` class.

5.3 `execution_handle` class

A SYCL command group encapsulates all the commands that will be enqueueing work on a SYCL device. SYCL provides a variety of OpenCL platforms and allows the usage of all core and optional features from OpenCL 1.2 to OpenCL 2.x. The developer can exploit all the different features provided in a seamless modern C++ environment by utilizing the corresponding execution mode handler for the feature set they want to exploit per command group.

In this way, the SYCL device compiler and runtime can have all the necessary information in order to enqueue correctly the kernel and any memory reads and writes but also provide diagnostics for mismatching features.

The execution capabilities of the command group are exposed through the `execution_handle` class. The parameterization of this class makes it possible to overide the default behaviour of the command group functor, which for SYCL 2.2 cover all the core features for SYCL 2.2 and targets core functionality of the OpenCL 2.2 runtime. This class ensures backwards compatibility with SYCL 1.2, as the `handler` class is not an alias to `execution_handle< opencl12>`. With this change there is no source or behavioural changes to the existing SYCL 1.2 applications.

The `execution_handle` class is passed by reference as an argument to the lambda or functor object of the submit function, and can only be constructed by the SYCL runtime. The `execution_handle` object is used for memory management but also for invoking kernels on the host or the device. An instance is given to accessors in order to provide access to non-shared memory allocations. Alternatively, it sets arguments for a pre-compiled OpenCL kernel. Finally, an of `execution_handle` is able to register access to a shared virtual memory pointer.

The `parallel_for` functions described in5.4.1,5.4.2 are providing the interface of the kernel invocations in the different execution modes.

`execution_handle` modes supported by the system	**Description**
`using handler = execution_handle<opencl12>;`	The command group `handler` type alias, supports OpenCL 1.2 execution model and the compilation flags for the kernels are going to be cl=1.2. All the OpenCL 1.2 features, can be used with the higher version, however, due to differences to the memory models applied between OpenCL 1.2 and OpenCL 2.2, the behavior of synchronization mechanisms may differ. In the case of using the type `opencl12` and features that are only available in the SYCL 2.2 specification, then errors may occur due to incompatibility with earlier versions of OpenCL devices.
`execution_handle<opencl22>`	The command group `execution_handle` class supports the OpenCL 2.2 core functionality supported by SYCL 2.2. The default execution mode is equivalent to `execution_handler<opencl22>`. In this case all OpenCL 2.x atomics are allowed and the memory model is following the OpenCL 2.0 memory model. In this case, the hierarchical parallel for has three nested levels, work-groups, sub-groups and work-items. In this command group execution mode, it is allowed to use coarse-grain SVM (Shared Virtual Memory) and it also compatible with all SYCL synchronization rules. OpenCL 2.2 atomics are also allowed in this mode.

`execution_handle` **modes supported by the system**	**Description**
`using exec_coarse = execution_handle< svm_coarse_grain>;`	In this command group execution mode, the usage of coarse grain buffer sharing of the shared virtual memory feature of OpenCL 2.0 platforms is defined. The user can use SYCL buffer/accessor mechanism for memory storage and management in a shared virtual memory context. In this mode, interoperability with different SYCL devices and systems is ensured in the same manner as SYCL buffers and images. The synchronization methods apply as in the SYCL 2.2 memory model. In this case the buffer constructor needs to used an `svm_allocator` instance with the context were the allocations will be done and used. Please note that all SVM allocations need to be created inside the SVM context.
`using exec_fg_buffer = execution_handle< svm_fine_grain<svm_sharing::buffer,svm_atomics::none >>;`	This command group execution mode is only available on devices with fine grain sharing capability in buffer scope. There are no SVM atomics available for synchronization for this mode. All the pointers that are allocated using the `svm_allocator`. All svm memory allocations need to be *registered* for access in the command group so that they can be used within a SYCL kernel. This method is `register_access(T*)` and is available only for fine grained SVM mode. Please note that all SVM allocations need to be in the same context and any non-SVM allocations still need to use the accessor objects within the kernel.
`using exec_fga_buffer = execution_handle< svm_fine_grain<svm_sharing::buffer,svm_atomics:: supported>>;`	This command group execution mode is only available on devices with fine grain sharing capability in buffer scope and atomics enabled. There are SVM atomics available for synchronization for this mode. All the pointers that are allocated using the `svm_allocator`. All svm memory allocations need to be *registered* for access in the command group so that they can be used within a SYCL kernel. This method is `register_access< access::mode>(T*)` and is available only for fine grained SVM mode. Please note that all SVM allocations need to be in the same context and any non-SVM allocations still need to use the accessor objects within the kernel.

execution_handle modes supported by the system	Description
`using exec_fg_system = execution_handle< svm_fine_grain<svm_sharing::system,svm_atomics::none >>;`	This command group execution mode is only available on devices with fine grain sharing capability in host and device memory scope. There are no SVM atomics available for synchronization for this mode. All the pointers allocated in the system using any allocation method valid for the whole of the host's virtual memory can be used inside the command group.
`using exec_fga_system = execution_handle< svm_fine_grain<svm_sharing::system,svm_atomics:: supported>>;`	This command group execution mode is only available on devices with fine grain sharing capability in host and device memory scope. There are SVM atomics available for synchronization for this mode. All the pointers allocated in the system using any allocation for the whole of the host's virtual memory, can be used inside the command group.
`device_handle`	The `device_handle` is used in the case of enqueueing kernels from device kernels using device_queue. This is a separate class that is constructable on device-only and is providing device-side scheduling functionality and access.

```cpp
namespace cl {
namespace sycl {

enum class svm_sharing : int { buffer, system };
enum class svm_atomics : int { none, supported };

struct opencl12;
struct opencl22;
struct svm_coarse_grain;
template<svm_sharing S=svm_sharing::buffer, svm_atomics A=svm_atomics::none>
struct svm_fine_grain;


class handler_event {
 public:
   event get_kernel() const;
   event get_complete() const;
   event get_end() const;
}
/* Default execution capabilities are the core
 * capabilities for OpenCL 2.x systems and no svm,
 * which is defined by using the type opencl22.
 */
template <typename E= opencl22>
class execution_handle {
 private:
   // implementation defined constructor
   execution_handle(___unespecified___);

 public:
   execution_handler(const execution_handler& rhs);

   void set_arg(int arg_index, accessor acc_obj);

   template <typename T>
   void set_arg(int arg_index, T scalar_value);

   // In the case of a functor with a globally visible name
   // the template parameter:"typename kernelName" can be ommitted
   // and the kernelType can be used instead.
   template <typename KernelName, class KernelType>
   void single_task(KernelType);

   // In the case of a functor with a globally visible name
   // the template parameter:"typename kernelName" can be ommitted
   // and the kernelType can be used instead.
   template <typename KernelName, class KernelType>
   void parallel_for(range<dimensions> numWorkItems, KernelType);

   // In the case of a functor with a globally visible name
   // the template parameter:"typename kernelName" can be ommitted
   // and the kernelType can be used instead.
   template <typename KernelName, class KernelType>
   void parallel_for(range<dimensions> numWorkItems,
                     id<dimensions> workItemOffset, KernelType);

   // In the case of a functor with a globally visible name
   // the template parameter:"typename kernelName" can be ommitted
   // and the kernelType can be used instead.
   template <typename KernelName, class KernelType>
   void parallel_for(nd_range<dimensions> executionRange, KernelType);

   // In the case of a functor with a globally visible name
   // the template parameter:"typename kernelName" can be ommitted
   // and the kernelType can be used instead.
   template <typename KernelName, class KernelType>
   void parallel_for(nd_range<dimensions> numWorkItems,
                     id<dimensions> workItemOffset, KernelType);

   // In the case of a functor with a globally visible name
   // the template parameter:"typename kernelName" can be ommitted
   // and the kernelType can be used instead.
   template <class KernelName, class WorkgroupFunctionType>
   void parallel_for_work_group(range<dimensions> numWorkGroups,
                                WorkgroupFunctionType);

   // In the case of a functor with a globally visible name
   // the template parameter:"typename kernelName" can be ommitted
   // and the kernelType can be used instead.
   template <class KernelName, class WorkgroupFunctionType>
   void parallel_for_work_group(range<dimensions> numWorkGroups,
                                range<dimensions> workGroupSize,
                                WorkgroupFunctionType);

   void single_task(kernel syclKernel);

   void parallel_for(range<dimensions> numWorkItems, kernel syclKernel);

   void parallel_for(nd_range<dimensions> ndRange, kernel syclKernel);


  template <typename T, access::mode AccessMode, std::enable_if<(
       std::is_same(E,svm_fine_grain), T>::type * = nullptr>
   void register_access(T *);

 };

 }  // namespace sycl
 }  // namespace cl
```

Constructors	Description

`template<typename ExecCap=opencl22core>`
`    execution_handler(const execution_handler &rhs)`
	Copy constructor of a command group *execution_handler*.

	End of table

Table 5.14: Constructors for the command group `execution_handler` class

Methods	Description
`void set_arg(int index, accessor & accObj)`	Set kernel args for an OpenCL kernel which is used through the SYCL/OpenCL interop interface. The index value specifies which parameter of the OpenCL kernel is being set and the accessor object, which OpenCL buffer or image is going to be given as kernel argument.

`template <typename T>`
`    void set_arg(int index, accessor & accObj)`
	Set kernel args for an OpenCL kernel which is used through the SYCL/OpenCL interoperability interface. The index value specifies which parameter of the OpenCL kernel is being set and the accessor object, which OpenCL buffer or image is going to be given as kernel argument.

	Continued on next page

Table 5.15: Methods for the `execution_handle` class

Methods	Description

`template <typename T, std::enable_if<(`
`    is_same(E,svm_fine_grain),`
`    T>::type * = nullptr>`
`    void register_access(T *)`
	Method for adding a fine grained buffer virtual memory pointer allocation to the command group, so that it can be available for usage by the SYCL kernel. This is only necessary for fine grained buffer SVM mode, since the arguments to the kernel have to be explicitly set as global memory pointer allocations for the OpenCL device.

`template <typename KernelName, class KernelType>`
`    void single_task(KernelType)`
	Kernel invocation method of a kernel defined as a lambda or functor. If it is a lambda function or the functor type is globally visible there is no need for the developer to provide a kernel name type (`typename KernelName`) for it, as described in 5.4

`template <typename KernelName, class KernelType>`
`    void parallel_for(`
`    range<dimensions> numWorkItems, KernelType)`
	Kernel invocation method of a kernel defined as a lambda or functor, for the specified range and given an id or item for indexing in the indexing space defined by range. If it is a lambda function or if the functor type is globally visible there is no need for the developer to provide a kernel name type (`typename KernelName`) for it, as described in detail in 5.4

`template <typename KernelName, class KernelType>`
`    void parallel_for(`
`    range<dimensions> numWorkItems,`
`    id<dimensions> workItemOffset, KernelType)`
	Kernel invocation method of a kernel defined as a lambda or functor, for the specified range and offset and given an id or item for indexing in the indexing space defined by range. If it is a lambda function or the if the functor type is globally visible there is no need for the developer to provide a kernel name type (`typename KernelName`) for it, as described in detail in 5.4

`template <typename KernelName, class KernelType>`
`    void parallel_for(`
`    nd_range<dimensions> executionRange, KernelType)`
	Kernel invocation method of a kernel defined as a lambda or functor, for the specified `nd_range` and given an `nd_item` for indexing in the indexing space defined by the `nd_range`. If it is a lambda function or the if the functor type is globally visible there is no need for the developer to provide a kernel name type (`typename KernelName`) for it, as described in detail in 5.4

Continued on next page

Table 5.15: Methods for the `execution_handle` class

Methods	Description

`template <typename KernelName, class KernelType>`
`    void parallel_for(`
`    nd_range<dimensions> numWorkItems`
`    id<dimensions> workItemOffset, KernelType)`
	Kernel invocation method of a kernel defined as a lambda or functor, for the specified `nd_range` and given an `nd_item` for indexing in the indexing space defined by the `nd_range`. If it is a lambda function or the if the functor type is globally visible there is no need for the developer to provide a kernel name type (`typename KernelName`) for it, as described in detail in 5.4

`template <class KernelName, class`
`WorkgroupFunctionType>`
`    void parallel_for_work_group(`
`    range<dimensions> numWorkGroups,`
`    WorkgroupFunctionType)`
	Hierarchical kernel invocation method of a kernel defined as a lambda encoding the body of each work-group to launch. May contain multiple kernel built-in `parallel_for_work_item` functions representing the execution on each work-item. Launches `num_work_groups` work-groups of runtime-defined size. Described in detail in 5.4.

`template <class KernelName, class`
`WorkgroupFunctionType>`
`    void parallel_for_work_group(`
`    range<dimensions> numWorkGroups,`
`    range<dimensions> workGroupSize,`
`    WorkgroupFunctionType)`
	Hierarchical kernel invocation method of a kernel defined as a lambda encoding the body of each work-group to launch. May contain multiple kernel built-in `parallel_for_work_item` functions representing the execution on each work-item. Launches `num_work_groups` work-groups of `work_group_size` work-items each. Described in detail in 5.4.

`void single_task(kernel syclKernel)`	Kernel invocation method of a kernel defined as pointer to a kernel object, described in detail in 5.4

`void parallel_for(`
`    range<dimensions> numWorkItems,`
`    kernel sycl_kernel)`
	Kernel invocation method of a kernel defined as pointer to a kernel object, for the specified range and given an id or item for indexing in the indexing space defined by range, described in detail in 5.4

`void parallel_for(`
`    nd_range<dimensions> ndRange,`
`    kernel syclKernel)`
	Kernel invocation method of a kernel defined as pointer to a kernel object, for the specified `nd_range` and given an `nd_item` for indexing in the indexing space defined by the `nd_range`, described in detail in 5.4

	End of table

Table 5.15: Methods for the `execution_handle` class

In the case of OpenCL/SYCL interoperability the `execution_handle` class provides a method for setting kernel arguments for a kernel that is built from the OpenCL host interface or other libraries, like CLU. The developer may have already set the kernel arguments or may choose to use this method, keeping in mind that this method is not thread safe, as per the OpenCL specification.

5.4 SYCL functions for invoking kernels

Kernels can be invoked as *single tasks*, basic *data-parallel kernels*, OpenCL-style *NDRanges* in *work-groups*, or SYCL *hierarchical parallelism*.

Kernels can be invoked from command groups submitted on host or on device. The kernels that are submitted on device, are using nested command groups, which can have different nd-ranges and are going to be executed after the completion of the body of the kernel that submits them. All of the kernel invocations are allowed when *nested parallelism* is used.

Each function takes a kernel name template parameter. The kernel name must be a datatype that is unique for each kernel invocation. If a kernel is a functor, and its type is globally visible, then the kernel's functor type will be automatically used as the kernel name and so the user does not need to supply a name. If the kernel function is a lambda function, then the user must manually provide a kernel name to enable linking between host and device code to occur.

All the functions for invoking kernels are methods of the command group `cl::sycl::execution_handle` class 5.3, which is used to encapsulate all the methods provided in a command group scope.

5.4.1 `single_task` invoke

SYCL provides a simple interface to enqueue a kernel that will be sequentially executed on an OpenCL device. Only one instance of the kernel will be executed. This interface is useful a primitive for more complicated parallel algorithms, as it can easily create a chain of sequential tasks on an OpenCL device with each of them managing its own data transfers.

This function can only be called inside a command group using the command group `execution_handle` object created by the runtime. Any accessors that are used in a kernel should be defined inside the same command group.

Local accessors are disallowed for single task invocations.

```
auto command_group_lambda = [&](execution_handle &cgh) {
  auto an_accessor = a_buffer.get_access<access::mode::read_write>(cgh);
cgh.single_task<class kernel_name>(
    [=] () {
    // [kernel code]
    an_accessor[0]++;
    }));
};

class afunctor {
  accessor<int, 1, access::mode::read_write, access::target::global_buffer> ptr;

 public:
  afunctor(accessor<int, 1, access::mode::read_write,
                    access::target::global_buffer> p)
      : ptr(p) {}

  void operator()() {
    // [kernel code]
    an_accessor[0]++;
  }
}; /*
   ***************************************************************************************************
   */

operator()() auto command_group_functor = [&](execution_handle &cgh) {
  auto an_accessor = a_buffer.get_access<access::mode::read_write>(cgh);
  cgh.single_task(afunctor(ptr));
}
```

For single tasks, the kernel method takes no parameters, as there is no need for indexing classes in a unary index space.

5.4.2 parallel_for invoke

The `parallel_for` interface offers the ability to the SYCL users to declare a kernel and enqueue it as a parallel execution over a range of instances. There are three variations to the parallel_for interface and they depend on the index space that the developer would like the kernels to be executed on and the feature set available in those kernels.

In the simplest case, the developer only needs to provide the number of work-items the kernel will use in total and the system will use the best range available to enqueue it on a device. In this case the developer, may only need to know the index over the total *range* that he has provided, by providing the number of work-items that will be executing on. This type of kernels will be using the parallel_for invocation with a `range` type to provide the range of the execution and an `id` to provide the index within that range. Whether it is a lambda function or a kernel functor the parameter to the invocation function need to be `id`.

An example of a parallel_for using a lambda function for a kernel invocation in this case of parallel_for is the following.

```
class MyKernel;

myQueue.submit( [&](execution_handle & cmdgroup)
{
    auto acc=myBuffer.get_access<read_write>();

    cmdgroup.parallel_for<class MyKernel>(range<1>(workItemNo),
                                   [=] (id<1> index)
    {
        acc[index] = 42.0f;
    });
});
```

Local accessors are disallowed for the basic `parallel_for` invocations described above.

Another case, which is based on this very basic parallel_for, is the case where the developer would like to let the runtime choose the index space that is matching best the *range* provided but would like to use information given the scheduled interface instead of the general interface. This is enabled by using the class `item` as an indexing class in the kernel, and of course that would mean that the kernel invocation would match the `range` with the `item` parameter to the kernel.

In this case, there is also the case where the developer may want to execute functions on a work-group or sub-group level. The developer in this invocation lets the SYCL runtime decide what is the preferred work-group range for the device the kernel is enqueued on. The functions `item::get_group()` and `item::get_sub_group` allow the developer to get access to the group and sub-group functionality.

The kernels in this kernel invocation can only be uniform and no barrier functionality is available.

```
class MyKernel;

myQueue.submit([&](execution_handle & cmdgroup)
{
    auto acc=myBuffer.get_access<read_write>();

    cmdgroup.parallel_for<class MyKernel>(range<1>(workItemNo),
                                 [=] (item<1> myItem)
    {
        size_t index = item.get_global();
        acc[index] = 42.0f;
    });
});
```

The following two examples show how a kernel functor can be launched over a 3D grid, 3 elements in each dimension. In the first case work-item IDs range from 0 to 2 inclusive, in the second case work-item IDs they run from 1 to 3.

```
auto command_group = [&](execution_handle & cgh) {
 cgh.parallel_for<class example_kernel1>(
    range<3>(3,3,3), // global range
      [=] (item<3> it) {
        //[kernel code]
      });
};
auto command_group2 = [&](execution_handle & cgh) {
 cgh.parallel_for<class example_kernel2>(
    range<3>(3,3,3), // global range
    id<3>(1,1,1), // offset
      [=] (item<3> it) {
        //[kernel code]
      });
};
```

The last case of a parallel_for invocation enables low-level functionality of work-items, work-groups and sub-groups. This becomes valuable, when an execution requires groups of work-items that communicate and synchronize. These are exposed in SYCL through `parallel_for (nd_range,...)` and the `nd_item` class, which provides all the functionality of OpenCL for an NDRange. In this case, the developer needs to define the `nd_range` that the kernel will execute on in order to have fine grained control of the enqueueing of the kernel. This variation of parallel_for expects an `nd_range`, specifying both local and global ranges, defining the global number of work-items and the number in each cooperating work-group. The resulting functor or lambda is passed an `nd_item<i>`nstance making all the information available as well as `barrier` primitives to synchronize the work-items in the group or sub_group.

The following example shows how sixty-four work-items may be launched in a three-dimensional grid with four in each dimension and divided into sixteen work-groups. Each group of work-items synchronizes with a barrier.

```
auto command_group = [&](execution_handle& cgh) {
  cgh.parallel_for<class example_kernel>(
      nd_range(range(4, 4, 4), range(2, 2, 2)), [=](nd_item<3> item) {
        //[kernel code]
        // Internal synchronization
        item.barrier(access::fence_space::global);
        //[kernel code]
      });
};
```

Work-items can also be grouped in sub-groups within a work-group. In that case, a sub_group5.2.7 is a vector of work-items and depending on the capabilities of the device, they can have independent forward progress. They can be synchronized using barriers and also work-item functions can also be applied to them.

Optionally, in any of these variations of parallel_for invocations, the developer may also pass an offset. An offset is an instance of the id class added to the identifier for each point in the range.

In all of these case the underlying *nd_range* will be created and the kernel defined as a lambda or as a kernel functor will be created and enqueued as part of the command group.

Another case for the parallel_for invocation is to be used inside a device command group, where the kernel is enqueued from a device kernel. In this case of *nested parallelism*, which is available from OpenCL 2.0 and onwards, the kernels are enqueued by a parent kernel and the accessors or svm pointers can be only the ones already defined in the main command group that was defined on host. This kernel invocation is only available through a device_handle class which is constructed and made available from a host command group handler of type execution_handler<opencl22>.

5.4.3 Hierarchical invoke

The hierarchical parallel kernel execution interface provides the same functionality as is available from the nd_range interface but exposed differently. The hierarchical parallelism approach models the four levels of parallelism which are available in the OpenCL execution model through three nested *parallel_for* function scopes. The first level of the hierarchy, or the *outer parallel_for* is a function call to parallel_for_work_group and is within the work-group level. This is semantically equivalent to executing the body once per work-group. The second level of the hierarchy, which is optional, but could be useful for optimization, is the sub-group level. The function scope of the parallel_for_sub_group function is defining what is going to be executed on a vector of work-items. A third nested level is the parallel_for_work_item which is executing the body of the lambda or functor object per work-item. In the case of *nested parallelism*, paralle_for_work_group can be called in device command group, however, there cannot be a command group that is scheduled in any of the levels of the hierarchical. It is undefined behavior, if a device command group is scheduled from within the hierarchical api. However, it valid to call a the hierarchical api from a device command group.

To show the different levels of parallelism, we show the previous example, using hierarchical api. To execute the same sixty-four work-items in sixteen work-groups that we saw in the previous example, we execute an outer parallel_for_work_group call to create the groups. parallel_for_work_group is parameterized by the number of work-groups, such that the size of each group is chosen by the runtime, or by the number of work-groups and number of work-items for users who need more control.

The body of the outer parallel_for_work_group call consists of a lambda function or function object. The body of this function object contains code that is executed only once for the entire work-group. If the code has no side-effects and the compiler heuristic suggests it is more efficient to do so this code will be executed for each work-item.

Within this region any variable declared will have the semantics of local memory, shared between all work-items in the work-group. If the compiler can prove that an array of such variables is accessed only by a single work-item throughout the lifetime of the work-group, for example if access is directly from the id of the work-item with no transformation, then it can allocate the data in private memory or registers instead.

To guarantee use of private per-work-item memory, the private_memory class can be used to wrap the data. This class very simply constructs private data for a given group across the entire group. The id of the current work-item is passed to any access to grab the correct data.

The private_memory class has the following interface:

```
template <typename T, int Dimensions>
class private_memory {
 public:
  // Construct based directly off the number of work-items
  private_memory(const group<Dimensions> &);

  // Access the instance for the current work-item
  T &operator()(const item<Dimensions> &id);
};
```

Private memory is allocated per underlying work-item, not per iteration of the parallel_for_work_item loop. The number of instances of a private memory object is only under direct control if a work-group size is passed to the parallel_for_work_group call. If the underlying work-group size is chosen by the runtime the number of private memory instances is opaque to the program. Explicit private memory declarations should therefore be used with care and with a full understanding of which instances of a parallel_for_work_item loop will share the same underlying variable.

Private memory will be reused modulo the underlying work-group size in each dimension. For example, a $2x2$ work-group will allocate 4 private memory variables and repeat them three times horizonally and twice vertically in a $6x4$ parallel_for_work_item loop.

Also within the lambda body can be a sequence of calls to parallel_for_work_item. At the edges of these inner parallel executions the work-group synchronizes. As a result the pair of parallel_for_work_item calls in the code below is equivalent to the parallel execution with a barrier in the earlier example.

```
auto command_group = [&](execution_handle & cgh) {
  // Issue 8 work-groups of 8 work-items each
  cgh.parallel_for_work_group<class example_kernel>(
      range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup) {

    //[workgroup code]
    int myLocal;  // this variable shared between workitems
    // this variable will be instantiated for each work-item separately
    private_memory<int> myPrivate(myGroup);

    // Issue parallel sets of work-items each sized using the runtime default
    parallel_for_work_item(myGroup, [=](item<3> myItem) {
      //[work-item code]
      myPrivate(myItem) = 0;
    });

    // Carry private value across loops
    parallel_for_work_item(myGroup, [=](item<3> myItem) {
      //[work-item code]
      output[myGroup.get_local_range()*myGroup.get()+myItem] =
          myPrivate(myItem);
    });
    //[workgroup code]
  });
});
```

It is valid to use more flexible dimensions of the work-item loops. In the following example we issue 8 work-groups but let the runtime choose their size, by not passing a work-group size to the `parallel_for_work_group` call. The `parallel_for_work_item` loops may also vary in size, with their execution ranges unrelated to the dimensions of the work-group and the compiler generating an appropriate iteration space to fill the gap.

```
auto flexible_command_group = [&](execution_handle & cgh) {
  // Issue 1000 work-groups of 8 work-items each
  cgh.parallel_for_work_group<class example_kernel>(
      range<3>(2, 2, 2), [=](group<3> myGroup) {

    // Launch a set of 8 work-items as requested in the parallel_for_work_group launch
    parallel_for_work_item(myGroup, [=](item<3> myItem) {
      //[work-item code]
    });
    // Launch 512 iterations that will map to the underlying 8
    parallel_for_work_item(myGroup, range<3>(8, 8, 8), [=](item<3> myItem) {
      //[work-item code]
    });
    //[workgroup code]
  });
});
```

This interface offers a more intuitive way to tiling parallel programming paradigms. In summary, the hierarchical model allows a developer to distinguish the execution at work-group level and at work-item level using the `parallel_for_workgroup` and the nested `parallel_for_work_item` functions. It also provides this visibility to the compiler without the need for difficult loop fission such that a host execution may be more efficient.

5.5 Nested parallelism

Nested parallelism is a form of parallelism where more work can be sumbitted to the device, with the scope of the submission to be the SYCL host command group. All of the subsequent device command groups that are submitted from the parent command group kernel will be enqueued on a `device_queue` and executed asynchronously in relation to the parent kernel. However, the parent SYCL command group kernel will not be complete until all the child device command groups have been completed. Any errors that occur during the device command groups will be reported back from the parent command group and the SYCL host queue that it is sumbitted to.

The device command group, is submitted to a `device_queue` and enqueues commands using the device-side class `device_handle`. The latter supports all the kernel invocation methods described above, however, it is not supported by the `accessor` class and does not have the method `register_access()`, as those methods can only be called on the host. The SYCL runtime does not schedule kernels on the device-side and as a result only the SYCL kernel library classes and functions can be used for scheduling SYCL kernels on the device.

The SYCL kernel library classes and functionality is described in 7.3

A simple example of a nested kernel invocation is the following:

```
cl::sycl::capability_selector selectDevice(
    cl::sycl::exec_capabilities::opencl22);

cl::sycl::queue q(selectDevice, asyncHandler);
cl::sycl::buffer<int, 1> aBuffer(aHostPtr, cl::sycl::range<1>(numElements));
cl::sycl::buffer<int, 1> anotherBuffer(aHostPtr,
                                       cl::sycl::range<1>(numElements2));

q.submit([&](cl::sycl::execution_handle<opencl22>& cgh) {
  auto anAcc = aBuffer.get_access<cl::sycl::access::mode::read_write>(cgh);
  auto anotherAcc =
      anotherBuffer.get_access<cl::sycl::access::mode::read_write>(cgh);

  cgh.single_task<class device_side_enqueue>(
      cl::sycl::range<1>(1), [=](cl::sycl::id<1> index) {
        anAcc[index]++;

        device_queue dq = q.get_default_queue();
        auto event = dq.submit(
            enqueue_policy::wait_kernel, [&](cl::sycl::device_handle& dh) {
              int error = dh.parallel_for(
                  range<1>(numElements2),
                  [=](cl::sycl::id<1> idx) { anotherAcc[idx]--; });
            });
      });
});
```

5.6 Kernel class

The *kernel class* is an abstraction of a host kernel object in SYCL. For the most common case, the kernel object will contain the compiled version of a kernel invoked inside a command group using one of the parallel interface functions as described in 5.4. The SYCL runtime will create a kernel object when it needs to enqueue the kernel on a command queue.

In the case where a developer would like to pre-compile a kernel or compile and link it with an existing program, then the kernel object will be created and contain that kernel using the program class, as defined in 5.7. An both the above cases, the developer cannot instantiate a kernel object but can instantiate an object a functor class that he could use or create a functor from a kernel method using C++11 features. The kernel class object needs a `parallel_for(...)` invocation or an explicit `compile_and_link()` call through the program class, for this compilation of the kernel to be triggered.

Finally, a kernel class instance may encapsulate an OpenCL *kernel* object that was created using the OpenCL C interface and the arguments of the kernel already set. In this case since the developer is providing the cl_kernel object, this constructor is allowed to be used by the developer.

The kernel class also provides the interface for getting information from a kernel object on host. The kernel information descriptor interface is described in C.5 and the description is in the table 5.18.

```
namespace cl {
namespace sycl {
class kernel {
 private:
  friend class program;

  // The default object is not valid because there is no
  // program or cl_kernel associated with it
  kernel();

 public:
  kernel(const kernel& rhs);

  kernel(cl_kernel openclKernelObejct);

  cl_kernel get() const;

  context get_context() const;

  program get_program() const;

  template <info::kernel param>
  typename info::param_traits<info::kernel, param>::type
    get_info() const;
};
}  // namespace sycl
}  // namespace cl
```

Constructor	**Description**
`kernel (cl_kernel openclKernelObj)`	Constructor for SYCL kernel class given an OpenCL kernel object with set arguments, valid for enqueuing. Retains a reference to the `cl_kernel` object. Caller should release the passed `cl_kernel` object when it is no longer needed.
`kernel (const kernel& rhs)`	Copy constructor for kernel class.
	End of table

Table 5.16: `kernel` class constructors

Methods	Description
`cl_kernel get()`	Return the OpenCL kernel object for this kernel. Retains a reference to the returned `cl_kernel` object. Caller should release it when finished.
`context get_context()`	Return the context that this kernel is defined for.
`program get_program()`	Return the program that this kernel is part of.
`template <info::kernel param> typename info::param_traits< info::kernel, param>::type get_info()const`	Query information from the kernel object using the `info::kernel_info` descriptor.
	End of table

Table 5.17: Methods for the `kernel` class.

Kernel Descriptors	Return type	Description
`info::kernel::function_name`	`string_class`	Return the kernel function name.
`info::kernel::num_args`	`cl_uint`	Return the number of arguments to the extracted OpenCL C kernel.
`info::kernel::reference_count`	`cl_uint`	Return the reference count of the kernel object.
`info::kernel::attributes`	`string_class`	Return any attributes specified using the `__attribute__` qualifier with the kernel function declaration in the program source.
		End of table

Table 5.18: Kernel class information descriptors.

5.7 Program class

A *program* contains one or more kernels and any functions or libraries necessary for the program's execution. A program will be enqueued inside a context and each of the kernels will be enqueued on a corresponding device. Program class can be really useful for pre-compiling kernels and enqueuing them on multiple command groups. It also allows usage of functions define in OpenCL kernels from SYCL kernels via compiling and linking them in the same program object.

The program class provides an interface for getting information from a program object. The program information descriptor interface is described in C.6 and the description is in the table 5.21.

```
namespace cl {
namespace sycl {
class program {
 public:
  // Create an empty program object
  explicit program(const context& context);

  // Create an empty program object
  program(const context& context, vector_class<device> deviceList);

  // Create a program object from a cl_program object
  program(const context& context, cl_program clProgram);

  // Create a program by linking a list of other programs
  program(vector_class<program> programList, string_class linkOptions = "");

  program(const program& rhs);

  ~program();

  /* This obtains a SYCL program object from a SYCL kernel name
     and compiles it ready to link */
  template <typename kernelT>
  void compile_from_kernel_name(string_class compileOptions = "");

  /* This obtains a SYCL program object from a SYCL kernel name
     and builds it ready-to-run */
  template <typename kernelT>
  void build_from_kernel_name(string_class compileOptions = "");

  void link(string_class linkingOptions = "");

  // Get a kernel from a given Name (Functor)
  template <typename kernelT>
  kernel get_kernel<kernelT>() const;

  template <info::program param>
  typename info::param_traits<info::program, param>::type
    get_info() const;

  vector_class<vector_class<char>> get_binaries() const;

  vector_class<::size_t> get_binary_sizes() const;

  vector_class<device> get_devices() const;

  string_class get_build_options() const;

  cl_program get() const;

  bool is_linked() const;
};
}  // namespace sycl
}  // namespace cl
```

Constructors	Description
`explicit program ( const context & context)`	Constructs an empty program object for context for all associated devices with context.

Continued on next page

Table 5.19: Constructors for the `program` class

Constructors	Description
`program ( const context & context, vector_class<device> deviceList)`	Constructs an empty program object for all the devices of *device_list* associated with the *context*.
`program ( vector_class<program> programList, string_class linkOptions="")`	Constructs a program object for a list of programs and links them together using the `linkOptions`.
`program ( const context & context, cl_program clProgram)`	Constructs a program object for an OpenCL program object. Retains a reference to the returned `cl_program` object. Calling context should release it when finished.
`program(const program& rhs);`	Copy constructor for the program class.
	End of table

Table 5.19: Constructors for the `program` class

Methods	Description
`template<typename kernelT> void compile_from_kernel_name( string_class compileOptions="")`	Compile the kernel defined to be of type *kerneT* into the program, with compile options given by compile_options". The kernel can be defined either as a functor of type *kerneT* or as a lambda function which is named with the class name *kernelT*. The program object will need to be linked later.
`template<typename kernelT> void build_from_kernel_name( string_class compileOptions="")`	Build the kernel defined to be of type *kerneT* into the program, with compile options given by compile_options". The kernel can be defined either as a functor of type *kerneT* or as a lambda function which is named with the class name *kernelT*.
`void link(string_class linking_options = "")`	Link all compiled programs that are added in the program class.
`template<info::program param> typename info::param_traits< info::program, param>::type get_info()`	Retrieve information of the built OpenCL program object.
`vector_class<char*> get_binaries()const`	Return the array of compiled binaries associated with the program, as compiled for each device.
`vector_class<device> get_devices()const`	Return the list of devices this program was constructed against
`string_class get_build_options()const`	Retrieve the set of build options of the program. A program is created with one set of build options.
`cl_program get()const`	Return the OpenCL program object for this program. Retains a reference to the returned `cl_program` object. Caller should release it when finished.
	End of table

Table 5.20: Methods for the `program` class

Program Descriptors	Return type	Description
`info::program::reference_count`	`cl_uint`	Return the reference count of the kernel object.
`info::program::context`	`cl_context`	Return the context object this program is associated with.
`info::program::devices`	`vector_class<cl_device_id>`	Return set of devices this program is built against.
		End of table

Table 5.21: Program class information descriptors.

Programs allow the developers to provide their own compilation and linking options and also compile and link on demand one or multiple kernels. Compiler options allowed are described in the OpenCL specification [1, p. 145, § 5.6.4] and the linker options are described in [1, p. 148, § 5.6.5].

5.8 Defining kernels

In SYCL functions that are executed in parallel on a SYCL device are referred to as *kernel functions*. A *kernel* containing such a *kernel function* is enqueued on a device queue in order to be executed on that particular device. The return type of the *kernel function* is `void`, and all kernel accesses between host and device are defined using the accessor class 4.7.

There are three ways of defining kernels, defining them as functors, as C++11 lambda functions or as OpenCL `cl_kernel` objects. However, in the case of OpenCL kernels, the developer is expected to have created the kernel and set the kernel arguments.

5.8.1 Defining kernels as functors

A kernel can be defined as a C++ functor. In this case, the *kernel function* is the function defined as `operator()` in the normal C++ functor style. These functors provide the same functionality as any C++ functors, with the restriction that they need to follow C++11 standard layout rules. The kernel function can be templated via templating the kernel functor class. The *operator()* function may take different parameters depending on the data accesses that defined for the specific kernel.For details on restrictions for kernel naming issues, please refer to 9.2.

In the following example we define a trivial functor with no outputs and no accesses of host or pre-allocated device data. The kernel is executed on a unary index space for the specific example, since its using single_task at its invocation.

```
class MyFunctor
{
    float m_parameter;

public:
    MyFunctor(float parameter):
        m_parameter(parameter)
    {
    }

    void operator() ()
    {
        // [kernel code]
    }
};

void workFunction(float scalarValue)
{
    MyFunctor myKernel(scalarValue);

    queue.submit([&] (execution_handle & cmdGroup) {
      cmdgroup.single_task(myKernel);
    });
}
```

5.8.2 Defining kernels as lambda functions

In C++11, functors can be defined using lambda functions. We allow lambda functions to define kernels in SYCL, but we have an extra requirement to *name lambda functions* in order to enable the linking of the SYCL device kernels with the host code to invoke them. The name of a lambda function in SYCL is a C++ class. If the lambda function relies on template arguments, then the name of the lambda function must contain those template arguments. The class used for the name of a lambda function is only used for naming purposes and is not required to be defined.For details on restrictions for kernel naming issues, please refer to 9.2.

To invoke a C++11 lambda, the kernel name must be included explicitly by the user as a template parameter to the kernel invoke function.

The kernel function for the lambda function is the lambda function itself. The kernel lambda must use copy for all of its captures (i.e. `[=]`).

```
class MyKernel;

command_queue.submit([&](execution_handle& cmdGroup) {
  cmdgroup.single_task<class MyKernel>([=]() {
    // [kernel code]
  });
});
```

5.8.3 Defining kernels using program objects

In case the developer needs to specify compiler flags or special linkage options for a kernel, then a kernel object can be used, as described in 5.7. The kernel is defined as a functor 5.8.1 or lambda function 5.8.2. The user can obtain a program object for the kernel with the `get_kernel` method. This method is templated by the *kernel name*, so that the user can specify the kernel whose associated kernel they wish to obtain.

In the following example, the kernel is defined as a lambda function. The example obtains the program object for the lambda function kernel and then passes it to the `parallel_for`.

```
class MyKernel;  // Forward declaration of the name of the lambda functor

cl::sycl::queue myQueue;
cl::sycl::program MyProgram(myQueue.get_context());

/* use the name of the kernel to obtain the associated program */
MyProgram.build_from_name<MyKernel>();

myQueue.submit([&](execution_handle& commandGroup) {
  commandgroup.parallel_for<class MyKernel>(
      cl::sycl::nd_range<2>(4, 4),
      MyProgram.get_kernel<MyKernel>(),  // execute the kernel as compiled in MyProgram
      ([=](cl::sycl::nd_item<2> index) {
        //[kernel code]
      }));
});
```

In the above example, the *kernel function* is defined in the `parallel_for` invocation as part of a lambda functor which is named using the type of the forward declared class "myKernel". The type of the functor and the program object enable the compilation and linking of the kernel in the program class, *a priori* of its actual invocation as a kernel object. For more details on the SYCL device compiler please refer to chapter 9.

In the next example, a SYCL kernel is linked with an existing pre-compiled OpenCL C program object to created a combined program object, which is then called in a `parallel_for`.

```
class MyKernel;  // Forward declaration of the name of the lambda functor

cl::sycl::queue myQueue;

// obtain an existing OpenCL C program object
cl_program myClProgram = ...;

// Create a SYCL program object from a cl_program object
cl::sycl::program myExternProgram(myQueue.get_context(), myClProgram);

// Release the program if we no longer need it as
// SYCL program retained a reference to it
clReleaseProgram(myClProgram);

// Add in the SYCL program object for our kernel
cl::sycl::program mySyclProgram(myQueue.get_context());
mySyclProgram.compile_from_kernel_name<MyKernel>("-my-compile-options");

// Link myClProgram with the SYCL program object
mySyclProgram.link(myExternProgram,"-my-link-options");

myQueue.submit([&](execution_handle& commandgroup) {
  commandgroup.parallel_for<class MyKernel>(
      cl::sycl::nd_range<2>(4, 4),
      myLinkedProgram.get_kernel<MyKernel>(),  // execute the kernel as compiled in MyProgram
      ([=](cl::sycl::nd_item<2> index) {
        //[kernel code]
      }));
});
```

5.8.4 Defining kernels using OpenCL C kernel objects

In OpenCL C [1] program and kernel objects can be created using the OpenCL C API, which is available in the SYCL system. Interoperability of OpenCL C kernels and the SYCL system is achieved by allowing the creation of a *SYCL kernel* object from an *OpenCL kernel* object.

The constructor using kernel objects from 5.16:

```
kernel::kernel(cl_kernel kernel)
```

creates a `cl::sycl::kernel` which can be enqueued using all the `parallel_for` functions which can enqueue a kernel object. This way of defining kernels assumes the developer is using OpenCL C to create the kernel and set the kernel arguments. The system assumes that the developer has already called set kernel arguments when they are trying to enqueue the kernel. Buffers do give ownership to their accessors on specific contexts and the developer can enqueue OpenCL kernels in the same way as enqueuing SYCL kernels. However, the system is not responsible for data management at this point. Note that like all constructors from OpenCL API objects, constructing a `cl::sycl::kernel` from a `cl_kernel` will retain a reference to the kernel and the user code should call `clReleaseKernel` if the `cl_kernel` is no longer needed in the calling context.

5.9 Rules for parameter passing to kernels

In a case where a kernel is a C++ functor or C++11 lambda object, any values in the functor or captured in the C++11 lambda object must be treated according to the following rules:

- Any accessor must be passed as an argument to the device kernel in a form that allows the device kernel to access the data in the specified way. For OpenCL 1.0–1.2 class devices, this means that the argument must be passed via `clSetKernelArg` and be compiled as a kernel parameter of the valid reference type. For global shared data access, the parameter must be an OpenCL `global` pointer. For an accessor that specifies OpenCL `constant` access, the parameter must be an OpenCL `constant` pointer. For images, the accessor must be passed as an `image_t` and/or sampler.

- The SYCL runtime and compiler(s) must produce the necessary conversions to enable accessor arguments from the host to be converted to the correct type of parameter on the device.

- A local accessor provides access to work-group-local memory. The accessor is not constructed with any buffer, but instead constructed with a size and base data type. The runtime must ensure that the work-group-local memory is allocated per work-group and available to be used by the kernel via the local accessor.

- C++ standard layout values must be passed by value to the kernel.

- C++ non-standard layout values must not be passed as arguments to a kernel that is compiled for a device.

- It is illegal to pass a buffer or image (instead of an accessor class) as an argument to a kernel. Generation of a compiler error in this illegal case is optional.

- Sampler objects (`cl::sycl::sampler`) can be passed as parameters to kernels.

- For OpenCL 1.2 devices, it is illegal to pass a pointer or reference argument to a kernel. Generation of a compiler error in this illegal case is optional.

- Any aggregate types such as structs or classes should follow the rules above recursively. It is not necessary to separate struct or class members into separate OpenCL kernel parameters if all members of the aggregate type are unaffected by the rules above.

5.10 Event classes for OpenCL interoperability

An *event* in SYCL abstracts the `cl::event` objects in OpenCL. In OpenCL events' mechanism is comprised of low-level event objects that require from the developer to use them in order to synchronize memory transfers, enqueuing kernels and signaling barriers.

In SYCL, events are an abstraction of the OpenCL event objects, but they retain the features and functionality of the OpenCL event mechanism. They accommodate synchronization between different contexts, devices and platforms. It is the responsibility of the SYCL implementation to ensure that when SYCL events are used in OpenCL queues, the correct synchronization points are created to allow cross-platform or cross-device synchronization.

The `device_event` is a class the represents OpenCL only device events, that are created only on device and are used as synchronization primitives when nested parallel device kernels are enqueued.

Since data management and storage is handled by the SYCL runtime, the `event` class is used for providing the appropriate interface for OpenCL/SYCL interoperability. In the case where the SYCL objects contain OpenCL memory objects created outside the SYCL mechanism then events can be used in order to provide to the SYCL runtime the initial events it has to synchronize against. However, the events mechanism does not provide full interoperability during the SYCL code execution with OpenCL. Interoperability is achieved by using the synchronization rules with `buffer` and `image` class.

A SYCL event can be constructed from an OpenCL event or can return an OpenCL event. The constructors and methods of the Event class are listed in Tables 5.22 and 5.23.

```
namespace cl {
namespace sycl {

enum class event_status {
  submitted,
  complete,
  error,
};

class event {
 public:
  event() = default;

  explicit event(cl_event clEvent);

  event(const event &rhs);

  ~event();

  cl_event get();

  vector_class<event> get_wait_list();

  void wait();

  static void wait(const vector_class<event> &eventList);

  void wait_and_throw();

  static void wait_and_throw(const vector_class<event> &eventList);

  template <info::event param>
  typename param_traits<info::event, param>::type get_info() const;

  template <info::event_profiling param>
  typename param_traits<info::event_profiling, param>::type get_profiling_info()
      const;
};

}  // namespace sycl
}  // namespace cl
```

Constructors	Description
`event ()= default`	Default construct a null event object.
`explicit event (cl_event clEvent)`	Construct a SYCL event from a `cl_event`, only used with the SYCL/ OpenCL interoperability interface for buffers and images.
`event (const event & rhs)`	Construct a copy sharing the same underlying event. The underlying event will be reference counted.
	End of table

Table 5.22: Constructors for the `event` class

Methods	Description
`cl_event get()`	Return the underlying OpenCL event reference. Retains a reference to the returned `cl_event` object. Caller should release it when finished.
`vector_class<event> get_wait_list()`	Return the list of events that this event waits for in the dependence graph.
`void wait()`	Wait for the event and the command associated with it to complete.
`void wait_and_throw()`	Wait for the event and the command associated with it to complete. If any uncaught asynchronous errors occurred on the context (or contexts) that the event is waiting on executions from, then will also call that context's asynchronous error handler with those errors.
`static void wait( const vector_class<event> &eventList)`	Synchronously wait on a list of events.
	Continued on next page

Table 5.23: Methods for the `event` class

Methods	Description

`static void wait_and_throw(`
`    const vector_class<event> &eventList)`
	Synchronously wait on a list of events. If any uncaught asynchronous errors occurred on the context (or contexts) that the events are waiting on executions from, then will also call those contexts' asynchronous error handlers with those errors.

`template <info::event param>`
`    typename param_traits`
`    <info::event, param>::type`
`    get_info ()const`
	Queries OpenCL information for the underlying *cl_event*.

`template <info::event_profiling param>`
`    typename param_traits`
`    <info::event_profiling, param>::type`
`    get_profiling_info ()const`
	Queries OpenCL profiling information for the underlying *cl_event*.

	End of table

Table 5.23: Methods for the `event` class

The `event_status` enum class is providing information on the status of the commands associated to the event.

Description	Description
`submitted`	This is the initial state of a user event.
`complete`	The event has completed
`error`	There was an error while executing the associated commands.
	End of table

Table 5.24: Description of the different states of an OpenCL event.

5.11 Error handling

5.11.1 Error Handling Rules

Error handling in SYCL uses exceptions. If an error occurs, it can be propagated at the point of a function call. An exception will be thrown and may be caught by the user using standard C++ exception handling mechanisms. For example, any exception which is triggered from code executed on host is able to be propagated at the call site and it will follow the standard C++ exception handling mechanisms.

SYCL applications are asynchronous in the sense that host and device code execution are executed asynchronously. As a result of this, the errors that occur on a device cannot be propagated directly from the call site, and they will not be detected until the error-causing task executes or tries to execute rather than been scheduled for execution. We refer to those errors as asynchronous errors. A good example of an asynchronous error, is an out-of-bounds access error. In this case, if the kernel is enqueued on SYCL OpenCL device then the out-of-bounds error is asynchronous with regards to the SYCL host application, as it is executed on the device. At the latter, the standard exception mechanisms will not be available as this is an asynchronous error.

SYCL queues are by default asynchronous, as they schedule tasks on SYCL devices. The queue constructor can optionally get an asynchronous handler object `async_handler`, which is a function class instance. If waiting and exception handling methods are used on queues, the *async_handler* is receiving a list of C++ exception objects.

If an asynchronous error occurs in a queue that has no user-supplied asynchronous error handler object `async_handler`, then no exception is thrown and the error is not available to the user in any specified way. Implementations may provide extra debugging information to users to trap and handle asynchronous errors. If a synchronous error occurs in a SYCL application and it is not handled, the application will exit abnormally.

If an error occurs when running or en-queuing a command group which has a secondary queue specified, then the command group may be enqueued to the secondary queue instead of the primary queue. The error handling in this case is also configured using the *async_handler* provided for both queues. If there is no *async_handler* given on any of the queues, then no asynchronous error reporting is done and no exceptions are thrown. If the primary queue fails and there is an *async_handler* given at this queue's construction, which populates the `exception_list` parameter, then any errors will be added and can be thrown whenever the user chooses to handle those exceptions. Since there were errors on the primary queue and a secondary queue was given then, the execution of the kernel is re-scheduled to the secondary queue and any error reporting for the kernel execution on that queue, is done through that queue, in the same way as described above. The secondary queue may fail as well, and the errors will be thrown if there is an *async_handler* and either `wait_and_throw()` or `throw()` are called on that queue. The command group functor handler event returned by that function will be relevant to the queue where the kernel has been enqueued.

5.11.2 Exception Class Interface

```
namespace cl {
namespace sycl {

using async_handler=function_class<void(cl::sycl::exception_list)>;

struct exception {
  string_class what();

  // returns associated context. nullptr if none
  context get_context();
};

struct cl_exception : exception {
  // thrown as a result of an OpenCL API error code
  cl_int get_cl_code() const;
};

struct async_exception : exception {
  // stored in an exception_list for asynchronous errors
};

class exception_list {
  // Used as a container for a list of asynchronous exceptions
 public:
  typedef exception_ptr value_type;
  typedef value_type& reference;
  typedef const value_type& const_reference;
```

```
    typedef size_t size_type;
    typedef /*unspecified*/ iterator;
    typedef /*unspecified*/ const_iterator;

    size_t size() const;
    iterator begin() const;  // first asynchronous exception
    iterator end() const;    // refer to past-the-end last asynchronous exception
  };

  typedef /*unspecified*/ exception_ptr;

  class runtime_error : public exception;

  class kernel_error : public runtime_error;

  class accessor_error : public runtime_error;

  class nd_range_error : public runtime_error;

  class event_error : public runtime_error;

  class invalid_parameter_error : public runtime_error;

  class device_error : public exception;

  class compile_program_error : public device_error;

  class link_program_error : public device_error;

  class invalid_object_error : public device_error;

  class memory_allocation_error : public device_error;

  class platform_error : public device_error;

  class profiling_error : public device_error;

  class feature_not_supported : public device_error;

  }  // namespace sycl
  }  // namespace cl
```

The cl_exception class is the exception thrown when the OpenCL API returns an error code. The OpenCL error code can be queried with the get_cl_code method. The async_exception is stored in exception_list objects and is generated when an asynchronous error occurs on a SYCL-managed context. The cl::sycl::exception_ptr class is used to store cl::sycl::exception objects and allows exception objects to be transferred between threads. It is equivalent to the std::exception_ptr class. The cl::sycl::exception_list class is also available.

The asynchronous handler object async_handler is a function_class with an exception_list as a parameter. The asynchronous handler is an optional parameter to a constructor of the queue class and it is the only way to handle asynchronous errors occurring on a SYCL device. The asynchronous handler may be a function_class that can be functor or lambda or function that can be given to the queue and it willbe executed on error. The exception_list object is constructed from the SYCL runtime and is populated with the errors caught during the execution of all the kernels running on the same queue.

Methods	Description
`string_class what()`	Returns a descriptive string for the error, if available.
`context get_context()`	Returns the context that caused the error. Returns `nullptr` if not a buffer error.
	End of table

Table 5.25: Methods of the `exception` class.

Methods	Description
`cl_int get_cl_code()`	Returns the OpenCL error code encapsulated in the exception. Only valid for the `cl_exception` subclass.
	End of table

Table 5.26: Methods of the `cl_exception` class.

Methods	Description
`size_t size()const`	Returns the size of the list
`iterator begin()const`	Returns an iterator to the beginning of the list of asynchronous exceptions.
`iterator end()const`	Returns an iterator to the end of the list of asynchronous exceptions.
	End of table

Table 5.27: Methods of the `exception_list`

Runtime Error Exception Type	Description
`kernel_error`	Error that occurred before or while enqueuing the SYCL kernel.
`nd_range_error`	Error regarding the `cl::sycl::nd_range` specified for the SYCL kernel
`accessor_error`	Error regarding the `cl::sycl::accessor` objects defined.
`event_error`	Error regarding associated `cl::sycl::event` objects.
`invalid_parameter_error`	Error regarding parameters to the SYCL kernel, it may apply to any captured parameters to the kernel lambda.
	End of table

Table 5.28: Exceptions types that derive from the `cl::sycl::runtime_error` class

Device Error Exception Type	**Description**
`compile_program_error`	Error while compiling the SYCL kernel to a SYCL device.
`link_program_error`	Error while linking the SYCL kernel to a SYCL device.
`invalid_object_error`	Error regarding any memory objects being used inside the kernel
`memory_allocation_error`	Error on memory allocation on the SYCL device for a SYCL kernel.
`device_error`	The SYCL device will trigger this exception on error.
`platform_error`	The SYCL platform will trigger this exception on error.
`profiling_error`	The SYCL runtime will trigger this error if there is an error when profiling info is enabled.
`feature_not_supported`	Exception thrown when an optional feature or extension is used in a kernel but its not available on the device the SYCL kernel is being enqueued on.
	End of table

Table 5.29: Exception types that derive from the `cl::sycl::device_error` class

6. Shared Virtual Memory

6.1 Overview

In SYCL 1.2, host and device need to transfer memory objects in order to share data for kernel execution. In SYCL 2.2, there is an option of having host and device share virtual memory address space and in effect avoid memory object transfers. Shared virtual memory, a.k.a. SVM, enables using complex data classes and pointers between host and devices, using atomics and in some cases the lifetime scope of the buffer and accessor classes as synchronization points. There are different flavors of shared virtual memory, depending on the capabilities of the OpenCL system to share device pointers with the host. The core capability is to have *coarse-grained* buffer sharing between host and device, by making the device pointer available on the host. This mode enables buffer objects to be allocated for host and device using the same context and in SYCL they are based on the lifetime of the `buffer` and `accessor` class. One restriction, that has to be noted for all SVM capabilities, is that they are only applicable to buffers, not to images or pipes.

There are different levels of sharing and the existence of atomics which are enabled for specifically this memory model are defining the behavior of this system. The common denominator to all of these modes and the difference of this feature with the rest of the SYCL buffer/accessor classes, is that all of the allocation need to be defined in the same SYCL context. The SYCL runtime does not manage the allocations out of that context and inside that context, depending on the level of sharing, it may be that any host allocation can be valid, even with using malloc on the host. All interoperability with the buffers and accessor is based only on standard layout structs that will be copied and the pointers are going to be invalidated due to different allocation context.

6.2 Coarse-grained shared virtual memory

Coarse-grained shared virtual memory is a core feature of SYCL and exposes the architecture capabilities for coarse-grained shared memory objects of OpenCL 2.2 devices. A `buffer` in a `context` with devices with support for `svm_coarse_grain` is able to allocate coarse-grained device pointer on the host. This is possible using the `svm_allocator`. Access on host or device is requested by the creating an instance of the `accessor` class with targets either `access::target::host_buffer` or `access::target::svm_buffer`.

There three differences between svm and non-svm `buffer` instances. Firstly, the underlying pointer is guaranteed to be the same for host and device. Secondly, the coarsed-grained buffer allows sycl kernels on the OpenCL devices of that context to be able to process the whole or parts of the buffer host synchronization points. Lastly, due to the fact that the underlying pointers are shared, more complex pointer structures can be allocated using the buffers and used by the host application and the kernels. For example, pre-allocating a list and populating the pointers of the linked list in SYCL kernels is applicable in this case.

Nevertheless, the coarse-grained svm buffer pointers share the same data consistency restrictions with non-svm buffers. The data consistency between host and device requires host synchronization points, which in SYCL are managed by *buffer* and *accessor* classes. The usage of those classes with the rules that are based on the lifetime of the buffer and the accessor instances, can provide the SYCL runtime with enough information to guarantee data consistency of this pointer allocation within that context. Same as in non-svm buffers, host and device *accessor* instances cannot have overlappling lifetime scopes.

6.2.1 Coarse-grained buffer and and accessor interface

The buffer class as described in 4.2 is provides the above capabilities by providing providing the buffer constructor that allocates a buffer with the size given, without a host pointer associated with it and an *svm_allocator*; as described in 6.4.

`buffer` constructor	**Description**
`template<typename T, int Dimensions, typename SVMAllocatorT = svm_allocator<T,svm_coarse_grain>> buffer (const range<dimensions> & bufferrange, SVMAllocatorT svm_allocator)`	Creates a new buffer of the given size with storage managed by the sycl runtime. The buffers is using the `svm_allocator` and the context that is created with in order to allocate an svm pointer which is going to be used on host and on device. Type trais can be used at the buffer object in order to query that the svm_allocator was used for the allocation of the underlying svm pointer. This buffer cannot be used in a different context, since the allocation is restricted in the svm_-allocator context. If the type of the buffer, has the const qualifier, then the default allocator will remove the qualifier to allow host access to the data.
	End of table

Table 6.1: Coarse-graind shared virtual memory (SVM) constructor for the `buffer` class.

The SYCL interface provides helper global functions for allocating coarse-grained buffers.

Utility functions for coarse-graied svm buffers	**Description**
`template <class T> buffer<T,1,svm_allocator<T,svm_coarse_grain>> * make_svm_buffer(size_t size)`	Creates a coarse grained svm buffer, by creating a new context using the system heuristics for choosing a context based on the capabilities that are required.
`template<class T> buffer<T,1,svm_allocator<T,svm_coarse_grain>> *`	make_svm_buffer(size_t size, context & contextInput)
Creates a coarse grained svm buffer in the context provided.	
	End of table

Table 6.2: Utility functions for coarse grained svm buffers

The simple following example show the different ways that the an svm buffer can be allocated and used.

```
{
  cl::sycl::capability_selector selectDevice(
      cl::sycl::exec_capabilities::svm_coarse_grain);

  cl::sycl::queue q(selectDevice, asyncHandler);
  cl::sycl::context coarseContext(selectDevice);
  cl::sycl::svm_allocator<int> coarseSVMAllocator(coarseContext);

  /* Allocate SVM buffer using coarse grained buffer sharing virtual memory
   * address space.
   */
  cl::sycl::buffer<int, 1,
                   cl::sycl::svm_allocator<
                       int, cl::sycl::exec_capabilities::svm_coarse_grain>>
      svmBuffer(cl::sycl::range<1>(numElements), coarseSVMAllocator);
  {
    /* Access the SVM buffer on host */
    auto hostSvmCoarsePointer =
        svmBuffer.get_access<cl::sycl::access::mode::write,
                             cl::sycl::access::target::host_buffer>();
    /* Within this block it is safe to use the raw SVM pointer. */
    auto rawSvmCoarsePointer = hostSvmCoarsePointer.get();
    *rawSvmCoarsePointer = 100;
  }  // The underlying SVM pointer gets updated and no access on host is
     // possible when the host accessor goes out of scope

  q.submit([&](cl::sycl::execution_handle<svm_coarse_grain>& cgh) {
    /* Make the SVM pointer allocation available for updating on the device */
    auto deviceCoarsePointer =
        svmBuffer.get_access<cl::sycl::access::mode::read_write>(cgh);
    cgh.single_task<class svm_sample1>(
        cl::sycl::range<1>(1), [=](cl::sycl::id<1> index) {
          int* rawSvmCoarsePointer = deviceCoarsePointer.get();
          rawSvmCoarsePointer[index] = a + 1;
        });
  });  // dependency tracking should make sure no unnecessary mapping is
       // happening here

  q.submit([&](execution_handle<svm_coarse_grain>& cgh) {
    auto deviceCoarsePointer2 = svmBuffer.get_access<access::mode::read>(cgh);
    cgh.single_task<class svm_sample2>(range<1>(1), [=](id<1> index) {
      int* rawSvmCoarsePointer = deviceCoarsePointer2.get();
      int b = rawSvmCoarsePointer[index];
    });

  });
  {
    auto svmHostAccBack =
        svmBuffer.get_access<access::mode::read, access::target::host_buffer>();

    /** rawSvmCoarsePtr is the same in all cases, as the underlying SVM
     * allocation pointer managed by buffer is the guaranteed to be the same. */
    int* rawSvmCoarsePtr = svmHostAccBack.get();
  }
  /* When the svmBuffer gets out of scope then the allocation is freed. No
```

```
     * copy back is required since the SVM buffer allocation, is happening in the
     * SVM virtual address space.
     */
}
```

6.3 Fine-grained shared virtual memory

An optional feature of OpenCL systems is to provide fine grained shared virtual memory between the host and device for allocations in the same context. The granularity of the shared virtual address space depends on the data consistency the system is able to provide. Depending on the sharing granularity of the virtual address space and the support of atomics as synchronization primitives, the systems are able to provide memory consistency across different levels of granularity and concurrent access.

6.3.1 Sharing within buffer scope

Shared virtual memory with buffer sharing support denotes the capability of being able to share memory allocations between host and device, at the granularity of each allocation. The SVM pointers that are used in this mode need to be allocated with respect to specific `cl::sycl::context` and depending on the availability of atomic operations on those pointers, the synchronization differs.

Atomic enabled fine grained buffer sharing virtual memory provides the capability of sharing SVM pointer allocations between host and device and using the atomic primitives in order to guarantee memory consistency across host and device for those pointers. The atomics can guarantee constistency between host and kernel updates, although, standard rules for using atomics for global memory space still apply. For example, the atomics can guarantee, that an access on host will atomically load and store a SVM pointer allocation, but they cannot guarantee the ordering among the work-group loads and stores. The latter is defined in the rules of the memory mode in OpenCL 2.2 specification document.

Non-atomic fine grained buffer sharing virtual memory exposes the same virtual memory address of an SVM allocation to both host and device. Due to the lack of atomic operations on that virtual memory space, the loads and stores of the allocations cannot be atomic. As a result, the SVM allocations can be accessed by both host and device, but the updates have to be on non- overlapping parts of that memory allocation. The latter has to be managed by the developer, and the system cannot make any guarantees of the memory consistency if there are overlapping reads and writes to an SVM allocation.

In both cases of fine-grained buffer sharing SVM allocations, the allocation needs to use an instance of the `svm_allocator`. The `svm_allocator` is following the interface of standard C++ allocators and can be used in the same manner to STL container classes or smart pointer classes. The `svm_allocator` , needs to use a context where all the devices associated with that context can support the same SVM capabilities and all of the allocations using the same allocator instance will be using the same context. If the user doesn't provide a relevant context, then a context will be initialized by the SYCL runtime using implementation defined heuristics to choose devices that support the above capabilities, `context(capability_selector(exec_capabilities::svm_fine_grained_buffer_atomics))` or `context(capability_selector(exec_capabilities::svm_fine_grained_buffer))`. Any command group containing the

SVM pointer allocations from the `svm_allocator` has to be submitted using a queue, created from the same context.

The following code example shows how to make a fine grained buffer sharing virtual allocation.

```
/*Choose devices that supports fine grained capabilities*/
cl::sycl::capability_selector fineGrainSelector(
    cl::sycl::exec_capabilities::svm_fine_buffer_sharing);
/*Create context with all the available devices that support fine grained buffer
* sharing SVM in a platform chosen by custom system heuristics.
*/
cl::sycl::context fineGrainedContext(fineGrainSelector);
/* Use the fine grained buffer sharing selector to choose a device that
 * supports this SVM mode and create a queue using the context instance given.*/
cl::sycl::queue q(fineGrainSelector, fineGrainedContext);
/* Create an instance of the SVM fine grained buffer sharing allocator
 * which is going to be used for all the allocations of the same context.
 */
cl::sycl::svm_allocator<int, cl::sycl::svm_fine_grain> fineGrainAllocator(
    fineGrainedContext);
/* Use the instance of the svm_allocator with any custom container, for
 * example, with std::shared_ptr.
 */
std::shared_ptr<int> svmFineBuffer =
    std::allocate_shared<int>(fineGrainAllocator, numElements);
/* Initialize the pointer on the host */
*svmFineBuffer = 66;

q.submit([&](
    cl::sycl::execution_handler<svm_fine_grain<
        cl::sycl::svm_sharing::buffer, cl::sycl::svm_atomics::none>>& cgh) {
  auto rawSvmFinePointer = svmFineBuffer.get();
  /* Register access on the device in the specific command group with
   * access::mode so that dependency tracking can be possible and the
   * kernel can use the raw pointer. Custom containers are not working
   * on the device side unless they are structs provided by the user
   * and compiled for the device.
   */
  cgh.register_access<cl::sycl::access::mode::read_write>(rawSvmFinePointer);
  cgh.single_task<class svm_fine_grained_kernel>(
      range<1>(1), [=](id<1> index) { rawSvmFinePointer[index]++; });
});

q.submit([&](
    cl::sycl::execution_handle<svm_fine_grain<
        cl::sycl::svm_sharing::buffer, cl::sycl::svm_atomics::none>>& cgh) {
  auto rawSvmFinePointer = svmFineBuffer.get();
  /* Register access on the device in the specific command group with
  * access::mode so that dependency tracking can be possible and the
  * kernel can use the raw pointer. Custom containers are not working
  * on the device side unless they are structs provided by the user
  * and compiled for the device.
  */
  cgh.register_access<cl::sycl::access::mode::read_write>(rawSvmFinePointer);
  cgh.single_task<class svm_fine_grained_kernel_2>(
      cl::sycl::range<1>(1),
      [=](cl::sycl::id<1> index) { int val = rawSvmFinePointer[index]; });

});

int result = *svmFineBuffer;
```

The SVM pointer allocations returned by `svm_allocator.allocate()` can be used on host and in SYCL kernels committed to a queues of the same context without any additional synchronization. This is the reason why in this case there is no need for using the SYCL memory management classes (buffers and accessors). These allocations also need be deallocated using the `svm_allocator.deallocate()` method. Depending on the rules of the containers, these methods should be called when trying to allocate and de-allocate their internal pointers. In the case, where the raw-pointers are used directy, then those methods need to be called explicitly.

6.3.2 Sharing across the system

System sharing virtual memory provides the capability to share the entire host's virtual memory with the device. In that case, any pointers allocated on the host with the standard C++ API, can be used on the device without any additional synchronization APIs.

Atomic enabled system sharing virtual memory allows the host allocations to be concurrenly accessed by host and device kernels, while maintaining memory consistency. The atomics can guarantee consistency between host and kernel updates given the rules for atomics on OpenCL global memory. The system can guarantee that with the use of atomics on the SVM pointers then the memory will be consistent for loads and stores on host and on device.[1]

Non-atomic system sharing virtual memory allows the host allocations to be concurrently updated on non-overlapping memory locations. Due to the lack of synchronization primitives for those SVM pointers there can be no guarantees for overlapping updates. The system, however, can guarantee that the use of the same pointer allocation can be used on both host and device kernels without the need of explicit synchronization points.

System sharing virtual memory allows all the host allocations to be made available on the device using the standard C++ API for allocating and deallocating memory. This is the reason, why the use of the `svm_allocator` is unessecary. The use of buffers and accessors are also unecessary in this case, as the system does not have to explicitly synchronize for host and device updates.

The following code example shows how to make a fine grained system sharing virtual allocation.

```
/*Choose devices that supports fine grained capabilities*/
cl::sycl::capability_selector fineGrainSelector(
    cl::sycl::exec_capabilities::svm_fine_buffer_sharing);
/*Create context with all the available devices that support fine grained buffer
* sharing SVM in a platform chosen by custom system heuristics.
*/
cl::sycl::context fineGrainedContext(fineGrainSelector);
```

[1]The SVM pointers on the device are in OpenCL global memory, and the rules for work-group visibility on global memory have to be taken into consideration in terms of atomic ordering guarantees.

```
/* Use the fine grained buffer sharing selector to choose a device that
 * supports this SVM mode and create a queue using the context instance given.*/
cl::sycl::queue q(fineGrainSelector, fineGrainedContext);
/* Create an instance of the SVM fine grained buffer sharing allocator
 * which is going to be used for all the allocations of the same context.
 */
cl::sycl::svm_allocator<int, cl::sycl::svm_fine_grain> fineGrainAllocator(
    fineGrainedContext);
/* Use the instance of the svm_allocator with any custom container, for
 * example, with std::shared_ptr.
 */
std::shared_ptr<int> svmFineBuffer =
    std::allocate_shared<int>(fineGrainAllocator, numElements);
/* Initialize the pointer on the host */
*svmFineBuffer = 66;

q.submit([&](
    cl::sycl::execution_handler<svm_fine_grain<
        cl::sycl::svm_sharing::buffer, cl::sycl::svm_atomics::none>>& cgh) {
  auto rawSvmFinePointer = svmFineBuffer.get();
  /* Register access on the device in the specific command group with
   * access::mode so that dependency tracking can be possible and the
   * kernel can use the raw pointer. Custom containers are not working
   * on the device side unless they are structs provided by the user
   * and compiled for the device.
   */
  cgh.register_access<cl::sycl::access::mode::read_write>(rawSvmFinePointer);
  cgh.single_task<class svm_fine_grained_kernel>(
      range<1>(1), [=](id<1> index) { rawSvmFinePointer[index]++; });
});

q.submit([&](
    cl::sycl::execution_handle<svm_fine_grain<
        cl::sycl::svm_sharing::buffer, cl::sycl::svm_atomics::none>>& cgh) {
  auto rawSvmFinePointer = svmFineBuffer.get();
  /* Register access on the device in the specific command group with
  * access::mode so that dependency tracking can be possible and the
  * kernel can use the raw pointer. Custom containers are not working
  * on the device side unless they are structs provided by the user
  * and compiled for the device.
  */
  cgh.register_access<cl::sycl::access::mode::read_write>(rawSvmFinePointer);
  cgh.single_task<class svm_fine_grained_kernel_2>(
      cl::sycl::range<1>(1),
      [=](cl::sycl::id<1> index) { int val = rawSvmFinePointer[index]; });

});

int result = *svmFineBuffer;
```

6.4 svm_allocator interface

SYCL provides a standard allocator in order to allow coarse-grained and buffer sharing fine-grained allocations. The svm_allocator provides the standard C++ interface for allocators and is compatible with any container classes that are compatible with it. In the case wher no container or wrapper classes are used then the methods svm_allocator::allocate() and svm_allocator::deallocate() need to be called.

An important characteristic of this allocator is that it has state, since it needs to have a context as a member and make sure all the allocations done using the allocator instance are using the same context and that the context will not go out of scope before the pointers are deallocated.

The SVM allocations with buffer granularity are allocated and deallocated using the svm_allocator and are in the scope of one context. All the devices in that context have to sharing the same SVM capabilities that are being required, e.g. if fine grain buffer sharing is required, then all the associated devices to that context have to able to provide that feature. The default svm_allocator will create a context with the corresponding required capabilities calling the corresponding *context*. The context that is used for the allocations need to be the same that any queues scheduling SYCL kernels are associated with, otherwise the system will be unable to synchronize the updates and may result in undefined behavior. Any copies or moves from one context to another will invalidate any pointers allocated.

All the allocations are happening on the host and the restrictions of banning dynamic allocations on the device remain, which means that if there is usage of containers that cause resizing in the non system sharing mode, the pointers will be invalidated. Dynamic allocation on the device is still illegal, although placement new operators are allowed.

The interface of the svm_allocator is the following:

```
#ifndef RUNTIME_INCLUDE_SYCL_SVM_H_
#define RUNTIME_INCLUDE_SYCL_SVM_H_

#ifdef _WIN32

#include <malloc.h>
#endif

#include <memory>
namespace cl {
namespace sycl {

/** \brief The svm_allocator provides the C++ Allocator interface for SVM
 * allocations.
 * The default case is the core feature of coarse grained SVM allocation,
 * which is used only with conjuction to cl::sycl::buffer objects.
 *
 * In the case of fined grained buffer sharing SVM, this allocator can be
 * used with any C++ container or wrapper class, or used directly to
 * retrieve the raw allocation pointer. Care has to be taken with the reference
 * counting of C++ container classes, to make sure that the deallocation is
 * happening using the allocator.deallocate() method.
 *
 * This allocator is not applicable for fine grained system sharing allocations
 * since the standard C++ allocators can be used.
 */
template <class T, exec_capabilities E = exec_capabilities::svm_coarse_grain>
class svm_allocator {
 private:
  /* reference counted context for allocations */
  shared_ptr<context> m_context;

 public:
  typedef T value_type;
  typedef value_type *pointer;
  typedef const value_type *const_pointer;
  typedef value_type &reference;
  typedef const value_type &const_reference;
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;

  template <typename U>
  struct rebind {
    typedef svm_allocator<U, E> other;
  };

  /** Default allocator constructor, creates a context implicitly, by calling
   *  context(capability_selector(E));
   */
  svm_allocator();

  /** Explicit allocator constructor with given context.
   */
  explicit svm_allocator(context &openclContext);

  /* Copy constructor */
  svm_allocator(const svm_allocator &otherAlloc)

      /* Copy constructor */
      template <typename U>
      svm_allocator(const svm_allocator<U, E> &otherAlloc);

  /* Destructor */
  ~svm_allocator();

  /* Retrieve the address of the value reference */
  pointer address(reference r) { return std::addressof(r); }

  /* Retrieve the address of the const value reference */
  const_pointer address(const_reference r) { return std::addressof(r); }

  /**  Allocate an SVM pointer.
   * If the allocator is coarse-grained, this will take ownership to allow
   * containers to correctly construct data in place.
   */
  pointer allocate(size_t size,
                   typename svm_allocator<void, E>::const_pointer = 0);

  /* Deallocate an SVM pointer. */
  void deallocate(pointer p, size_t);
  /**
   * Return the maximum possible allocation size.
   * This is the minimum of the maximum sizes of all devices in the context.
   */
  size_t max_size() const;

  template <class U, class... Args>
  void construct(U *p, Args &&... args) {
    new (p) T(args...);
  }

  template <class U>
  void destroy(U *p) {
    p->~U();
  }

  /* Returns true if the contexts match. */
  inline bool operator==(svm_allocator const &rhs);

  /* Returns true if the contexts match. */
  inline bool operator!=(svm_allocator const &a);
};  // class svm_allocator

/* SVM Allocator providing a void pointer type in case this is needed
 * for container classes or deleter classes.
 */
template <exec_capabilities E>
class svm_allocator<void, E> {
 public:
  typedef void value_type;
  typedef value_type *pointer;
  typedef const value_type *const_pointer;

  template <typename U>
  struct rebind {
    typedef svm_allocator<U, E> other;
  };
};

}  // namespace svm
}  // namespace sycl
}  // namespace cl

#endif  // RUNTIME_INCLUDE_SYCL_SVM_H_
```

svm_allocator constructors	Description

`temlate<typename T, exec_capbilities E>`
`    svm_allocator()`
	Default constructor for creating an svm_allocator instance, using a default context constructor.
	The context constructor called will be taking as an argument a `capability_selector` with matching `cl::sycl::exec_capabilities` as the constructor. All allocations using this constructor will be using the same reference counted context. The SYCL kernels using these SVM pointer allocation should be using a queue created in the same context.
	The default value of exec_capabilities is `exec_capabilities::svm_coarse_grain` and in that respect if no other value is supplied, the default svm_allocator object can be used only in conjuction with the *buffer* class for SVM buffer allocations.

`temlate<typename T, typename ExecCapabilities>`
`    svm_allocator(context & contextInput)`
	Default constructor for creating an svm_allocator instance, using the contextInput.
	All allocations using this constructor will be using the same reference counted context. The SYCL kernels using these SVM pointer allocation should be using a queue created in the same context.
	Default value `ExecCapabilities = svm_coarse_grain` and the default svm_allocator object can be used only in conjuction with the *buffer* class for SVM buffer allocations.

7. SYCL kernel library

7.1 SYCL built-in functions for SYCL host and device

SYCL kernels may execute on any SYCL device- OpenCL device or SYCL host, which requires that the functions used in the kernels to be compiled and linked for both device and host. In the SYCL system the OpenCL built-ins are available for the SYCL host and device within the `cl::sycl` namespace, although, their semantics may be different. This section follows the OpenCL 1.2 specification document [1, ch. 6.12] and describes the behavior of these functions for SYCL host and device.

The SYCL built-in functions are available throughout the SYCL application, and depending where they execute, they are either implemented using their host implementation or the device implementation. The SYCL system guarantees that all of the built-in functions fulfill the same requirements for both host and device.

7.1.1 Description of the built-in types available for SYCL host and device

All the OpenCL built-in types are available in the namespace *cl::sycl*. For the purposes of this document we use type names for describing sets of SYCL valid types. The type names themselves are not valid SYCL types, but they represent a set of valid types, as defined in tables 7.1.

In the OpenCL 1.2 specification document [1, ch. 6.12.1] in Table 6.7 the work-item functions are defined where they provide the size of the enqueued kernel NDRange. These functions are available in SYCL through the item and group classes see sections 5.2.4, 5.2.5 and 5.2.6.

Generic type name	Description
`floatn`	`cl::sycl::float2, cl::sycl::float3, cl::sycl::float4, cl::sycl::float8, cl::sycl::float16`
`genfloatf`	`float, floatn`
`doublen`	`cl::sycl::double2, cl::sycl::double3 , cl::sycl::double4, cl::sycl:: double8, cl::sycl::double16`
`genfloatd`	`double, doublen`
`genfloat`	`float, floatn` `double, doublen`
`sgenfloat`	`float, double` It is a scalar type that matches the corresponding vector type *floatn* or *doublen*.

Continued on next page

Table 7.1: Generic type name description, which serves as a description for all valid types of parameters to kernel functions. [1]

Generic type name	Description
charn	cl::sycl::char2, cl::sycl::char3, cl::sycl::char4, cl::sycl::char8, cl::sycl::char16
ucharn	cl::sycl::uchar2, cl::sycl::uchar3, cl::sycl::uchar4, cl::sycl::uchar8, cl::sycl::uchar16
genchar	char, charn
ugenchar	unsigned char, ugenchar
shortn	cl::sycl::short2, cl::sycl::short3, cl::sycl::short4, cl::sycl::short8, cl::sycl::shor16
genshort	short, shortn
ushortn	cl::sycl::ushort2, cl::sycl::ushort3, cl::sycl::ushort4, cl::sycl::ushort8, cl::sycl::ushor16
ugenshort	unsigned short, ugenshort
uintn	cl::sycl::uint2, cl::sycl::uint3, cl::sycl::uint4, cl::sycl::uint8, cl::sycl::uint16
ugenint	int, uintn
intn	cl::sycl::int2, cl::sycl::int3, cl::sycl::int4, cl::sycl::int8, cl::sycl::int16
genint	int, intn
ulonglongn	cl::sycl::ulonglong2, cl::sycl::ulonglong3, cl::sycl::ulonglong4, cl::sycl::ulonglong8, cl::sycl::ulonglong16
ugenlonglong	unsigned long long int, ulonglongn
longlongn	cl::sycl::longlong2, cl::sycl::longlong3, cl::sycl::longlong4, cl::sycl::longlong8, cl::sycl::longlong16
genlonglong	long long int, longlongn
geninteger	genchar, ugenchar, genshort, ugenshort, genint, ugenint, genlonglong, ugenlonglong
sgeninteger	char,short, int, long long int, unsigned char, unsigned short, unsigned int, unsigned long long int
ugeninteger	uchar, ucharn, ushort, ushortn, uint, uintn, ulonglong, ulonglongn
gentype	char, charn, uchar, ucharn, short, shortn, ushort, ushortn, int, intn, uint, uintn, longlong, longlongn, ulonglong, ulonglongn, float, floatn, double, doublen.
	End of table

Table 7.1: Generic type name description, which serves as a description for all valid types of parameters to kernel functions. [1]

7.1.2 Work-item functions

In the OpenCL 1.2 specification document [1, ch. 6.12.1] in Table 6.7 the work-item functions are defined where they provide the size of the enqueued kernel NDRange. These functions are available in SYCL through the `nd_item` and `group` classes see section 5.2.5 and 5.2.6.

7.1.3 Math functions

In SYCL the OpenCL math functions are available in the namespace *cl::sycl* on host and device with the same precision guarantees as defined in the OpenCL 1.2 specification document [1, ch. 7] for host and device. For a SYCL platform the numerical requirements for host need to match the numerical requirements of the OpenCL math built-in functions. The built-in functions can take as input float or optionally double and their *vec* counterparts, for dimensions 2, 3, 4, 8 and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types.

The built-in functions available for SYCL host and device with the same precision requirements for both host and device, are described in table 7.2.

Math Function	Description
`genfloat acos ( genfloat x )`	Inverse cosine function.
`genfloat acosh ( genfloat x )`	Inverse hyperbolic cosine.
`genfloat acospi ( genfloat x )`	Compute $acos x/\pi$
`genfloat asin ( genfloat x )`	Inverse sine function.
`genfloat asinh ( genfloat x )`	Inverse hyperbolic sine.
`genfloat asinpi ( genfloat x)`	Compute $asin x/\pi$
`genfloat atan ( genfloat y_over_x )`	Inverse tangent function.
`genfloat atan2 ( genfloat y, genfloat x )`	Compute atan(y / x).
`genfloat atanh ( genfloat x )`	Hyperbolic inverse tangent.
`genfloat atanpi ( genfloat x )`	Compute atan (x) / π.
`genfloat atan2pi ( genfloat y, genfloat x )`	Compute atan2 (y, x) / π.
`genfloat cbrt ( genfloat x )`	Compute cube-root.
`genfloat ceil ( genfloat x )`	Round to integral value using the round to positive infinity rounding mode.
`genfloat copysign ( genfloat x, genfloat y )`	Returns x with its sign changed to match the sign of y.
`genfloat cos ( genfloat x )`	Compute cosine.
`genfloat cosh ( genfloat x )`	Compute hyperbolic cosine.
`genfloat cospi ( genfloat x )`	Compute cos (πx).
`genfloat erfc ( genfloat x )`	Complementary error function.
`genfloat erf ( genfloat x )`	Error function encountered in integrating the normal distribution.
`genfloat exp ( genfloat x )`	Compute the base-e exponential of x.
`genfloat exp2 ( genfloat x )`	Exponential base 2 function.
`genfloat exp10 ( genfloat x )`	Exponential base 10 function.
`genfloat expm1 ( genfloat x )`	Compute $\exp(x) - 1.0$.

Continued on next page

Table 7.2: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1]

Math Function	Description
`genfloat fabs ( genfloat x)`	Compute absolute value of a floating-point number.
`genfloat fdim ( genfloat x, genfloat y )`	$x - y$ if $x > y$,+0 if x is less than or equal to y.
`genfloat floor (genfloat x)`	Round to integral value using the round to negative infinity rounding mode.
`genfloat fma (genfloat a, genfloat b, genfloat c)`	Returns the correctly rounded floating-point representation of the sum of c with the infinitely precise product of a and b. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.

`genfloat fmax ( genfloat x, genfloat y )`
`genfloat fmax ( genfloat x, sgenfloat y )`
	Returns y if $x < y$, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN.

`genfloat fmin ( genfloat x, genfloat y )`
`genfloat fmin ( genfloat x, sgenfloat y )`
	Returns y if $y < x$, otherwise it returns x. If one argument is a NaN, fmin() returns the other argument. If both arguments are NaNs, fmin() returns a NaN.

`genfloat fmod ( genfloat x, genfloat y )`	Modulus. Returns $xy * trunc(x/y)$.

`floatn fract ( floatn x, intn *iptr )`
`float fract ( float x, int * iptr )`
	Returns fmin(x – floor (x), 0x1.fffffep-1f). floor(x) is returned in iptr.

`doublen frexp ( doublen x, intn *exp )`
`double frexp ( double x, int * exp )`
	Extract mantissa and exponent from x. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of x equals mantissa returned * 2exp.

`genfloat hypot ( genfloat x, genfloat y )`	Compute the value of the square root of x2+ y2 without undue overflow or underflow.

`int logb ( float x )`
`intn ilogb ( genfloat x )`
`int logb ( double x )`
`intn logb ( doublen x )`
	Return the exponent as an integer value.

`genfloat ldexp (genfloat x, genint k)`
`floatn ldexp (floatn x, int k)`
`doublen ldexp (doublen x, int k)`
	Multiply x by 2 to the power k.

`genfloat lgamma ( genfloat x )`	Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of lgamma_r.
`genfloat lgamma_r ( genfloat x, genint *signp )`	Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of lgamma_r.

Continued on next page

Table 7.2: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1]

Math Function	**Description**
`genfloat log (genfloat)`	Compute natural logarithm.
`genfloat log2 (genfloat)`	Compute a base 2 logarithm.
`genfloat log10 (genfloat)`	Compute a base 10 logarithm.
`genfloat log1p (genfloat x)`	Compute $loge(1.0 + x)$.
`genfloat logb (genfloat x)`	Compute the exponent of x, which is the integral part of logr ($\|x\|$).
`genfloat mad (genfloat a,genfloat b, genfloat c)`	mad approximates a * b + c. Whether or how the product of a * b is rounded and how supernormal or subnormal intermediate products are handled is not defined. mad is intended to be used where speed is preferred over accuracy.
`genfloat maxmag ( genfloat x, genfloat y )`	Returns x if $\|x\| > \|y\|$, y if $\|y\| > \|x\|$, otherwise fmax(x, y).
`genfloat minmag (genfloat x, genfloat y)`	Returns x if $\|x\| < \|y\|$, y if $\|y\| < \|x\|$, otherwise fmin(x, y).
`genfloat modf (genfloat x, genfloat *iptr)`	Decompose a floating-point number. The modf function breaks the argument x into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by iptr.

`floatn nan ( unintn nancode )`
`float nan (unsigned int nancode )`
`doublen nan (ulonglongn nancode )`
`double nan (unsigned long long int nancode )`
	Returns a quiet NaN. The nancode may be placed in the significand of the resulting NaN.

`genfloat nextafter (genfloat x, genfloat y)`	Computes the next representable single-precision floating-point value following x in the direction of y. Thus, if y is less than x, nextafter() returns the largest representable floating-point number less than x.
`genfloat pow (genfloat x, genfloat y)`	Compute x to the power y.
`genfloat pown (genfloat x, genint y)`	Compute x to the power y, where y is an integer.
`genfloat powr (genfloat x, genfloat y)`	Compute x to the power y, where $x >= 0$.
`genfloat remainder (genfloat x, genfloat y)`	Compute the value r such that r = x - n*y, where n is the integer nearest the exact value of x/y. If there are two integers closest to x/y, n shall be the even one. If r is zero, it is given the same sign as x.

Continued on next page

Table 7.2: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1]

Math Function	Description
`genfloat remquo (genfloat x, genfloat y, genint *quo )`	The remquo function computes the value r such that r = x - k*y, where k is the integer nearest the exact value of x/y. If there are two integers closest to x/y, k shall be the even one. If r is zero, it is given the same sign as x. This is the same value that is returned by the remainder function. remquo also calculates the lower seven bits of the integral quotient x/y, and gives that value the same sign as x/y. It stores this signed value in the object pointed to by quo.
`genfloat rint (genfloat)`	Round to integral value (using round to nearest even rounding mode) in floating-point format. Refer to section 7.1 of the OpenCL 1.2 specification document [1] for description of rounding modes.
`genfloat rootn (genfloat x, genint y)`	Compute x to the power 1/y.
`genfloat round (genfloat x)`	Return the integral value nearest to x rounding halfway cases away from zero, regardless of the current rounding direction.
`genfloat rsqrt (genfloat)`	Compute inverse square root.
`genfloat sin (genfloat)`	Compute sine.
`genfloat sincos (genfloat x, genfloat *cosval)`	Compute sine and cosine of x. The computed sine is the return value and computed cosine is returned in cosval.
`genfloat sinh ( genfloat x )`	Compute hyperbolic sine.
`genfloat sinpi ( genfloat x)`	Compute sin (π x).
`genfloat sqrt ( genfloat x )`	Compute square root.
`genfloat tan ( genfloat x )`	Compute tangent.
`genfloat tanh ( genfloat x )`	Compute hyperbolic tangent.
`genfloat tanpi ( genfloat x)`	Compute tan (π x).
`genfloat tgamma ( genfloat x )`	Compute the gamma function.
`genfloat trunc ( genfloat x )`	Round to integral value using the round to zero rounding mode.
	End of table

Table 7.2: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1]

In SYCL the implementation defined precision math functions are defined in the namespace *cl::sycl::native*. The functions that are available within this namespace are specified in tables 7.3.

Native Math Function	Description
`genfloat cos (genfloat x)`	Compute cosine over an implementation-defined range. The maximum error is implementation-defined.
	Continued on next page

Table 7.3: Native functions which work on SYCL Host and device, are available in the *cl::sycl::native* namespace. They correspond to Table 6.9 of the OpenCL 1.2 specification [1]

Native Math Function	**Description**
`genfloat divide (genfloat x, genfloat y)`	Compute x / y over an implementation-defined range. The maximum error is implementation-defined.
`genfloat exp (genfloat x)`	Compute the base- e exponential of x over an implementation-defined range. The maximum error is implementation-defined.
`genfloat exp2 (genfloat x)`	Compute the base- 2 exponential of x over an implementation-defined range. The maximum error is implementation-defined.
`genfloat exp10 (genfloat x)`	Compute the base- 10 exponential of x over an implementation-defined range. The maximum error is implementation-defined.
`genfloat log (genfloat x)`	Compute natural logarithm over an implementation defined range. The maximum error is implementation-defined.
`genfloat log2 (genfloat x)`	Compute a base 2 logarithm over an implementation-defined range. The maximum error is implementation-defined.
`genfloat log10 (genfloat x)`	Compute a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined.
`genfloat powr (genfloat x, genfloat y)`	Compute x to the power y, where $x >= 0$. The range of x and y are implementation-defined. The maximum error is implementation-defined.
`genfloat recip (genfloat x)`	Compute reciprocal over an implementation-defined range. The maximum error is implementation-defined.
`genfloat rsqrt (genfloat x)`	Compute inverse square root over an implementation-defined range. The maximum error is implementation-defined.
`genfloat sin ( genfloat x)`	Compute sine over an implementation-defined range. The maximum error is implementation-defined.
`genfloat sqrt ( genfloat x )`	Compute square root over an implementation-defined range. The maximum error is implementation-defined.
`genfloat tan ( genfloat x)`	Compute tangent over an implementation-defined range. The maximum error is implementation-defined.
	End of table

Table 7.3: Native functions which work on SYCL Host and device, are available in the *cl::sycl::native* namespace. They correspond to Table 6.9 of the OpenCL 1.2 specification [1]

7.1.4 Integer functions

In SYCL the OpenCL integer math functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.3]. The built-in functions can take as input char, unsigned char, short, unsigned short, int, unsigned int, long long int, unsigned long long int and their *vec* counterparts, for dimensions 2, 3, 4, 8 and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types. The supported integer math functions are described in tables 7.4.

Integer Function	Description
`ugeninteger abs (geninteger x)`	Returns $\|x\|$.
`ugeninteger abs_diff (geninteger x, geninteger y)`	Returns $\|x - y\|$ without modulo overflow.
`geninteger add_sat (geninteger x, geninteger y)`	Returns $x + y$ and saturates the result.
`geninteger hadd (geninteger x, geninteger y)`	Returns $(x+y) >> 1$. The intermediate sum does not modulo overflow.
`geninteger rhadd (geninteger x, geninteger y)`	Returns $(x + y + 1) >> 1$. The intermediate sum does not modulo overflow.
`geninteger clamp (geninteger x, sgeninteger minval, sgeninteger maxval)`	Returns min(max(x, minval), maxval). Results are undefined if minval > maxval.
`geninteger clz (geninteger x)`	Returns the number of leading 0-bits in x, starting at the most significant bit position.
`geninteger clamp (geninteger x, geninteger minval, geninteger maxval)`	Returns min(max(x, minval), maxval). Results are undefined if minval > maxval.
`geninteger mad_hi ( geninteger a, geninteger b, geninteger c)`	Returns mul_hi(a, b) + c.
`geninteger mad_sat (geninteger a, geninteger b, geninteger c)`	Returns a * b + c and saturates the result.

`geninteger max (geninteger x, geninteger y)`
`geninteger max (geninteger x, sgeninteger y)`
	Returns y if $x < y$, otherwise it returns x.

`geninteger min (geninteger x, geninteger y)`
`geninteger min (geninteger x, sgeninteger y)`
	Returns y if $y < x$, otherwise it returns x.

`geninteger mul_hi (geninteger x, geninteger y)`	Computes x * y and returns the high half of the product of x and y.
`geninteger rotate (geninteger v, geninteger i)`	For each element in v, the bits are shifted left by the number of bits given by the corresponding element in i (subject to usual shift modulo rules described in section 6.3). Bits shifted off the left side of the element are shifted back in from the right.
`geninteger sub_sat (geninteger x, geninteger y)`	Returns $x - y$ and saturates the result.
`shortn upsample (charn hi, ucharn lo)`	$result[i] = ((short)hi[i] << 8)\|lo[i]$
`ushortn upsample (ucharn hi, ucharn lo)`	$result[i] = ((ushort)hi[i] << 8)\|lo[i]$
`intn upsample (shortn hi, ushortn lo)`	$result[i] = ((int)hi[i] << 16)\|lo[i]$
`uintn upsample (ushortn hi, ushortn lo)`	$result[i] = ((uint)hi[i] << 16)\|lo[i]$
`longlongn upsample (intn hi, uintn lo)`	$result[i] = ((long)hi[i] << 32)\|lo[i]$
`ulonglongn upsample (uintn hi, uintn lo)`	$result[i] = ((ulong)hi[i] << 32)\|lo[i]$
`geninteger popcount (geninteger x)`	Returns the number of non-zero bits in x.

Continued on next page

Table 7.4: Integer functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.10 of the OpenCL 1.2 specification [1]

Integer Function	**Description**
`intn mad24 (intn x, intn y, intn z)` `uintn mad24 (uintn x, uintn y, uintn z)`	Multipy two 24-bit integer values x and y and add the 32-bit integer result to the 32-bit integer z. Refer to definition of mul24 to see how the 24-bit integer multiplication is performed.
`intn mul24 ( intn x, intn y )` `uintn mul24 ( uintn x, uintn y )`	Multiply two 24-bit integer values x and y. x and y are 32-bit integers but only the low 24-bits are used to perform the multiplication. mul24 should only be used when values in x and y are in the range [- 223, 223-1] if x and y are signed integers and in the range [0, 224-1] if x and y are unsigned integers. If x and y are not in this range, the multiplication result is implementation-defined.
	End of table

Table 7.4: Integer functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.10 of the OpenCL 1.2 specification [1]

7.1.5 Common functions

In SYCL the OpenCL *common* functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.4]. Description is in table 7.5. The built-in functions can take as input float or optionally double and their *vec* counterparts, for dimensions 2, 3, 4, 8 and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types.

Common Function	**Description**
`genfloat clamp ( genfloat x, genfloat minval, genfloat maxval )` `floatn clamp ( floatn x, float minval, float maxval )` `doublen clamp ( doublen x, double minval, double maxval )`	Returns fmin(fmax(x, minval), maxval). Results are undefined if $minval > maxval$.
`genfloat degrees (genfloat radians)`	Converts radians to degrees, i.e.$(180/\pi) * radians$.
`genfloat max ( genfloat x, genfloat y)` `genfloatf max (genfloatf x, float y)` `genfloatd max ( genfloatd x, double y )`	Returns y if $x < y$, otherwise it returns x. If x or y are infinite or NaN, the return values are undefined.
`genfloat min ( genfloat x, genfloat y )` `genfloatf min ( genfloatf x, float y )` `genfloatd min ( genfloatd x, double y )`	Returns y if $y < x$, otherwise it returns x. If x or y are infinite or NaN, the return values are undefined.
	Continued on next page

Table 7.5: Common functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1]

Common Function	Description
`genfloat mix ( genfloat x, genfloat y, genfloat a )` `genfloatf mix ( genfloatf x, genfloatf y, float a )` `genfloatd mix ( genfloatd x, genfloatd y, double a )`	Returns the linear blend of $x\&y$ implemented as: $x+(y-x)*a$. a must be a value in the range 0.0 ... 1.0. If a is not in the range 0.0 ... 1.0, the return values are undefined.
`genfloat radians (genfloat degrees)`	Converts degrees to radians, i.e. $(\pi/180) * degrees$.
`genfloat step (genfloat edge, genfloat x)` `genfloatf step (float edge, genfloatf x)` `genfloatd step (double edge, genfloatd x)`	Returns 0.0 if $x < edge$, otherwise it returns 1.0.
`genfloat smoothstep (genfloat edge0, genfloat edge1, genfloat x)` `genfloatf smoothstep (float edge0, float edge1, genfloatf x)` `genfloatd smoothstep (double edge0, double edge1 , genfloatd x)`	Returns 0.0 if $x <= edge0$ and 1.0 if $x >= edge1$ and performs smooth Hermite interpolation between 0 and 1 when $edge0 < x < edge1$. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to: `gentype t;` `t = clamp ((x <= edge0)/ (edge1 >= edge0), 0, 1);` `return t * t * (3 - 2 * t);` Results are undefined if $edge0 >= edge1$ or if x, edge0 or edge1 is a NaN.
`genfloat sign (genfloat x)`	Returns 1.0 if $x > 0$, -0.0 if $x = -0.0$, $+0.0$ if $x = +0.0$, or -1.0 if $x < 0$. Returns 0.0 if x is a NaN.
	End of table

Table 7.5: Common functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1]

7.1.6 Geometric Functions

In SYCL the OpenCL *geometric* functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.5]. The built-in functions can take as input float or optionally double and their *vec* counterparts, for dimensions 2, 3, 4, 8 and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types. All of the geometric functions are using round-to-nearest-even rounding mode. Tables 7.6 contain all the definitions of supported geometric functions.

Geometric Function	Description
`float4 cross (float4 p0, float4 p1)` `float3 cross (float3 p0, float3 p1)` `double4 cross (double4 p0, double4 p1)` `double3 cross (double3 p0, double3 p1)`	Returns the cross product of p0.xyz and p1.xyz. The w component of float4 result returned will be 0.0.
	Continued on next page

Table 7.6: Geometric functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1]

Geometric Function	Description

`float dot (floatn p0, floatn p1)`
`double dot (doublen p0, doublen p1)`
	Compute dot product.

`float distance (floatn p0, floatn p1)`
`double distance (doublen p0, doublen p1)`
	Returns the distance between p0 and p1. This is calculated as $length(p0 - p1)$.

`float length (floatn p)`
`double length (doublen p)`
	Return the length of vector p, i.e., $\sqrt{p.x^2 + p.y^2 + ...}$

`floatn normalize (floatn p)`
`doublen normalize (doublen p)`
	Returns a vector in the same direction as p but with a length of 1.

`float fast_distance (floatn p0, floatn p1)`	Returns $fast_length(p0 - p1)$.
`float fast_length (floatn p)`	Returns the length of vector p computed as: `sqrt((half)(pow(p.x,2)+ pow(p.y,2) + ...))`

`floatn fast_normalize (floatn p)`	Returns a vector in the same direction as p but with a length of 1. fast_normalize is computed as: `p * rsqrt((half)(pow(p.x ,2)+ pow(p.y,2)+ ... ))`
	The result shall be within 8192 ulps error from the infinitely precise result of `if (all (p == 0.0f))`
	`result = p;`
	`else`
	`result = p / sqrt (pow(p.x,2)+ pow(p. y,2)+ ... );`
	with the following exceptions:
	1. If the sum of squares is greater than FLT_MAX then the value of the floating-point values in the result vector are undefined.
	2. If the sum of squares is less than FLT_MIN then the implementation may return back p.
	3. If the device is in "denorms are flushed to zero" mode, individual operand elements with magnitude less than $sqrt(FLT_MIN)$ may be flushed to zero before proceeding with the calculation.

	End of table

Table 7.6: Geometric functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1]

7.1.7 Relational functions

In SYCL the OpenCL *relational* functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.6]. The built-in functions can take as input char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float or optionally double and their *vec* counterparts, for dimensions 2,3,4,8, and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types. The relational operators are available in both host and device, these relational functions are provided in addition to the the operators and will return 0 if the conditional is *false* and 1 otherwise. The available built-in functions are described in tables 7.7

Relational Function	Description

`int isequal (float x, float y)`
`intn isequal (floatn x, floatn y)`
`longlong isequal (double x, double y)`
`longlongn isequal (doublen x, doublen y)`
	Returns the component-wise compare of $x == y$.

`int isnotequal (float x, float y)`
`intn isnotequal (floatn x, floatn y)`
`longlong isnotequal (double x, double y)`
`longlongn isnotequal (doublen x, doublen y)`
	Returns the component-wise compare of $x! = y$.

`int isgreater (float x, float y)`
`intn isgreater (floatn x, floatn y)`
`longlong isgreater (double x, double y)`
`longlongn isgreater (doublen x, doublen y)`
	Returns the component-wise compare of $x > y$.

`int isgreaterequal (float x, float y)`
`intn isgreaterequal (floatn x, floatn y)`
`longlong isgreaterequal (double x, double y)`
`longlongn isgreaterequal (doublen x, doublen y)`
	Returns the component-wise compare of $x >= y$.

`int isless (float x, float y)`
`intn isless (floatn x, floatn y)`
`longlong isless (double x, double y)`
`longlongn isless (doublen x, doublen y)`
	Returns the component-wise compare of $x < y$.

`int islessequal (float x, float y)`
`intn islessequal (floatn x, floatn y)`
`longlong islessequal (double x, double y)`
`longlongn islessequal (doublen x, doublen y)`
	Returns the component-wise compare of $x <= y$.

`int islessgreater (float x, float y)`
`intn islessgreater (floatn x, floatn y)`
`longlong islessgreater (double x, double y)`
`longlongn islessgreater (doublen x, doublen y)`
	Returns the component-wise compare of $(x < y) \|\| (x > y)$.

`int isfinite (float)`
`intn isfinite (floatn)`
`longlong isfinite (double)`
`longlongn isfinite (doublen)`
	Test for finite value.

`int isinf (float)`
`intn isinf (floatn)`
`longlong isinf (double)`
`longlongn isinf (doublen)`
	Test for infinity value (positive or negative) .

Continued on next page

Table 7.7: Relational functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1]

Relational Function	Description

`int isnan (float)`
`intn isnan (floatn)`
`longlong isnan (double)`
`longlongn isnan (doublen)`
	Test for a NaN.

`int isnormal (float)`
`intn isnormal (floatn)`
`longlong isnormal (double)`
`longlongn isnormal (doublen)`
	Test for a normal value.

`int isordered (float x, float y)`
`intn isordered (floatn x, floatn y)`
`longlong isordered (double x, double y)`
`longlongn isordered (doublen x, doublen y)`
	Test if arguments are ordered. isordered() takes arguments x and y, and returns the result isequal(x, x) && isequal(y, y).

`int isunordered (float x, float y)`
`intn isunordered (floatn x, floatn y)`
`longlong isunordered (double x, double y)`
`longlongn isunordered (doublen x, doublen y)`
	Test if arguments are unordered. isunordered() takes arguments x and y, returning non-zero if x or y is NaN, and zero otherwise.

`int signbit (float)`
`intn signbit (floatn)`
`longlong signbit (double)`
`longlongn signbit (doublen)`
	Test for sign bit. The scalar version of the function returns a 1 if the sign bit in the float is set else returns 0.
	The vector version of the function returns the following for each component in *floatn*: -1 (i.e all bits set) if the sign bit in the float is set else returns 0.

`int any (geninteger x)`	Returns 1 if the most significant bit in any component of x is set; otherwise returns 0.
`int all (ugeninteger x)`	Returns 1 if the most significant bit in all components of x is set; otherwise returns 0.
`gentype bitselect (gentype a, gentype b, gentype c)`	Each bit of the result is the corresponding bit of a if the corresponding bit of c is 0. Otherwise it is the corresponding bit of b.

`gentype select (gentype a, gentype b, geninteger c)`	For each component of a vector type:
	`result[i] = (MSB of c[i] is set)? b[i ] : a[i].`
	For a scalar type:
	`result = c ? b : a.`
	`geninteger` must have the same number of elements and bits as `gentype`.

	End of table

Table 7.7: Relational functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1]

7.1.8 Vector data and store functions

The functionality from the OpenCL functions as defined in in the OpenCL 1.2 specification document [1, par. 6.12.7] is available in SYCL through the *vec* class in section 4.10.2.

7.1.9 Synchronization Functions

In SYCL the OpenCL *synchronization* functions are available through the item class 5.2.4, as they are applied to work-item for local or global address spaces. Please see 5.9.

7.1.10 `printf` function

The functionality of the `printf` function is covered by the `cl::cycle::stream` class 7.4, which has the capability to print to standard output all the SYCL classes and primitives and covers the capabilities defined in the OpenCL 1.2 specification document [1, par. 6.12.13].

7.2 Synchronization and atomics

The SYCL specification offers the same set of synchronization operations that are available to OpenCL C programs, for compatibility and portability across OpenCL devices. The available features are:

- Accessor classes: Accessor classes specify acquisition and release of buffer and image data structures to provide points at which underlying queue synchronization primitives must be generated.
- Atomic operations: OpenCL 1.2 devices only support the equivalent of relaxed C++ atomics and SYCL uses the C++11 library syntax to make this available. This is provided for forward compatibility with future SYCL versions.
- Barriers: Barrier primitives are made available to synchronize sets of work-items within individual work-groups. They are exposed through the `nd_item` class that abstracts the current point in the overall iteration space.
- Hierarchical parallel dispatch: In the hierarchical parallelism model of describing computations, synchronization within the work-group is made explicit through multiple instances of the `parallel_for_work_item` function call, rather than through the use of explicit barrier operations.

Barriers may provide ordering semantics over the local address space, global address space or both. All memory operations initiated before the barrier in the specified address space(s) will be completed before any memory operation after the barrier. Address spaces are described using the `fence_space` enum class:

```
namespace cl {
namespace sycl {
namespace access {
enum class fence_space : char {
  local_space,
  global_space,
  global_and_local
};  // enum class address_space
}  // namepaces access
}  // namespace sycl
}  // namespace cl
```

The SYCL specification provides atomic operations based on the C++11 library syntax. The only available ordering, due to constraints of the OpenCL 1.2 memory model, is `memory_order_relaxed`. No default order is supported because a default order would imply sequential consistency. The SYCL atomic library may map directly to the underlying C++11 library in host code, and must interact safely with the host C++11 atomic library when used in host code. The SYCL library must be used in device code to ensure that only the limited subset of functionality is available. SYCL 1.2 device compilers should give a compilation error on use of the `std::atomic` classes and functions in device code. Only `atomic<int>`, `atomic<unsigned int>` and `atomic<float>` types are available in SYCL 1.2. Only the `exchange` operation is available for float atomics.

No construction of atomic objects is possible in SYCL 1.2. All atomic objects must be obtained by-reference from an accessor (see Section 4.7.0.3).

The atomic types are defined as follows, and methods are listed in Table 7.8:

```
namespace cl {
namespace sycl {
template <typename T>
class atomic<T> {
 public:
  // Constructors
  atomic() = delete;

  // Methods
  // Only memory_order_relaxed is supported in SYCL 1.2
  void store(T operand, std::memory_order = std::memory_order_relaxed);
  void store(T operand, std::memory_order = std::memory_order_relaxed) volatile;
  T load(memory_order = std::memory_order_relaxed) const;
  T load(memory_order = std::memory_order_relaxed) const volatile;

  T exchange(T operand, std::memory_order = std::memory_order_relaxed);
  T exchange(T operand, std::memory_order = std::memory_order_relaxed) volatile;

  T compare_exchange_strong(T* expected, T desired, std::memory_order success,
                            std::memory_order fail);
  T compare_exchange_strong(T* expected, T desired, std::memory_order success,
                            std::memory_order fail) volatile;

  T fetch_add(T operand, std::memory_order = std::memory_order_relaxed);
  T fetch_add(T operand,
            std::memory_order = std::memory_order_relaxed) volatile;

  T fetch_sub(T operand, std::memory_order = std::memory_order_relaxed);
  T fetch_sub(T operand,
            std::memory_order = std::memory_order_relaxed) volatile;

  T fetch_and(T operand, std::memory_order = std::memory_order_relaxed);
  T fetch_and(T operand,
            std::memory_order = std::memory_order_relaxed) volatile;

  T fetch_or(T operand, std::memory_order = std::memory_order_relaxed);
  T fetch_or(T operand, std::memory_order = std::memory_order_relaxed) volatile;

  T fetch_xor(T operand, std::memory_order = std::memory_order_relaxed);
  T fetch_xor(T operand,
            std::memory_order = std::memory_order_relaxed) volatile;

  // Additional functionality provided beyond that of C++11
  T fetch_min(T operand, std::memory_order = std::memory_order_relaxed);
  T fetch_min(T operand,
            std::memory_order = std::memory_order_relaxed) volatile;

  T fetch_max(T operand, std::memory_order = std::memory_order_relaxed);
  T fetch_max(T operand,
            std::memory_order = std::memory_order_relaxed) volatile;
};

typedef atomic<int> atomic_int;
typedef atomic<unsigned int> atomic_uint;
typedef atomic<float> atomic_float;
}  // namespace sycl
}  // namespace cl
```

As well as the methods, a matching set of operations on atomic types is provided by the SYCL library. As in the previous case, the only available memory order in SYCL 1.2 is `memory_order_relaxed`. The global functions are as follows and described in Table 7.9.

```
namespace cl {
namespace sycl {
template <class T>
T atomic_load_explicit(atomic<T>* object,
                       std::memory_order = std::memory_order_relaxed);
template <class T>
T atomic_load_explicit(volatile atomic<T>* object,
                       std::memory_order = std::memory_order_relaxed);

template <class T>
void atomic_store_explicit(atomic<T>* object, T,
                           std::memory_order = std::memory_order_relaxed);
template <class T>
void atomic_store_explicit(volatile atomic<T>* object, T operand,
                           std::memory_order = std::memory_order_relaxed);

template <class T>
T atomic_exchange_explicit(atomic<T>* object, T operand,
                           std::memory_order = std::memory_order_relaxed);
template <class T>
T atomic_exchange_explicit(volatile atomic<T>* object, T,
                           std::memory_order = std::memory_order_relaxed);

template <class T>
bool atomic_compare_exchange_strong_explicit(atomic<T>* object, T* expected,
                                             T desired,
                                             std::memory_order success,
                                             std::memory_order fail);
template <class T>
bool atomic_compare_exchange_strong_explicit(volatile atomic<T>*, T*, T,
                                             std::memory_order success,
                                             std::memory_order fail);
template <class T>
T atomic_fetch_add_explicit(atomic<T>* object, T operand,
                            std::memory_order = std::memory_order_relaxed);
template <class T>
T atomic_fetch_add_explicit(volatile atomic<T>* object, T operand,
                            std::memory_order = std::memory_order_relaxed);

template <class T>
T atomic_fetch_sub_explicit(atomic<T>* object, T operand,
                            std::memory_order = std::memory_order_relaxed);
template <class T>
T atomic_fetch_sub_explicit(volatile atomic<T>* object, T operand,
                            std::memory_order = std::memory_order_relaxed);

template <class T>
T atomic_fetch_and_explicit(atomic<T>* object, T operand,
                            std::memory_order = std::memory_order_relaxed);
template <class T>
T atomic_fetch_and_explicit(volatile atomic<T>* object, T operand,
                            std::memory_order = std::memory_order_relaxed);

template <class T>
T atomic_fetch_or_explicit(atomic<T>*, T, std::memory_order);
template <class T>
T atomic_fetch_or_explicit(volatile atomic<T>* object, T operand,
                           std::memory_order = std::memory_order_relaxed);

template <class T>
T atomic_fetch_xor_explicit(atomic<T>* object, T operand,
                            std::memory_order = std::memory_order_relaxed);
template <class T>
T atomic_fetch_xor_explicit(volatile atomic<T>*, T,
                            std::memory_order = std::memory_order_relaxed);

// Additional functionality beyond that provided by C++11
template <class T>
T atomic_fetch_min_explicit(atomic<T>* object, T operand,
                            std::memory_order = std::memory_order_relaxed);
template <class T>
T atomic_fetch_min_explicit(volatile atomic<T>* object, T,
                            std::memory_order = std::memory_order_relaxed);

template <class T>
T atomic_fetch_max_explicit(atomic<T>* object, T operand, std::memory_order);
template <class T>
T atomic_fetch_max_explicit(volatile atomic<T>* object, T operand,
                            std::memory_order);
}  // namespace sycl
}  // namespace cl
```

The atomic operations and methods behave as described in the C++11 specification, barring the restrictions discussed above. Note that car must be taken when using `compare_exchange_strong` to perform many of the operations that would be expected of it in standard CPU code due to the lack of forward progress guarantees between work-items in SYCL. No work-item may be dependent on another work-item to make progress if the code is to be portable.

Methods	**Description**
`void store( T operand, std::memory_order = std::memory_order_relaxed );`	Atomically store operand in `*this`. T may be int, unsigned int or float. order must be `memory_order_relaxed`.
`T load( memory_order = std::memory_order_relaxed ) const;`	Atomically load the current value of `*this` and return the value before the call. T may be int, unsigned int or float. order must be `memory_order_relaxed`.
`T exchange( T operand, std::memory_order = std::memory_order_relaxed );`	Atomically replace `*this` with operand. Return the original value of object. T may be int, unsigned int or float. order must be `memory_order_relaxed`.
`T compare_exchange_strong( T& expected, T desired, std::memory_order success, std::memory_order fail );`	Atomically compare the value of `*this` against `expected`. If equal replace `*this` with `desired` otherwise store the original value of `*this` in `*expected`. Returns true if the comparison succeeded. Both memory orders must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_add( T operand, std::memory_order = std::memory_order_relaxed );`	Atomically add operand to `*this`. Store the result in `*this` and return the value before the call. order must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_sub( T operand, std::memory_order order );`	Atomically subtract operand from `*this` . Store the result in `*this` and return the value before the call. order must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_and( T operand, std::memory_order order );`	Atomically perform a bitwise and of operand and `*this`. Store the result in `*this` and return the value before the call. order must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_or( T operand, std::memory_order order );`	Atomically perform a bitwise or of operand and `*this`. Store the result in `*this` and return the value before the call. order must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_xor( T operand, std::memory_order order );`	Atomically perform a bitwise exclusive-or of operand and `*this`. Store the result in `*this` and return the value before the call. order must be `memory_order_relaxed`. T must be int or unsigned int.

Continued on next page

Table 7.8: Methods available on an object of type atomic<T>.

Methods	Description
`T fetch_min( T operand, std::memory_order order );`	Atomically compute the minimum of operand and *this. Store the result in *this and return the value before the call. order must be memory_order_relaxed. T must be int or unsigned int.
`T fetch_max( T operand, std::memory_order order );`	Atomically compute the maximum of operand and *this. Store the result in *this and return the value before the call. order must be memory_order_relaxed. T must be int or unsigned int.
	End of table

Table 7.8: Methods available on an object of type atomic<T>.

Functions	Description
`template<class T> T atomic_load_explicit( atomic<T>* object, std::memory_order order);`	Atomically load the current value of object and return that value. T may be int, unsigned int or float. order must be memory_order_relaxed.
`template<class T> void atomic_store_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically store operand in object. T may be int, unsigned int or float. order must be memory_order_relaxed.
`template<class T> T atomic_exchange_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically replace object with operand. Return the original value of object. T may be int, unsigned int or float. order must be memory_order_relaxed.
`template<class T> bool atomic_compare_exchange_strong_explicit( atomic<T>* object, T* expected, T desired, std::memory_order success, std::memory_order fail);`	Atomically compare the value of object against expected. If equal replace object with desired. Otherwise store the original value of object in expected. Returns true if the comparison succeeded. Both memory orders must be memory_order_relaxed. T must be int or unsigned int.
`template<class T> T atomic_fetch_add_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically add operand to object. Store the result in object. order must be memory_order_relaxed. T must be int or unsigned int.
`template<class T> T atomic_fetch_sub_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically subtract operand from object. Store the result in object. order must be memory_order_relaxed. T must be int or unsigned int.
	Continued on next page

Table 7.9: Global functions available on atomic types.

Functions	Description

`template<class T> T atomic_fetch_and_explicit(`
`    atomic<T>* operand,`
`    T object,`
`    std::memory_order order);`
	Atomically perform a bitwise and of `operand` and `object`. Store the result in `object`.
	`order` must be `memory_order_relaxed`. T must be int or unsigned int.

`template<class T> T atomic_fetch_or_explicit(`
`    atomic<T>* object,`
`    T operand,`
`    std::memory_order order);`
	Atomically perform a bitwise or of `operand` and `object`. Store the result in `object`.
	`order` must be `memory_order_relaxed`. T must be int or unsigned int.

`template<class T> T atomic_fetch_xor_explicit(`
`    atomic<T>* object,`
`    T operand,`
`    std::memory_order order);`
	Atomically perform a bitwise exclusive-or of `operand` and `object`. Store the result in `object`.
	`order` must be `memory_order_relaxed`. T must be int or unsigned int.

`template<class T> T atomic_fetch_min_explicit(`
`    atomic<T>* object,`
`    T operand,`
`    std::memory_order order);`
	Atomically compute the minimum of `operand` and `object`. Store the result in `object`.
	`order` must be `memory_order_relaxed`. T must be int or unsigned int.

`template<class T> T atomic_fetch_max_explicit(`
`    atomic<T>* object,`
`    T operand,`
`    std::memory_order order);`
	Atomically compute the maximum of `operand` and `object`. Store the result in `object`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.

	End of table

Table 7.9: Global functions available on atomic types.

7.3 Device-side enqueue Interface

Device-side enqueue is supported for OpenCL 2.2 systems and supports a mode which in OpenCL terms is called *nested parallelism*. This mode allows kernels to be sumbitted from other kernels without the need of host synchronization. The capabilities of device-enqueue matches the capabilities of host-enqueue, with the difference that on the device, some SYCL runtime features for enqueuing kernels are not available. Those features are: defining accessors within the device command group; being able to track dependencies among the submitted kernels, and C++ exceptions for reporting errors from the queue.

Execution of kernels on the device in SYCL is done using a `device_queue`. The `device_queue` can submit device command groups, i.e. command group functors instantiated on the device. There can be no exceptions thrown from the `device_queue` and as a result there are no `wait_and_throw()` or `throw()` methods available for that class, as described in 7.3.1.

All device command groups can only be called with an instance of a `device_handle`. The `device_handle` supports all the SYCL kernel invocations (such as `parallel_for`) with both C++ lambdas and functors being usable to define the kernels. The device-enqueued kernels must be defined in the same C++ translation unit as the kernels enqueued from host so that the device compiler can construct the kernel instances for the SYCL device. The `device_handle` does not provide support for constructing new accessors, as the methods for providing access or registering access to host memory are only available in command groups executed on the host. However, accessors constructed on the host can be captured as parameters to the device enqueued kernel. The `device_handle` class is described in detail in 7.3.2.

In order to provide scheduling information for the device kernels, an enqueue_policy is defined, which defines whether the nested command groups will not start until: the parent command group kernel has finished execution; the work-group that enqueued the device-kernel has finished execution, or there will be no wait at all. Along with this information, `device_event` is a class that provides event sychronisation on the device between the parent and the children command group functor submissions. Even in the case of no wait being requested, there is never a guarantee that child kernels will run in parallel with parent kernels, i.e. serialization of kernel execution is always valid for an implementation.

7.3.1 device_queue interface

The constructors and methods of the `device_queue` class are listed in Tables 7.10 and 7.11.

```
namespace cl {
namespace sycl {

enum enqueue_status {
  success,
  failure,
  invalid_queue,
  invalid_ndrange,
  invalid_event_wait_list,
  queue_full,
  invalid_arg_size,
  event_allocation_failure,
  out_of_resources
};

enum class enqueue_policy { no_wait, wait_kernel, wait_work_group };

struct event {
  event();
  event(const event&) = default;
  event(event&) = default;
  event& operator=(const event&) = default;
  event& operator=(event&&) = default;
  event* operator&() = delete;
  bool is_valid() const noexcept;
  void retain() noexcept;
  void release() noexcept;
  void set_status(event_status status) noexcept;
  template <class T>
  void profiling_info(event_profiling_info name, global_ptr<T> value) noexcept;
};

event make_user_event();

class device_queue {
 public:
  device_queue();

  device_queue(const device_queue&) = default;
  device_queue(device_queue&) = default;

  device_queue& operator=(const device_queue&) = default;
  device_queue& operator=(device_queue&&) = default;
  device_queue* operator&() = delete;

  ~device_queue();

  bool is_host() const;

  bool is_valid() const noexcept;

  template <typename T>
  event submit(T cgf);
};

}  // namespace sycl
}  // namespace cl
```

Constructors	Description
`device_queue ()`	Creates a device_queue using the default device_queue provided by the OpenCL runtime.
`device_queue (device_queue &)`	Copy constructor
	End of table

Table 7.10: Constructors of the `device_queue` class.

Methods	Description
`bool is_host ()const`	Returns whether the queue is executing on a SYCL host device.

`template <typename T>`
`    event submit(enqueue_policy flag, T cgf)`
	Submit a command group functor to the queue, in order to be scheduled for execution on the device according to the `enqueue_policy` given.

	End of table

Table 7.11: Methods for class device_queue

7.3.2 device_handle interface

```
namespace cl {
namespace sycl {
```

```
class device_handle {
 private:
  // implementation defined constructor
  device_handle(___unespecified___);

 public:
  device_handle(const device_handler& rhs);

  // In the case of a functor with a globally visible name
  // the template parameter:"typename kernelName" can be ommitted
  // and the kernelType can be used instead.
  template <typename KernelName, class KernelType, class... Args>
  int single_task(KernelType, Args... args);

  // In the case of a functor with a globally visible name
  // the template parameter:"typename kernelName" can be ommitted
  // and the kernelType can be used instead.
  template <typename KernelName, class KernelType, class... Args>
  int parallel_for(range<dimensions> numWorkItems, KernelType, Args... args);

  // In the case of a functor with a globally visible name
  // the template parameter:"typename kernelName" can be ommitted
  // and the kernelType can be used instead.
  template <typename KernelName, class KernelType, class... Args>
  int parallel_for(range<dimensions> numWorkItems,
                   id<dimensions> workItemOffset, KernelType, Args... args);

  // In the case of a functor with a globally visible name
  // the template parameter:"typename kernelName" can be ommitted
  // and the kernelType can be used instead.
  template <typename KernelName, class KernelType, class... Args>
  int parallel_for(nd_range<dimensions> executionRange, KernelType, Args... args);

  // In the case of a functor with a globally visible name
  // the template parameter:"typename kernelName" can be ommitted
  // and the kernelType can be used instead.
  template <typename KernelName, class KernelType, work_group_exec Workgroup, Args... args>
  int parallel_for(nd_range<dimensions> numWorkItems,
                   id<dimensions> workItemOffset, KernelType, Args... args);

  int single_task(kernel syclKernel);

  int parallel_for(range<dimensions> numWorkItems, kernel syclKernel);

  int parallel_for(nd_range<dimensions> ndRange, kernel syclKernel);

  // In the case of a functor with a globally visible name
  // the template parameter:"typename kernelName" can be ommitted
  // and the kernelType can be used instead.
  template <class KernelName, class WorkgroupFunctionType>
  int parallel_for_work_group(range<dimensions> numWorkGroups,
                              WorkgroupFunctionType);

  // In the case of a functor with a globally visible name
  // the template parameter:"typename kernelName" can be ommitted
```

```
    // and the kernelType can be used instead.
    template <class KernelName, class WorkgroupFunctionType>
    int parallel_for_work_group(range<dimensions> numWorkGroups,
                                 range<dimensions> workGroupSize,
                                 WorkgroupFunctionType);

};

}  // namespace sycl
}  // namespace cl
```

7.3.3 device_event interface

The device_event class is only available on an device and does not support any interoperability with host, only using the events which are bound to the parent kernel, which was used to enqueue kernels on device.

```
namespace cl {
namespace sycl {

class device_event {
 public:
  event() = default;

  event(const event &rhs);

  ~event();

  cl::event get() noexcept;

  bool is_valid() const noexcept;

  void retain() noexcept;
  void release() noexcept;

  void set_status(event_status status) noexcept;

  template <info::event_profiling param>
  typename param_traits<info::event_profiling, param>::type get_profiling_info()
      const noexcept;
};
}  // namespace sycl
}  // namespace cl
```

Constructors	Description
`device_event ()= default`	Default construct a null event object.
`event (const device_event & rhs)`	Construct a copy sharing the same underlying event. The underlying event will be reference counted.
	End of table

Table 7.12: Constructors for the `device_event` class

Methods	Description
`cl::event get()`	Return the underlying OpenCL event reference. Retains a reference to the returned `cl_event` object. Caller should release it when finished.
`is_valid()`	Returns true if event is a valid event. Otherwise returns false. No exceptions are allowed.
`void retain()noexcept`	Increments the event reference count. No exceptions are allowed.
`void release()noexcept`	Decrements the event reference count. The event object is deleted once the event reference count is zero, the specific command identified by this event has completed (or terminated) and there are no commands in any device command queue that require a wait for this event to complete.
`template <info::event_profiling param> typename param_traits <info::event_profiling, param>::type get_profiling_info ()const`	Queries OpenCL profiling information for the underlying *cl_event*.
	End of table

Table 7.13: Methods for the `device_event` class

7.4 Stream class

SYCL stream class is available in SYCL instead of the `printf()` function in order to support C++ classes and structs and some commonly used stream output manipulators used in C++.

The `cl::sycl::stream` object can be used in order to output a sequence of characters on standard output. The stream object can be default constructed inside a command group scope and it can also have implementation defined constructors for output size and width.

Stream Operators	Description
`cl::sycl::stream=`	operator for assigning a stream to another.
`cl::sycl::operator<<`	operator for outputting a sequence of character or structs on standard output. All SYCL vector classes and id structs can be passed in this operator.
`cl::sycl::endl`	method that adds an end of line character to the output
`cl::sycl::hex`	Display the output in hexadecimal base
`cl::sycl::oct`	Display the output in octal base
`cl::sycl::setw`	Sets the field width of the output
`cl::sycl::precision`	Sets the decimal precision for the output
`cl::sycl::scientific`	Set the notation of the floating-point values as the C++ scientific notation

Stream Operators	**Description**
`cl::sycl::fixed`	Set the notation of the floating-point values as the C++ fixed notation
`cl::sycl::hexfloat`	Displays the floating point values in hexadecimal format
`cl::sycl::defaultfloat`	Displays the floating point values in the default notation.

The usage of the `cl::sycl::stream` object is not recommend for performance critical applications, as optimization levels for streaming operations are implementation defined, and the corresponding implementation documentation should be consulted.

8. SYCL Support of Non-Core OpenCL Features

OpenCL apart from *core* features that are supported in *every* platform, has *optional* features as well as *extensions* that are only supported in some platforms. The *optional* features, as described in the specification [1], and the OpenCL “khr” *extensions*, as described in the extension specification [2], are supported by the SYCL framework, but the ability to use them is completely dependent on the underlying OpenCL platforms. A SYCL implementation may support some vendor extensions in order to enable optimizations on certain platforms.

All OpenCL extensions are available through SYCL interoperability with OpenCL C, so all the extensions can be used through the OpenCL API as described in the extensions specification [2].

When running command groups on the host device, not all extensions are required to be available. The extensions available for the *host* are available to query in the same way as for SYCL devices, see Table 3.5.

8.1 Enable extensions in a SYCL kernel

In order to enable extensions in an OpenCL kernel the following compiler directive is used:

```
#pragma OPENCL EXTENSION <extension_name> : <behaviour>
```

The keyword *extension_name* can be:

- **all**, which refers to all the extensions available on a platform
- an **extension name** from the available extensions on a platform.

They keyword *behaviour* can be:

- **enable**: it will enable the extension specified by *extension_name* if it is available on the platform or otherwise it triggers a compiler warning. If *all* is specified in the *extension_name* then it will enable all extensions available.
- **disable**: it will disable all or any extension provided in the *extension_name*.

The following table 8.1 summarizes the levels of SYCL support to the API extensions for OpenCL 1.2 [2]. These extensions can be supported using OpenCL/SYCL interoperability API or by the extended SYCL API calls. This only applies for using them in the framework and only for devices that are supporting these extensions.

Continued on next page

Table 8.1: SYCL support for OpenCL 1.2 API extensions.

Extension	Support using SYCL/OpenCL API	Support using SYCL API
cl_khr_int64_base_atomics	Yes	Yes
cl_khr_int64_extended_atomics	Yes	Yes
cl_khr_fp16	Yes	Yes
cl_khr_3d_image_writes	Yes	Yes
cl_khr_gl_sharing	Yes	Yes
cl_apple_gl_sharing	Yes	Yes
cl_khr_d3d10_sharing	Yes	No
cl_khr_d3d11_sharing	Yes	No
cl_khr_dx9_media_sharing	Yes	No
		End of table

Table 8.1: SYCL support for OpenCL 1.2 API extensions.

8.2 Half Precision Floating-Point

The half precision floating-point data scalar and vector types are supported in the SYCL system. The SYCL host device supports those types, however they are optional on an OpenCL device and the developer always needs to check whether the device the kernel is going to run on has the corresponding extension.

The extension name is **`cl_khr_fp16`** and it needs to be used in order to enable the usage of the half data type on an SYCL OpenCL device.

The half type class, along with any OpenCL macros and definitions, is defined in the namespace `cl::sycl` as `half`. The vector type of half is supported sizes 2, 3, 4, 8 and 16 using the SYCL vectors (§ 4.10.2) along with all the methods supported for vectors.

The conversion rules follows the same rules as in the OpenCL 1.2 extensions specification [2, par. 9.5.1].

The math, common, geometric and relational functions can take `cl::SYCL ::opencl::half` as a type as they are defined in [2, par. 9.5.2, 9.5.3, 9.5.4, 9.5.5]. The valid type for the functions defined for half is described by the generic type name *genhalf* is described in table 8.2.

Generic type name	Description
`genhalf`	`cl::sycl::half, cl::sycl::half2, cl ::sycl::half3, cl::sycl::half4, cl:: sycl::half8, cl::sycl::half16`
	End of table

Table 8.2: Generic type name description for all valid types of kernel function parameters. [1]

The elementary floating-point functions available for SYCL host and device is extended to allow half as input. If the half type is given as a parameter then the allowed error in ULP(Unit in the Last Place) is less than 8192. They correspond to Table 6.9 of the OpenCL 1.2 specification [1]

Math function	Description
`genhalf cos (genhalf x)`	Compute cosine. x must be in the range -216 to +216.
`genhalf divide (genhalf x, genhalf y)`	Compute x / y.
`genhalf exp (genhalf x)`	Compute the base- e exponential of x.
`genhalf exp2 (genhalf x)`	Compute the base- 2 exponential of x.
`genhalf exp10 (genhalf x)`	Compute the base- 10 exponential of x.
`genhalf log (genhalf x)`	Compute natural logarithm.
`genhalf log2 (genhalf x)`	Compute a base 2 logarithm.
`genhalf log10 (genhalf x)`	Compute a base 10 logarithm.
`genhalf powr (genhalf x, genhalf y)`	Compute x to the power y, where *x* >= 0.
`genhalf recip (genhalf x)`	Compute reciprocal.
`genhalf rsqrt (genhalf x)`	Compute inverse square root.
`genhalf sin (genhalf x)`	Compute sine. x must be in the range -216 to +216.
`genhalf sqrt (genhalf x)`	Compute square root.
`genhalf tan (genhalf x)`	Compute tangent. x must be in the range -216 to +216.
	End of table

Table 8.3: Extended elementary functions which work on SYCL host and device.

8.3 Writing to 3D image memory objects

The `accessor` class for target `access::target::image` in SYCL support methods for writing 3D image memory objects, but this functionality is *only allowed* on a device if the extension `cl_khr_3d_image_writes` is supported on that *device*.

8.4 Interoperability with OpenGL

OpenCL has a standard extension that allows interoperability with OpenGL objects. The features described in this section are only defined within SYCL if the underlying OpenCL implementation supports the OpenCL/OpenGL interoperability extension (`cl_khr_gl_sharing`).

8.4.1 OpenCL/OpenGL extensions to the context class

If the `cl_khr_gl_sharing` extension is present then the developer can create an OpenCL context from an OpenGL context by providing the corresponding attribute names and values to *properties* for the devices chosen by device selector. Table 3.8 has the additions shown on Table 8.5.

cl_context_properties flag	Description
`CL_GL_CONTEXT_KHR`	OpenGL context handle (default: 0)
`CL_EGL_DISPLAY_KHR`	CGL share group handle (default: 0)
`CL_GLX_DISPLAY_KHR`	EGLDisplay handle (default: `EGL_NO_DISPLAY`)
`CL_WGL_HDC_KHR`	X handle (default: None)
`CL_CGL_SHAREGROUP_KHR`	HDC handle (default: 0)
	End of table

Table 8.4: Additional optional properties for creating context for SYCL/OpenGL sharing.

The following table 8.5 describes the additional methods of the context class defined for the OpenCL/OpenGL interop that are also available for SYCL/OpenGL interop. If the OpenGL extensions are not available then their behavior is implementation defined.

Methods	Description
`device get_gl_current_device ()`	Returns the OpenGL enabled device in the current context.
`vector_class<device> get_gl_context_devices ()`	Returns the OpenGL supported devices in this context.
	End of table

Table 8.5: Extended `context` class interface.

The SYCL extension for creating OpenCL context from an OpenGL context is based on the OpenCL extension specification and all the capabilities and restrictions are based on it and developers and implementers are advised to refer to [2, sec. 9.6].

8.4.2 Sharing SYCL/OpenGL memory objects

It is possible to share objects between SYCL and OpenGL, if the corresponding OpenCL platform extensions for these are available on available platforms. OpenCL memory objects based on OpenGL objects can only be created only if the OpenCL context is created from an OpenGL share group object or context. As the latter are OS specific, the OpenCL extensions are platform specific as well. In MacOS X the extension `cl_apple_gl_sharing` needs to be available for this functionality. If it is Windows/Linux/Unix, then the extension `cl_khr_gl_sharing` needs to be available. All the OpenGL objects within the shared group used for the creation of the context can be used apart from the default OpenGL objects.

Any of the buffers or images created through SYCL using the shared group objects for OpenGL are invalid if the corresponding OpenGL context is destroyed through usage of the OpenGL API. If buffers or images are used after the destruction of the corresponding OpenGL context then the behaviour of the system is undefined.

8.4.2.1 SYCL/OpenGL extensions to SYCL buffer

A SYCL *buffer* can be created from an OpenGL buffer object but the lifetime of the SYCL buffer is bound to the lifetime of the OpenCL context given in order to create the buffer. The GL buffer has to be created a priori using the OpenGL API, although it doesn't need to be initialized. If the OpenGL buffer object is destroyed or otherwise manipulated through the OpenGL API, before its usage through SYCL is completed, then the behaviour is undefined.

The functionality of the buffer and the accessor class is retained as for any other SYCL buffer defined in this system.

Constructor	Description

`template <typename T, int dimensions = 1>`
`buffer(context &clGlContext, GLuint glBufferObj)`
	Constructs a buffer from a OpenCL/OpenGL interop context and a `gl_buffer` object.

	End of table

Table 8.6: Extended constructors for the `buffer` class.

The extended methods of the buffer class, which have defined only when behavior when the OpenGL extensions are available on device, otherwise its undefined.

Method	Description
`cl_gl_object_type get_gl_info ( GLuint glBufferObj )`	Returns the cl_gl_object_type of the underlying OpenGL buffer.
	End of table

Table 8.7: Extended `buffer` class interface

8.4.2.2 SYCL/OpenGL extensions to SYCL image

A SYCL `image` can be created from an OpenGL *buffer*, from an OpenGL *texture* or from an OpenGL *renderbuffer*. However, the lifetime of the SYCL image is bound to the lifetime of the OpenCL/OpenGL context given in order to create the image and the OpenGL object's lifetime. The GL buffer, texture or renderbuffer has to be created a priori via the OpenGL API, although it doesn't need to be initialized. If the OpenGL object is destroyed or otherwise manipulated through the OpenGL API before its usage through SYCL is completed, then the behaviour is undefined.

Constructor	Description

`template<int dimensions = 1>`
`    image(context &clGlContext, GLuint glBufferObj)`
	Creates an 1-D Image from an OpenGL buffer object.

`template<int dimensions = 2>`
`    image( context &clGlContext, GLuint`
`glRenderbufferObj)`
	Create a 2-D image from an OpenGL renderbuffer object.

	Continued on next page

Table 8.8: Additional optional *image* class constructors.

Constructor	Description

`template<int dimensions = 1>`
`    image(context &clGlContext, GLenum textureTarget`
`, GLuint glTexture, GLint glMiplevel)`
	Creates a 1-D image from an OpenGL texture object with given textureTarget and mipmap level. The textureTarget can be one of the following:
	• `GL_TEXTURE_1D`
	• `GL_TEXTURE_1D_ARRAY`
	• `GL_TEXTURE_BUFFER`

`template<int dimensions = 2>`
`    image(context &clGlContext, GLenum textureTarget`
`, GLuint glTexture, GLint glMiplevel)`
	Creates a 2-D image from an OpenGL texture object with given textureTarget and mipmap level. The textureTarget can be one of the following:
	• `GL_TEXTURE_2D`
	• `GL_TEXTURE_2D_ARRAY`
	• `GL_TEXTURE_CUBE_MAP_-POSITIVE_X`
	• `GL_TEXTURE_CUBE_MAP_-POSITIVE_Y`
	• `GL_TEXTURE_CUBE_MAP_-POSITIVE_Z`
	• `GL_TEXTURE_CUBE_MAP_-NEGATIVE_X`
	• `GL_TEXTURE_CUBE_MAP_-NEGATIVE_Y`
	• `GL_TEXTURE_CUBE_MAP_-NEGATIVE_Z`
	• `GL_TEXTURE_RECTANGLE`

`template<int dimensions = 3>`
`    image(context &clGlContext, GLenum textureTarget`
`, GLuint glTexture, GLint glMiplevel)`
	Creates a 3-D image from an OpenGL texture object with given textureTarget and mipmap level. The textureTarget can be one of the following:
	• `GL_TEXTURE_3D`

	End of table

Table 8.8: Additional optional *image* class constructors.

Method	Description
`GLenum get_gl_texture_target ()`	Returns the OpenGL texture_target corresponding to the underlying texture which the context was created with.
`GLint get_gl_mipmap_level ()`	Returns the mipmap level of the underlying texture.
	End of table

Table 8.9: Additional optional *image* class method.

The *texture* provided has to be an OpenGL texture created through the OpenGL API and has to be a valid 1D, 2D, 3D texture or 1D array, 2D array texture or a cubemap, rectangle or buffer texture object. The format and the dimensions provided for the miplevel of the texture are used to create the OpenCL image object. The format of the OpenGL texture or renderbuffer object needs to match the format of the OpenCL image format. The compatible formats are specified in Table 9.4 of the OpenCL 1.2 extensions document [2, par. 9.7.3.1] and are also included in Table 8.10. If the texture or renderbuffer has a different format than the ones specified in 8.10, it is not guaranteed that the image created will be mapped to the the original texture.

OpenGL internal format	**Corresponding OpenCL image format (channel order, channel data type)**
`GL_RGBA8`	`CL_RGBA`, `CL_UNORM_INT8`, `CL_BGRA`, `CL_UNORM_INT8`
`GL_RGBA`, `GL_UNSIGNED_INT_8_8_8_8_REV`	`CL_RGBA`, `CL_UNORM_INT8`
`GL_BGRA`, `GL_UNSIGNED_INT_8_8_8_8_REV`	`CL_BGRA`, `CL_UNORM_INT8`
`GL_RGBA16`	`CL_RGBA`, `CL_UNORM_INT16`
`GL_RGBA8I`, `GL_RGBA8I_EXT`	`CL_RGBA`, `CL_SIGNED_INT8`
`GL_RGBA16I`, `GL_RGBA16I_EXT`	`CL_RGBA`, `CL_SIGNED_INT16`
`GL_RGBA32I`, `GL_RGBA32I_EXT`	`CL_RGBA`, `CL_SIGNED_INT32`
`GL_RGBA8UI`, `GL_RGBA8UI_EXT`	`CL_RGBA`, `CL_UNSIGNED_INT8`
`GL_RGBA16UI`, `GL_RGBA16UI_EXT`	`CL_RGBA`, `CL_UNSIGNED_INT16`
`GL_RGBA32UI`, `GL_RGBA32UI_EXT`	`CL_RGBA`, `CL_UNSIGNED_INT32`
`GL_RGBA16F`, `GL_RGBA16F_ARB`	`CL_RGBA`, `CL_HALF_FLOAT`
`GL_RGBA32F`, `GL_RGBA32F_ARB`	`CL_RGBA`, `CL_FLOAT`
	End of table

Table 8.10: Mapping of GL internal format to CL image format (reference: [2, table 9.4])

8.4.2.3 SYCL/OpenGL extensions to SYCL accessors

In order for SYCL to support the OpenCL/OpenGL interoperability, the classes for buffers and images have to be extended so that the OpenCL memory objects to be created from OpenGL objects. This extension, apart from restrictions on the creation and the life-time of the OpenCL objects, also requires that before the usage of any of these objects in an OpenCL *command_queue* an `acquire` command has to be enqueued first. In SYCL, the `accessor` classes make sure that the data are made available on a device. For this extension the SYCL kernel has to capture and through the accessor classes acquire any targets that are declared as interoperability targets.

The required extension for the accessor class are shown on Table 8.11.

Enumerator `access::target` values	**Description**
`acces::target::cl_gl_buffer`	access `buffer` which is created from an OpenGL buffer
`access::target::cl_gl_image`	access an `image` or that is created from an OpenGL shared object
	End of table

Table 8.11: Enumerator description for `access::target`

The SYCL system is responsible for synchronizing the OpenCL and OpenGL objects in use inside a `command_group` when the SYCL API is used and given that all the accessors for the buffers and images are marked as the interoperability targets.

8.4.2.4 SYCL/OpenGL extensions to SYCL events

In the case where the extension `cl_khr_gl_event` is available on a platform, the functionality for creating synchronizing OpenCL events with OpenGL events is available and the `event` class is extended to include them.

A SYCL event can be constructed from an OpenGL sync object with the extensions to the `event` class shown on Table 8.12.

Constructor	Description
`event(context &clGlContext, GL_sync syncObj)`	Creates an event which enables waiting on events to also include synchronization with OpenGL objects that are shared using the OpenCL/OpenGL context
	End of table

Table 8.12: Additional optional class constructors for `event` class.

Method	Description
`GL_sync get_gl_info ()`	Returns GL_sync object.
	End of table

Table 8.13: Additional optional class method for `event` class.

The specification of the underlying OpenCL/OpenGL interoperability system for synchronizing OpenCL event with OpenGL sync objects is in the OpenCL extensions specification [2, sec. 9.8].

8.4.2.5 Extension for depth and depth-stencil images

The extension `cl_khr_depth_images` adds support for depth images and the extension `cl_khr_gl_depth_images` allows sharing between OpenCL depth images and OpenGL depth or depth-stencil textures. The SYCL system doesn't add any additional functionality towards this extension and follows the OpenCL 1.2 Specification [2, sec. 9.12] for depth and depth-Stencil images extension. All the image class constructors and methods of the SYCL API as described in Table 4.4 and 4.5 on page 89 are extended to enable the use of the same API when this extension is present. The API is able to support the type `image2d_depth_t` and `image2d_array_depth_t`. The OpenCL C API defined in [2, sec. 9.12] can be used as well with all the rules that apply for SYCL/OpenCL C interoperability.

9. SYCL Device Compiler

This section specifies the requirements of the SYCL device compiler. Most features described in this section relate to underlying OpenCL capabilities of target devices and limiting the requirements of device code to ensure portability.

9.1 Offline compilation of SYCL source files

There are two alternatives for a SYCL device compiler: a *shared source device compiler* and a *single-source device compiler*.

A SYCL shared source device compiler takes in a C++ source file, extracts only the SYCL kernels and outputs the device code in a form that can be enqueued from host code by the associated SYCL runtime. How the SYCL runtime invokes the kernels is implementation defined, but a typical approach is for a device compiler to produce a header file with the compiled kernel contained within it. By providing a command-line option to the host compiler, it would cause the implementation's SYCL header files to `#include` the generated header file. The SYCL specification has been written to allow this as an implementation approach in order to allow shared-source compilation. However, any of the mechanisms needed from the SYCL compiler, the SYCL runtime and build system are implementation defined, as they can vary depending on the platform and approach.

A SYCL single-source device compiler takes in a C++ source file and compiles both host and device code at the same time. This specification specifies how a SYCL single-source device compiler parses and outputs device code for kernels, but does not specify the host compilation.

9.2 Naming of kernels

SYCL kernels are extracted from C++ source files and stored in an implementation- defined format. In the case of the shared-source compilation model, the kernels have to be uniquely identified by both host and device compiler. This is required in order for the host runtime to be able to load the kernel by using the OpenCL host runtime interface.

From this requirement the following rules apply for naming the kernels:

- The kernel name is a *C++ typename*.
- The kernel needs to have a *globally-visible* name. In the case of a functor, the name can be the typename of the functor, as long as it is globally-visible. In the case where it isn't, a globally-visible name has to be provided, as template parameter to the kernel invoking interface, as described in 5.4. In C++11, lambdas[1] do not have a globally-visible name, so a globally-visible typename has to be provided in the kernel invoking

[1]C++14 lambdas have the same naming rules as C++11 lambdas.

interface, as described in 5.4.

- The kernel name has to be a unique identifier in the program.

In both single-source and shared-source implementations, a device compiler should detect the kernel invocations (e.g. `parallel_for<kernelname>`) in the source code and compile the enclosed kernels, storing them with their associated type name.

The format of the kernel and the compilation techniques are implementation defined. The interface between the compiler and the runtime for extracting and executing SYCL kernels on the device is implementation defined.

9.3 Language restrictions for kernels

The extracted SYCL kernels need to be compiled by an OpenCL online or offline compiler and be executed by the OpenCL 1.2 runtime. The extracted kernels need to be OpenCL 1.2 compliant kernels and as such there are certain restrictions that apply to them.

The following restrictions are applied to device functions and kernels:

- Structures containing pointers may be shared but the value of any pointer passed between SYCL devices or between the host and a SYCL device is undefined.

- Memory storage allocation is not allowed in kernels, all memory allocation for the device is done on host, using accessor classes. Consequently, the default allocation `operator new` overloads that allocate storage are disallowed in a SYCL kernel. The placement `new` operator and any user-defined overloads that do not allocate storage are permitted.

- No virtual methods are allowed to be called in a SYCL kernel or any functions called by the kernel.

- No function pointers are allowed to be called in a SYCL kernel or any functions called by the kernel.

- No class with a *vtable* can be used in a SYCL kernel or any code included in the kernel.

- RTTI is disabled inside kernels.

- Exception-handling cannot be used inside a SYCL kernel or any code called from the kernel.

- Recursion is not allowed in a SYCL kernel or any code called from the kernel.

- Global variables are not allowed to be used in kernel code.

- Non-const static member variables are not allowed to be used in kernel code.

- The rules for kernels apply to both the kernel functors themselves and all functions, operators, methods, constructors and destructors called by the kernel. This means that kernels can only use library functions that have been adapted to work with SYCL. Implementations are not required to support any library routines in kernels beyond those explicitly mentioned as usable in kernels in this spec. Developers should refer to the SYCL built-in functions in 7.1 to find functions that are specified to be usable in kernels.

9.4 Compilation of functions

The SYCL device compiler parses an entire C++ source file supplied by the user. This also includes C++ header files, using `#include` directives. From this source file, the SYCL device compiler must compile kernels for the device, as well as any functions that the kernels call.

In SYCL, kernels are invoked using a kernel invoke function (e.g. `parallel_for`). The kernel invoke functions are templated by their kernel parameter, which is a function object (either a functor or a lambda). The code inside the function object that is invoked as a kernel is called the "kernel function". The "kernel function" must always return void. Any function called by the kernel function is compiled for device and called a "device function". Recursively, any function called by a device function is itself compiled as a device function.

For example, this source code shows three functions and a kernel invoke with comments explaining which functions need to be compiled for device.

```
void f ()
{
    // function "f" is not compiled for device

    single_task<class kernel_name>([=] ()
        {
            // This code compiled for device
            g (); // this line forces "g" to be compiled for device
        });
}

void g ()
{
    // called from kernel, so "g" is compiled for device
}

void h ()
{
    // not called from a device function, so not compiled for device
}
```

In order for the SYCL device compiler to correctly compile device functions, all functions in the source file, whether device functions or not, must be syntactically correct functions according to this specification. A syntactically correct function is a function that matches at least the C++11 specification, plus any extensions from the C++14 specification.

9.5 Built-in scalar data types

In a SYCL device compiler, the standard C++ fundamental types, including `int`, `short`, `long`, `long long int` need to be configured so that the device definitions of those types match the host definitions of those types. A device compiler may have this preconfigured so that it can match them based on the definitions of those types on the platform. Or there may be a necessity for a device compiler command-line option to ensure the types are the same.

The standard C++ fixed width types, e.g. `int8_t`, `int16_t`, `int32_t`,`int64_t`, should have the same size as defined by the C++ standard for host and device.

SYCL Scalar Datatypes	Description
`char`	a signed 8-bit integer.
`unsigned char`	an unsigned 8-bit integer.
`short int`	a signed integer of at least 16 bits and whose size must match the definition on the host.
`unsigned short int`	an unsigned integer of at least 16 bits and whose size must match the definition on the host.
`int`	a signed integer of at least 16 bits and whose size must match the definition on the host.
`unsigned int`	an unsigned integer of at least 16 bits and whose size must match the definition on the host.
`long int`	a signed integer of at least 32 bits and whose size must match the definition on the host.
`unsigned long int`	an unsigned integer of at least 32 bits and whose size must match the definition on the host.
`long long int`	an integer of at least 64 bits and whose size must match the definition on the host.
`unsigned long long int`	an unsigned integer of at least 64 bits and whose size must match the definition on the host.
`float`	a 32-bit IEEE 754 floating-point value.
`double`	a 64-bit IEEE 754 floating-point value.
`half`	a 16-bit IEEE 754-2008 half-precision floating-point value.
`size_t`	the unsigned integer type of the result of the `sizeof` operator on host.
	End of table

Table 9.1: SYCL compiler fundamental scalar datatypes

The SYCL device compiler also supports the OpenCL C scalar types which map to the OpenCL C language fundamental scalar datatypes 4.43.

9.6 Preprocessor directives and macros

The standard C++ preprocessing directives and macros are supported.

- `CL_SYCL_LANGUAGE_VERSION` substitutes an integer reflecting the version number of the SYCL language being supported by the device compiler. The version of SYCL defined in this document will have `CL_SYCL_LANGUAGE_VERSION` substitute the integer 120;

- `__FAST_RELAXED_MATH__` is used to determine if the `-cl-fast-relaxed-math` optimization option is specified in the build options given to the SYCL device compiler. This is an integer constant of 1 if the option is specified and undefined otherwise;
- `__SYCL_DEVICE_ONLY__` is defined to 1 if the source file is being compiled with a SYCL device compiler which does not produce host binary;
- `__SYCL_SINGLE_SOURCE__` is defined to 1 if the source file is being compiled with a SYCL single-source compiler which produces host as well as device binary;
- `__SYCL_TARGET_SPIR__` is defined to 1 if the source file is being compiled with a SYCL compiler which is producing OpenCL SPIR binary.
- `SYCL_EXTERNAL` is a macro which enables external linkage of SYCL functions and methods to be included in a SYCL kernel. For more details see 9.9.1

9.7 Attributes

The `attribute` syntax defined in the OpenCL C specification is supported in SYCL.

The `vec_type_hint`, `work_group_size_hint` and `reqd_work_group_size` kernel attributes in OpenCL C apply to kernel functions, but this is not syntactically possible in SYCL. In SYCL, these attributes are legal on device functions and their specification is propagated down to any caller of those device functions, such that the kernel attributes are the sum of all the kernel attributes of all device functions called. If there are any conflicts between different kernel attributes, then the behaviour is undefined.

9.8 Address-space deduction

In SYCL, there are several different types of pointer, or reference:

- Accessors give access to shared data. They can be bound to a memory object in a command group and passed into a kernel. Accessors are used in scheduling of kernels to define ordering. Accessors to buffers have a compile-time OpenCL address space based on their access mode.
- Explicit pointer classes (e.g. `global_ptr`) contain an OpenCL address space. This allows the compiler to determine whether the pointer references global, local, constant or private memory.
- C++ pointer and reference types (e.g. `int*`) are allowed within SYCL kernels. They can be constructed from the address of local variables, from explicit pointer classes, or from accessors. In all cases, a SYCL device compiler will need to auto-deduce the address space.

Inside kernels, conversions between accessors to buffers, explicit pointer classes and C++ pointers are allowed as long as they reference the same datatype and have compatible qualifiers and address spaces.

If a kernel function or device function contains a pointer or reference type, then address-space deduction must be attempted using the following rules:

- If a an explicit pointer class is converted into a C++ pointer value, then the C++ pointer value will have the address space of the explicit pointer class.

- If a variable is declared as a pointer type, but initialized in its declaration to a pointer value with an already-deduced address space, then that variable will have the same address space as its initializer.

- If a function parameter is declared as a pointer type, and the argument is a pointer value with a deduced address space, then the function will be compiled as if the parameter had the same address space as its argument. It is legal for a function to be called in different places with different address spaces for its arguments: in this case the function is said to be "duplicated" and compiled multiple times. Each duplicated instance of the function must compile legally in order to have defined behavior.

- The rules for pointer types also apply to reference types. i.e. a reference variable takes its address space from its initializer. A function with a reference parameter takes its address space from its argument.

- If no other rule above can be applied to a declaration of a pointer, then (for SYCL 1.2) it is assumed to be in the `private` address space. When compiling for SYCL 1.2, the default address space is the new `generic` address space for OpenCL v2.x devices.

It is illegal to assign a pointer value of one address space to a pointer variable of a different address space, except in the case of SYCL 2.2, where assigning pointers of the `private`, `global` or `local` address space can be assigned to pointers in the `generic` address space.

9.9 SYCL offline linking

9.9.1 SYCL functions and methods linkage

The default behavior in SYCL applications is that all the definitions and declarations of the functions and methods are available to the SYCL compiler, in the same translation unit. When this is not the case, all the symbols that need to be exported to a SYCL library or from a C++ library to a SYCL application need to be defined using the macro: `SYCL_EXTERNAL`.

The `SYCL_EXTERNAL` macro is implementation defined. It is the only requirement in the SYCL system for any function or method to be able to be linked against a SYCL application or library. The SYCL linkage mechanism is implementation defined, but the existence of the macro is required across all implementations.

9.9.2 Offline linking with OpenCL C libraries

The SYCL system supports external offline linking of OpenCL C libraries with a SYCL application. An OpenCL C function can be included and used in a SYCL program by defining it as an `extern "C"` function and adding the OpenCL library to the SYCL program. Any kernel which uses the external function needs to be included in a SYCL program which is linked against the OpenCL C library.

The data types for SYCL/OpenCL C interoperability are defined in 4.43. Only those data types can be used in the `extern "C"` declaration. These data types are invalid in an OpenCL C kernel, but these should be used in case of interop with a SYCL kernel and can be converted to and from the C++ fundamental types that are the default types in SYCL. The underlying OpenCL built-in types for pointers and vectors are defined as `typedef`s within the SYCL `vec` and explicit pointer types. The `vec` class contains a `vector_t` `typedef` for the underlying OpenCL

C data type, while the explicit pointer classes contain a `pointer_t` `typedef` for the underling OpenCL C pointer type.

A. Glossary

The purpose of this glossary is to define the key concepts involved in specifying OpenCL SYCL. This section includes definitions of terminology used throughout the specification document.

Accessor: An accessor is an interface which allows a kernel function to access data maintained by a buffer.

Application scope: The application scope is the normal C++ source code in the application, outside of command groups and kernels.

async_handler: An asynchronous error handler object which is defined as a function class instance encapsulating the code for receiving all the asynchronous exceptions triggered by the commands of the SYCL command queue or context.

Buffer: A buffer is an interface which maintains an area of memory which is to be accessed by a kernel function. It represents storage of data only, with access to that data achieved via accessors. The storage is managed by the SYCL runtime, but may involve OpenCL buffers.

Barrier: SYCL barriers are the same as OpenCL barriers. In SYCL, OpenCL's command-queue-barriers are created automatically on demand by the SYCL runtime to ensure kernels are executed in a semantically-correct order across multiple OpenCL contexts and queues. OpenCL's work-group barriers are available as an intrinsic function (same as in OpenCL) or generated automatically by SYCL's hierarchical parallel-for loops.

Command Group: All of the OpenCL commands, memory object creation, copying, mapping and synchronization operations to correctly execute a kernel on a device are defined in a functor and called a *command group*. Command groups, which are functor objects are executed in different threads are added to queues atomically, so it is safe to *submit* command group functors operating on shared queues, buffers and images.

Command Group Scope: The command group scope is the scope defined by the command group functor.

Command Queue: SYCL's command queues abstrac the OpenCL command queue functionality and add a SYCL-specific host command queue, which executes SYCL kernels on the host.

Constant Memory: " A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory." As defined in [1, p.15]

Device: SYCL's devices encapsulate OpenCL devices and add support for executing SYCL kernels on host.

Device Compiler: A SYCL device compiler is a compiler that is capable of taking in C++ source code containing SYCL kernels and outputting a binary object suitable for executing on an OpenCL device.

Functor: Functors are a concept from C++. An alternative name for functions in C++ is "function object". A functor is a C++ class with an **operator()** method that enables the object to be executed in a way that looks like a function call, but where the object itself is also passed in as a parameter.

Global ID: As in OpenCL, a global ID is used to uniquely identify a work-item and is derived from the number of global work-items specified when executing a kernel. A global ID is an N-dimensional value that starts at (0, 0, ...0).

Global Memory: As in OpenCL, global memory is a memory region accessible to all work-items executing in a context. Buffers are mapped or copied into global memory for individual contexts inside the SYCL runtime in order to enable accessors within command groups to give access to buffers from kernels.

Group ID: As in OpenCL, SYCL kernels execute in work groups. The group ID is the ID of the work group that a work item is executing within.

Group Range: A group range is the range specifying the size of the work group.

Host: As in OpenCL, the host is the system that executes the SYCL API and the rest of the application.

Host pointer: A pointer to memory that is in the virtual address space on the host.

ID: An id is a one, two or three dimensional vector of integers. There are several different types of ID in SYCL: global ID, local ID, group ID. These different IDs are used to define work items

Image: Images in SYCL, like buffers, are abstractions of the OpenCL equivalent. As in OpenCL, an image stores a two- or three-dimensional structured array. The SYCL runtime will map or copy images to OpenCL images in OpenCL contexts in order to execute semantically correct kernels in different OpenCL contexts. Images are also accessible on the host via the various SYCL accessors available.

Implementation defined: Behavior that is explicitly allowed to vary between conforming implementations of SYCL. A SYCL implementer is required to document the implementation defined behavior.

Item ID: An item id is an interface used to retrieve the global id, group id and local id of a work item.

Kernel: A SYCL kernel is a C++ functor or lambda function that is compiled to execute on a device. There are several ways to define SYCL kernels defined in the SYCL specification. It is also possible in SYCL to use OpenCL kernels as specified in the OpenCL specification. Kernels can execute on either an OpenCL device or on the host.

Kernel Name: A kernel name is a class type that is used to assign a name to the kernel function, used to link the host system with the kernel object output by the device compiler.

Kernel Scope: The scope inside the kernel functor or lambda is called kernel scope. Also, any function or method called from the kernel is also compiled in kernel scope. The kernel scope allows C++ language extensions as well as restrictions to reflect the capabilities of OpenCL devices. The extensions and restrictions are defined in the SYCL device compiler specification.

Local ID: A local id is an id which specifies a work items location within a group range.

Local Memory: As in OpenCL, local memory is a memory region associated with a work-group and accessible only by work-items in that work-group.

NDRange: An NDRange consists of two vectors of integers of one, two or three-dimensions that define the total number of work items to execute as well as the size of the work groups that the work items are to be executed within.

Platform: A platform in SYCL is encapsulates an OpenCL platform as defined in the OpenCL specification.

Private Memory: As in OpenCL, private memory is a region of memory private to a work-item. Variables defined in one work-items private memory are not visible to another work-item.

Program Object: A program object in SYCL is an OpenCL program object encapsulated in A SYCL class. It contains OpenCL kernels and functions compiled to execute on OpenCL devices. A program object can be generated from SYCL C++ kernels by the SYCL runtime, or obtained from an OpenCL implementation.

Shared Source Build System: A shared source build system means that a single source file passed through both a host compiler and one or more device compilers. This enables multiple devices, instruction sets and binary formats to be produced from the same source code and integrated into the same piece of software.

SYCL Runtime: A SYCL runtime is an implementation of the SYCL runtime specification. The SYCL runtime manages the different OpenCL platforms, devices, contexts as well as the mapping or copying of data between host and OpenCL contexts to enable semantically correct execution of SYCL kernels.

Work-Group: A work group is an OpenCL work group, defined in OpenCL as a collection of related work-items that execute on a single compute unit. The work-items in the group execute the same kernel and share local memory and work-group barriers.

Work-Item: A work item is an OpenCL work item, defined in OpenCL as one of a collection of parallel executions of a kernel invoked on a device by a command. A work-item is execute by one or more processing elements as part of a work-group executing on a compute unit. A work-item is distinguished from other executions within the collection by its global ID and local ID.

B. Interface of SYCL Classes in Full

B.1 Explicit pointer classes

The explicit pointer classes `global_ptr`, `local_ptr`, `private_ptr` and `constant_ptr` are defined in 4.8.1. The available functions for these classes in full are the following:

```
namespace cl {
namespace sycl {

template <typename ElementType>
class global_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr global_ptr();
  global_ptr(pointer);
  template <access::mode Mode>
  global_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  global_ptr(const global_ptr&);
  global_ptr &operator=(global_ptr &&r);
  constexpr global_ptr(nullptr);
  ~global_ptr();

  global_ptr &operator=(pointer r);
  global_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(global_ptr& r);
  global_ptr &operator++();
  global_ptr operator++(int);
  global_ptr &operator--();
  global_ptr operator--(int);
  global_ptr &operator+=(difference_type r);
  global_ptr &operator-=(difference_type r);
  global_ptr operator+(difference_type r);
  global_ptr operator-(difference_type r);

  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};

template <typename ElementType>
class constant_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr constant_ptr();
  constant_ptr(pointer);
  template <access::mode Mode>
  constant_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  constant_ptr(const constant_ptr&);
  constant_ptr &operator=(constant_ptr &&r);
  constexpr constant_ptr(nullptr);
  ~constant_ptr();

  constant_ptr &operator=(pointer r);
  constant_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(constant_ptr& r);
  constant_ptr &operator++();
  constant_ptr operator++(int);
  constant_ptr &operator--();
  constant_ptr operator--(int);
  constant_ptr &operator+=(difference_type r);
  constant_ptr &operator-=(difference_type r);
  constant_ptr operator+(difference_type r);
  constant_ptr operator-(difference_type r);

  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};

template <typename ElementType>
class local_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr local_ptr();
  local_ptr(pointer);
  template <access::mode Mode>
  local_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  local_ptr(const local_ptr&);
  local_ptr &operator=(local_ptr &&r);
  constexpr local_ptr(nullptr);
  ~local_ptr();

  local_ptr &operator=(pointer r);
  local_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(local_ptr& r);
  local_ptr &operator++();
  local_ptr operator++(int);
  local_ptr &operator--();
  local_ptr operator--(int);
  local_ptr &operator+=(difference_type r);
  local_ptr &operator-=(difference_type r);
  local_ptr operator+(difference_type r);
  local_ptr operator-(difference_type r);
  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};

template <typename ElementType>
class private_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr private_ptr();
  private_ptr(pointer);
  template <access::mode Mode>
  private_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  private_ptr(const private_ptr&);
  private_ptr &operator=(private_ptr &&r);
  constexpr private_ptr(nullptr);
  ~private_ptr();

  private_ptr &operator=(pointer r);
  private_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(private_ptr& r);
  private_ptr &operator++();
  private_ptr operator++(int);
  private_ptr &operator--();
  private_ptr operator--(int);
  private_ptr &operator+=(difference_type r);
  private_ptr &operator-=(difference_type r);
  private_ptr operator+(difference_type r);
  private_ptr operator-(difference_type r);
  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};

template <typename ElementType>
class generic_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // default generic space and forces disabling the address-space deduction
  typedef __undefined__ pointer;
  typedef ElementType element_type;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ ElementType& reference;
  typedef const __undefined__ ElementType& const_reference;
  typedef const __undefined__ const_pointer;

  constexpr generic_ptr();
  generic_ptr(pointer);
  template <access::mode Mode>
  generic_ptr(accessor<ElementType, 1, Mode, global_buffer>);
  generic_ptr(const generic_ptr&);
  generic_ptr &operator=(generic_ptr &&r);
  constexpr generic_ptr(nullptr);
  ~generic_ptr();

  generic_ptr &operator=(pointer r);
  generic_ptr &operator=(nullptr_t);
  reference operator*();
  reference operator[](size_t i);

  pointer release();
  void reset(pointer p = pointer());
  void swap(generic_ptr& r);
  generic_ptr &operator++();
  generic_ptr operator++(int);
  generic_ptr &operator--();
  generic_ptr operator--(int);
  generic_ptr &operator+=(difference_type r);
  generic_ptr &operator-=(difference_type r);
  generic_ptr operator+(difference_type r);
  generic_ptr operator-(difference_type r);
  // implementation defined implicit conversion
  // to OpenCL C pointer types.
  operator pointer();
};
}  // namespace sycl
}  // namespace cl

template <typename ElementType>
bool operator==(const global_ptr<ElementType>& lhs,
                const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator!=(const global_ptr<ElementType>& lhs,
                const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<(const global_ptr<ElementType>& lhs,
               const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>(const global_ptr<ElementType>& lhs,
               const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const global_ptr<ElementType>& lhs,
                const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const global_ptr<ElementType>& lhs,
                const global_ptr<ElementType>& rhs);

template <typename ElementType>
bool operator!=(const global_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator!=(nullptr_t lhs, const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator==(const global_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator==(nullptr_t lhs, const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>(const global_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>(nullptr_t lhs, const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<(const global_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator<(nullptr_t lhs, const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const global_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>=(nullptr_t lhs, const global_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const global_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
```

```
bool operator<=(nullptr_t lhs, const global_ptr<ElementType>& rhs);

template <typename ElementType>
bool operator==(const constant_ptr<ElementType>& lhs,
                const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator!=(const constant_ptr<ElementType>& lhs,
                const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<(const constant_ptr<ElementType>& lhs,
               const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>(const constant_ptr<ElementType>& lhs,
               const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const constant_ptr<ElementType>& lhs,
                const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const constant_ptr<ElementType>& lhs,
                const constant_ptr<ElementType>& rhs);

template <typename ElementType>
bool operator!=(const constant_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator!=(nullptr_t lhs, const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator==(const constant_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator==(nullptr_t lhs, const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>(const constant_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>(nullptr_t lhs, const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<(const constant_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator<(nullptr_t lhs, const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const constant_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>=(nullptr_t lhs, const constant_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const constant_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator<=(nullptr_t lhs, const constant_ptr<ElementType>& rhs);

template <typename ElementType>
bool operator==(const local_ptr<ElementType>& lhs,
                const local_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator!=(const local_ptr<ElementType>& lhs,
                const local_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<(const local_ptr<ElementType>& lhs,
               const local_ptr<ElementType>& rhs);
```

```
template <typename ElementType>
bool operator>(const local_ptr<ElementType>& lhs,
               const local_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const local_ptr<ElementType>& lhs,
                const local_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const local_ptr<ElementType>& lhs,
                const local_ptr<ElementType>& rhs);

template <typename ElementType>
bool operator!=(const local_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator!=(nullptr_t lhs, const local_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator==(const local_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator==(nullptr_t lhs, const local_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>(const local_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>(nullptr_t lhs, const local_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<(const local_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator<(nullptr_t lhs, const local_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const local_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>=(nullptr_t lhs, const local_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const local_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator<=(nullptr_t lhs, const local_ptr<ElementType>& rhs);

template <typename ElementType>
bool operator==(const private_ptr<ElementType>& lhs,
                const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator!=(const private_ptr<ElementType>& lhs,
                const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<(const private_ptr<ElementType>& lhs,
               const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>(const private_ptr<ElementType>& lhs,
               const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const private_ptr<ElementType>& lhs,
                const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const private_ptr<ElementType>& lhs,
                const private_ptr<ElementType>& rhs);

template <typename ElementType>
```

```
bool operator!=(const private_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator!=(nullptr_t lhs, const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator==(const private_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator==(nullptr_t lhs, const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>(const private_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>(nullptr_t lhs, const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<(const private_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator<(nullptr_t lhs, const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const private_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>=(nullptr_t lhs, const private_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const private_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator<=(nullptr_t lhs, const private_ptr<ElementType>& rhs);

template <typename ElementType>
bool operator==(const generic_ptr<ElementType>& lhs,
                const generic_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator!=(const generic_ptr<ElementType>& lhs,
                const generic_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<(const generic_ptr<ElementType>& lhs,
               const generic_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>(const generic_ptr<ElementType>& lhs,
               const generic_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const generic_ptr<ElementType>& lhs,
                const generic_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const generic_ptr<ElementType>& lhs,
                const generic_ptr<ElementType>& rhs);

template <typename ElementType>
bool operator!=(const generic_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator!=(nullptr_t lhs, const generic_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator==(const generic_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator==(nullptr_t lhs, const generic_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>(const generic_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>(nullptr_t lhs, const generic_ptr<ElementType>& rhs);
```

```
template <typename ElementType>
bool operator<(const generic_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator<(nullptr_t lhs, const generic_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator>=(const generic_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator>=(nullptr_t lhs, const generic_ptr<ElementType>& rhs);
template <typename ElementType>
bool operator<=(const generic_ptr<ElementType>& lhs, nullptr_t rhs);
template <typename ElementType>
bool operator<=(nullptr_t lhs, const generic_ptr<ElementType>& rhs);
```

B.2 Multi pointer pointer class

The multi_ptr class is defined in 4.8.1.1. The available functions for this class in full is the following:

```
namespace cl {
namespace sycl {
namespace access {
enum class address_space : int {
  global_space,
  local_space,
  constant_space,
  private_space,
  generic_space
};
}  // namespace access

template <typename ElementType, access::address_space Space>
class multi_ptr {
 public:
  // Implementation defined pointer type that corresponds to the SYCL/OpenCL
  // interoperability type for OpenCL C functions
  typedef __undefined__ pointer;
  typedef ptrdiff_t difference_type;
  typedef __undefined__ T& reference;
  typedef __undefined__ const T& const_reference;
  typedef __undefined__ T* pointer;
  typedef __undefined__ const T* const_pointer;

  const address_space space;
  constexpr multi_ptr();
  multi_ptr(pointer);
  multi_ptr(const multi_ptr&);
  multi_ptr(multi_ptr&& r);
  constexpr multi_ptr(nullptr_t);
  ~multi_ptr();

  reference operator*();
  // Only if Space == global_space
  operator global_ptr<ElementType>();
  global_ptr<ElementType> pointer();

  // Only if Space == local_space
  operator local_ptr<ElementType>();
  local_ptr<ElementType> pointer();

  // Only if Space == constant_space
  operator constant_ptr<ElementType>();
  constant_ptr<ElementType> pointer();

  // Only if Space == private_space
  operator private_ptr<ElementType>();
  private_ptr<ElementType> pointer();

  // Only if Space == generic
  operator generic_ptr<ElementType>();
  generic_ptr<ElementType> pointer();

  pointer release();
  void reset(pointer p = pointer());
  void swap(multi_ptr& r);

  multi_ptr& operator++();
  multi_ptr operator++(int);
  multi_ptr& operator--();
  multi_ptr operator--(int);
  multi_ptr& operator+=(difference_type r);
  multi_ptr& operator-=(difference_type r);
  multi_ptr operator+(difference_type r);
  multi_ptr operator-(difference_type r);
};

template <typename ElementType, access::address_space Space>
multi_ptr<ElementType, Space> make_ptr(pointer);
}  // namespace sycl
}  // namespace cl

template <typename ElementType, access::address_space Space>
bool operator==(const multi_ptr<ElementType, Space>& lhs,
                const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator!=(const multi_ptr<ElementType, Space>& lhs,
                const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator<(const multi_ptr<ElementType, Space>& lhs,
               const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator>(const multi_ptr<ElementType, Space>& lhs,
               const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator<=(const multi_ptr<ElementType, Space>& lhs,
                const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
```

```
bool operator>=(const multi_ptr<ElementType, Space>& lhs,
                const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator!=(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs);
template <typename ElementType, access::address_space Space>
bool operator!=(nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator==(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs);
template <typename ElementType, access::address_space Space>
bool operator==(nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator>(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs);
template <typename ElementType, access::address_space Space>
bool operator>(nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator<(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs);
template <typename ElementType, access::address_space Space>
bool operator<(nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator>=(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs);
template <typename ElementType, access::address_space Space>
bool operator>=(nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
template <typename ElementType, access::address_space Space>
bool operator<=(const multi_ptr<ElementType, Space>& lhs, nullptr_t rhs);
template <typename ElementType, access::address_space Space>
bool operator<=(nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
```

B.3 range class

The range class is defined in 5.2.1. The available functions for this class in full is the following:

```
namespace cl {
namespace sycl {
template <size_t dimensions>
struct range {
  range(const range<dimensions> &);

  range(size_t x);                   // When dimensions==1
  range(size_t x, size_t y);         // When dimensions==2
  range(size_t x, size_t y, size_t z); // When dimensions==3

  size_t get(int dimension) const;
  size_t &operator[](int dimension);

  range &operator=(const range &rhs);
  range &operator+=(const range &rhs);
  range &operator*=(const range &rhs);
  range &operator/=(const range &rhs);
  range &operator%=(const range &rhs);
  range &operator>>=(const range &rhs);
  range &operator<<=(const range &rhs);
  range &operator&=(const range &rhs);
  range &operator^=(const range &rhs);
  range &operator|=(const range &rhs);

  size_t size() const;
};

template <size_t dimensions>
bool operator==(const range<dimensions> &a, const range<dimensions> &b);
template <size_t dimensions>
bool operator!=(const range<dimensions> &a, const range<dimensions> &b);
template <size_t dimensions>
bool operator>(const range<dimensions> &a, const range<dimensions> &b);
template <size_t dimensions>
bool operator<(const range<dimensions> &a, const range<dimensions> &b);
template <size_t dimensions>
bool operator<=(const range<dimensions> &a, const range<dimensions> &b);
template <size_t dimensions>
bool operator>=(const range<dimensions> &a, const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator*(const range<dimensions> &a,
                            const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator/(const range<dimensions> &a,
                            const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator+(const range<dimensions> &a,
                            const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator-(const range<dimensions> &a,
                            const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator%(const range<dimensions> &a,
                            const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator<<(const range<dimensions> &a,
                             const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator>>(const range<dimensions> &a,
                             const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator&(const range<dimensions> &a,
                            const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator|(const range<dimensions> &a,
                            const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator||(const range<dimensions> &a,
                             const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator&&(const range<dimensions> &a,
                             const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator^(const range<dimensions> &a,
                            const range<dimensions> &b);
```

```
template <size_t dimensions>
range<dimensions> operator*(const size_t &a, const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator*(const range<dimensions> &a, const size_t &b);
template <size_t dimensions>
range<dimensions> operator/(const size_t &a, const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator/(const range<dimensions> &a, const size_t &b);
template <size_t dimensions>
range<dimensions> operator+(const size_t &a, const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator+(const range<dimensions> &a, const size_t &b);
template <size_t dimensions>
range<dimensions> operator-(const size_t &a, const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator-(const range<dimensions> &a, const size_t &b);
template <size_t dimensions>
range<dimensions> operator%(const size_t &a, const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator%(const range<dimensions> &a, const size_t &b);
template <size_t dimensions>
range<dimensions> operator<<(const size_t &a, const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator<<(const range<dimensions> &a, const size_t &b);
template <size_t dimensions>
range<dimensions> operator>>(const size_t &a, const range<dimensions> &b);
template <size_t dimensions>
range<dimensions> operator>>(const range<dimensions> &a, const size_t &b);
}  // sycl
}  // cl
```

B.4 id class

The id class is defined in 5.2.3. The available functions for this class in full is the following:

```
namespace cl {
namespace sycl {
template <size_t dimensions>
struct id {
  id(size_t x);                         // When dimensions==1
  id(size_t x, size_t y);               // When dimensions==2
  id(size_t x, size_t y , size_t z);    // When dimensions==3
  id(const id<dimensions> & rhs);
  id(const range<dimensions> & rangeSize);
  id(const item<dimensions> & rhs);

  size_t get(int dimension) const;
  size_t &operator[](int dimension);
  operator size_t();            // When dimensions==1

  id &operator=(const id & rhs);
  id &operator+=(const id & rhs);
  id &operator*=(const id & rhs);
  id &operator/=(const id & rhs);
  id &operator%=(const id & rhs);
  id &operator>>=(const id & rhs);
  id &operator<<=(const id & rhs);
  id &operator&=(const id & rhs);
  id &operatorˆ=(const id & rhs);
  id &operator|=(const id & rhs);
};

template <size_t dimensions>
bool operator==(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
bool operator!=(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
bool operator>(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
bool operator<(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
bool operator<=(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
bool operator>=(const id<dimensions> &a, const id<dimensions> &)b;
template <size_t dimensions>
id<dimensions> operator*(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator/(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator+(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator-(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator%(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator<<(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator>>(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator&(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator|(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operatorˆ(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator&&(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator||(const id<dimensions> &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator*(const size_t &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator*(const id<dimensions> &a, const size_t &b);
template <size_t dimensions>
id<dimensions> operator/(const size_t &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator/(const id<dimensions> &a, const size_t &b);
```

```
template <size_t dimensions>
id<dimensions> operator+(const size_t &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator+(const id<dimensions> &a, const size_t &b);
template <size_t dimensions>
id<dimensions> operator-(const size_t &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator-(const id<dimensions> &a, const size_t &b);
template <size_t dimensions>
id<dimensions> operator%(const size_t &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator%(const id<dimensions> &a, const size_t &b);
template <size_t dimensions>
id<dimensions> operator<<(const size_t &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator<<(const id<dimensions> &a, const size_t &b);
template <size_t dimensions>
id<dimensions> operator>>(const size_t &a, const id<dimensions> &b);
template <size_t dimensions>
id<dimensions> operator>>(const id<dimensions> &a, const size_t &b);
}  // namespace sycl
}  // namespace cl
```

B.5 vec class

The vec class is defined in 4.10.2. The available functions for this class in full is the following:

```
namespace cl {
namespace sycl {
template <typename dataT, int numElements>
class vec {
 public:
  typedef dataT element_type;
  //Underlying OpenCL type
  typedef __undefined__ vector_t;

  vec();

  explicit vec(const dataT &arg);

  vec(const T0 &arg0... args);

  vec(const vec<dataT, numElements> &rhs);

  size_t get_count();

  size_t get_size();

  template <typename asDataT, int width>
  vec<asDataT, width> as() const;
```

```
// genvector is a generic typename for describing
// all OpenCL/SYCL types.
operator __genvector__() const;

// arithmetic operators
vec operator+(const vec &rhs) const;
vec operator-(const vec &rhs) const;
vec operator*(const vec &rhs) const;
vec operator/(const vec &rhs) const;
vec operator%(const vec &rhs) const;
vec operator++(const vec &rhs) const;
vec operator++();
vec operator++(int);
vec operator--();
vec operator--(int);
vec operator+(const dataT &rhs) const;
vec operator-(const dataT &rhs) const;
vec operator*(const dataT &rhs) const;
vec operator/(const dataT &rhs) const;
vec operator%(const dataT &rhs) const;

// bitwise and logical operators
vec operator|(const vec &rhs) const;
vec operator|(const dataT &rhs) const;
vec operator^(const vec &rhs) const;
vec operator^(const dataT &rhs) const;
vec operator&&(const vec<dataT, numElements> &rhs) const;
vec operator&&(const dataT &rhs) const;
vec operator||(const vec<dataT, numElements> &rhs) const;
vec operator||(const dataT &rhs) const;
vec operator>>(const vec<dataT, numElements> &rhs) const;
vec operator>>(const dataT &rhs) const;
vec operator<<(const vec &rhs) const;
vec operator<<(const dataT &rhs) const;
vec operator~();
vec operator!();

// assignment operators
vec operator+=(const vec &rhs);
vec operator+=(const dataT &rhs);
vec operator-=(const vec &rhs);
vec operator-=(const dataT &rhs);
vec operator*=(const vec &rhs);
vec operator*=(const dataT &rhs);
vec operator/=(const vec &rhs);
vec operator/=(const dataT rhs);
vec operator|=(const vec &rhs);
vec operator|=(const dataT &rhs);
vec operator^=(const vec &rhs);
vec operator^=(const dataT &rhs);
vec operator<<=(const vec &rhs);
vec operator<<=(const dataT &rhs);
vec operator>>=(const vec &rhs);
vec operator>>=(const dataT &rhs);
vec operator&=(const vec &rhs);
```

```
  vec operator&=(const dataT &rhs);
  vec &operator=(const vec &rhs);
  vec &operator=(const dataT &rhs);
  vec &operator%=(const vec &rhs);
  vec &operator%=(const dataT &rhs);

  // relational operators
  vec<int, numElements> operator==(const vec &rhs) const;
  vec<int, numElements> operator!=(const vec &rhs) const;
  vec<int, numElements> operator<=(const vec &rhs) const;
  vec<int, numElements> operator>=(const vec &rhs) const;
  vec<int, numElements> operator>(const vec &rhs) const;
  vec<int, numElements> operator<(const vec &rhs) const;

  // Swizzle methods (see notes)
  swizzled_vec<T, out_dims> swizzle<elem s1, ...>();

#ifdef SYCL_SIMPLE_SWIZZLES
  swizzled_vec<T, 4> xyzw();
  ...
#endif  // #ifdef SYCL_SIMPLE_SWIZZLES
};
}  // namespace sycl
}  // namespace cl
```

C. Interface of Memory Object Information Descriptors

C.1 Platform Information Descriptors

The following interface includes all the information descriptors for the `platform` class as described in table 3.6.

```
namespace cl {
namespace sycl {
namespace info {
enum class platform : unsigned int {
  profile,
  version,
  name,
  vendor,
  extensions,
  host_timer_resolution
};
}
```

C.2 Context Information Descriptors

The following interface includes all the information descriptors for the `context` class as described in table 3.9.

```
namespace cl {
namespace sycl {
namespace info {
bool gl_context_interop;
enum class context : int {
  reference_count,
  num_devices,
  gl_interop
};
}  // info
}  // sycl
}  // cl
```

C.3 Device Information Descriptors

The following interface includes all the information descriptors for the `device` class as described in table 3.12.

```
namespace cl {
namespace sycl {
namespace info {
enum class device_type : unsigned int {
  cpu,
  gpu,
  accelerator,
  custom,
  defaults,
  host,
  all
};

enum class device : int {
  type,
  vendor_id,
  max_compute_units,
  max_work_item_dimensions,
  max_work_item_sizes,
  max_work_group_size,
  preferred_vector_width_char,
  preferred_vector_width_short,
  preferred_vector_width_int,
  preferred_vector_width_long_long,
  preferred_vector_width_float,
  preferred_vector_width_double,
  preferred_vector_width_half,
  native_vector_witdth_char,
  native_vector_witdth_short,
  native_vector_witdth_int,
  native_vector_witdth_long_long,
  native_vector_witdth_float,
  native_vector_witdth_double,
  native_vector_witdth_half,
  max_clock_frequency,
  address_bits,
  max_mem_alloc_size,
  IL_version
  image_support,
  max_read_image_args,
  max_write_image_args,
  max_read_write_image_args,
  image2d_max_height,
  image2d_max_width,
  image3d_max_height,
  image3d_max_width,
  image3d_mas_depth,
  image_max_buffer_size,
  image_max_array_size,
  max_samplers,
  image_pitch_alignment,
  max_pipe_args,
  pipe_max_active_reservations,
  max_packet_size,
  max_parameter_size,
  mem_base_addr_align,
  single_fp_config,
  double_fp_config,
  global_mem_cache_type,
  global_mem_cache_line_size,
  global_mem_cache_size,
  global_mem_size,
  max_constant_buffer_size,
  max_constant_args,
  max_global_variable_size,
  global_variable_preferred_total_size,
  local_mem_type,
  local_mem_size,
  error_correction_support,
  host_unified_memory,
  profiling_timer_resolution,
  is_endian_little,
  is_available,
  is_compiler_available,
  is_linker_available,
  execution_capabilities,
  opencl_queue_out_of_order_exec,
  queue_profiling_enabled,
  device_queue_out_of_order_exec,
  device_queue_profiling_enabled,
  device_queue_preferred_size,
  device_queue_max_size,
  max_device_queues,
  max_device_events,
  built_in_kernels,
  platform,
  name,
  vendor,
  driver_version,
  profile,
  version,
  opencl_version,
  extensions,
  printf_buffer_size,
  preferred_interop_user_sync,
  parent_device,
  partition_max_sub_devices,
  partition_properties,
  partition_affinity_domain,
  partition_type,
  reference_count,
  svm_capabilities,
  preferred_platform_atomic_alignment,
  preferred_global_atomic_alignment,
  preferred_local_atomic_alignment,
  max_num_sub_groups,
  sub_group_independent_forward_progress
};

enum class partition_property : int {
  unsupported,
  partition_equally,
  partition_by_counts,
  partition_by_affinity_domain,
  partition_affinity_domain_next_partitionable
};

enum class affinity_domain : int {
  unsupported,
  numa,
  L4_cache,
  L3_cache,
  L2_cache,
  next_partitionable
};

enum class partition_type : int {
  no_partition,
  numa,
  L4_cache,
  L3_cache,
  L2_cache,
  L1_cache
};

enum class local_mem_type : int { none, local, global };

enum class fp_config : int {
  denorm,
  inf_nan,
  round_to_nearest,
  round_to_zero,
  round_to_inf,
  fma,
  correctly_rounded_divide_sqrt,
  soft_float
};

enum class global_mem_cache_type : int { none, read_only, write_only };

enum class execution_capabilities : unsigned int {
  exec_kernel,
  exec_native_kernel
};

enum class queue_properties : int { profiling_enable };

}  // namespace info
}  // namespace sycl
}  // namespace cl
```

C.4 Queue Information Descriptors

The following interface includes all the information descriptors for the queue class as described in table 3.15.

```
namespace cl {
namespace sycl {
namespace info {
bool queue_profiling;
enum class queue : int {
  context,
  device,
  reference_count,
  properties
};
}  // namespace info
}  // namespace sycl
}  // namespace cl
```

C.5 Kernel Information Descriptors

The following interface includes all the information descriptors for the kernel class as described in table 5.18.

```
namespace cl {
namespace sycl {
namespace info {
enum class kernel: int {
  function_name,
  num_args,
  reference_count,
  attributes
};
}  // namespace info
}  // namespace sycl
}  // namespace cl
```

C.6 Program Information Descriptors

The following interface includes all the information descriptors for the program class as described in table 5.21.

```
namespace cl {
namespace sycl {
namespace info {
enum class program: int {
    reference_count,
    context,
    devices
};
}  // namespace info
}  // namespace sycl
}  // namespace cl
```

C.7 Event Information Descriptors

The following interface includes all the information descriptors for the `event` class as described in table 5.18.

```
namespace cl {
namespace sycl {
namespace info {
enum class event: int {
  command_type,
  command_execution_status,
  reference_count
};

enum class event_profiling : int {
  command_queued,
  command_submit,
  command_start,
  command_end
};
}  // namespace info
}  // namespace sycl
}  // namespace cl
```

References

[1] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.2.19*, 2012. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

[2] ——, *The OpenCL Extension Specification, version 1.2.19*, 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf

[3] ——, *The OpenCL Specification, version 2.2*, 2016. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-2.2-api.pdf