SYCL™ (pronounced “sickle”) uses generic programming to enable higher-level application software to be cleanly coded with optimized acceleration of kernel code across a range of devices.

Developers program at a higher level than the native acceleration API, but always have access to lower-level code through seamless integration with the native acceleration API.

All definitions in this reference guide are in the sycl namespace. [n.n] refers to sections in the SYCL 2020 (revision 2) specification at khronos.org/registry/sycl

**Common interfaces**

**Common reference semantics [4.5.2]**

T may be accelerator, buffer, context, device, device_image, event, host_accessor, host_[un]sampled_image_accessor, kernel, kernel_id, kernel_bundle, local_accessor, platform, queue, [un]sampled_image, [un]sampled_image_accessor: T(const T &rhs); T(T &); T &operator=(const T &rhs); T &operator=(T &&); friend bool operator==(const T &rhs, const T &); friend bool operator==(T &rhs, const T &);

**Common by-value semantics [4.5.3]**

T may be id, range, item, nd_item, h_item, group, sub_group, or nd_range.

friend bool operator==(const T &rhs, const T &); friend bool operator==(T &rhs, const T &);

**Properties [4.5.4]**

Each of the constructors in the following SYCL runtime classes has an optional parameter to provide a property list containing zero or more properties: accelerator, buffer, host_accessor, host_[un]sampled_image_accessor, context, local_accessor, queue, [un]sampled_image, [un]sampled_image_accessor, stream, and usm_allocator.

- template<typename propertyT>
  struct is_property;
  template<typename propertyT>
  struct is_propertyT;
- template<typename param>
  get_devices()
  template<typename param>
  get_devices(std::vector<device> & devices);
- template<typename param>
  get_platforms
  template<typename param>
  get_platforms(platform & platforms);
- template<typename param>
  get_backend
  template<typename param>
  get_backend(platform & backend);
- template<typename param>
  get_platform_info
  template<typename param>
  get_platform_info(platform & platform);
- template<typename param>
  get_backend_info
  template<typename param>
  get_backend_info(platform & backend);
- template<typename param>
  get_kernel_info
  template<typename param>
  get_kernel_info(platform & platform);
- template<typename param>
  get_kernel_image_info
  template<typename param>
  get_kernel_image_info(platform & platform);
- template<typename param>
  get_queue_info
  template<typename param>
  get_queue_info(platform & platform);
- template<typename param>
  get_event_info
  template<typename param>
  get_event_info(platform & platform);

**Device selection [4.6.1]**

Device selection is done either by already having a specific instance of a device or by providing a device selector. The actual interface for a device selector is a callable taking a const device reference and returning a value implicitly convertible to an int. The system calls the function for each device, and the device with the highest value is selected.

**Pre-defined SYCL device selectors**

- default_selector_v
  Device selected by system heuristics
- gpu_selector_v
  Select a device according to device type info::device_type::gpu
- cpu_selector_v
  Select a device according to device type info::device_type::cpu
- accelerator_selector_v
  Select an accelerator device.

**Anatomy of a SYCL application [3.2]**

Below is an example of a typical SYCL application which schedules a job to run in parallel on any OpenCL accelerator. USM versions of this example are shown on page page 15 of this reference guide.

```cpp
#include <iostream>
#include <sycl/sycl.hpp>

int data[1024];

using namespace sycl; // (optional) avoids need for "sycl:" before SYCL names

int main() {
  int data[1024]; // Allocates data to be worked on
  queue myQueue; // Create default queue to enqueue work

  // By wrapping all the SYCL work in a { } block, we ensure all
  // SYCL tasks must complete before exiting the block,
  // because the destructor of resultBuf will wait.
  {
    // Wrap our data variable in a buffer:
    buffer<int, 1> resultBuf(data, data + 1024);

    // Create a command group to issue commands to the queue.
    myQueue.submit([&](handler &cgh) {
      // Request access to the buffer without initialization
      accessor readResult = resultBuf.get_access<accessor_read>(cgh);
      accessor writeResult = resultBuf.get_access<accessor_write>(cgh, write_only, no_init);

      // Enqueue a parallel task with 1024 work-items.
      cgh.parallel_for(1024, [=](auto idx) {
        // Initializer each buffer element with its own rank number starting at 0
        // and writeResult[idx] = idx;

        // The command kernel scope
        // specifies a single kernel function compiled by a device compiler and executed on a device.

        // The command group scope
        // specifies a unit of work which is comprised of a kernel function and its accessors.
        // The application scope
        // specifies all other code outside of a command group scope.

        // End of the kernel function
      });

      // End of the queue commands
    });

    // End of scope, so wait for the queued work to complete
    print_result();
  }

  return 0;
}
```

```
Also see an example of how to write a reduction kernel on page page 9 and examples of how to invoke kernels on page page 16.
```

**Platform class [4.6.2]**

The platform class encapsulates a single platform on which kernel functions may be executed. A platform is associated with a single backend.

```cpp
platform();
```

**Device class [4.6.4]**

The device class encapsulates a single device on which kernels can be executed. All member functions of the device class are synchronous.

```cpp
device();
```

Header file
SYCL programs must include the <sycl/sycl.hpp> header file to provide all of the SYCL features used in this example.

Namespace
SYCL names are defined in the sycl namespace.

Queue
This line implicitly selects the best underlying device to execute on. See queue class functions [4.6.5] on page 2 of this reference guide.

Buffer
All required data in a kernel must be inside a buffer or image or else USM is used. See buffer class functions [4.7.2] on page 3 of this reference guide.

Handler
See handler class functions [4.9.4] on page 9 of this reference guide.

Scopes
The kernel scope specifies a single kernel function compiled by a device compiler and executed on a device.

The command group scope specifies a unit of work which is comprised of a kernel function and its accessors.

The application scope specifies all other code outside of a command group scope.
Device class (cont.)

Context class [4.6.3]
The context class represents a context. A context represents the runtime data structures and state required by a backend API to interact with a group of devices associated with a platform.

```cpp
template <typename DeviceSelector>
context() const noexcept;
```

Context queries using get_info():
The following descriptor names are in the info::context namespace.

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver_version</td>
<td>std::string</td>
</tr>
<tr>
<td>version</td>
<td>std::string</td>
</tr>
<tr>
<td>backend_version</td>
<td>std::string</td>
</tr>
<tr>
<td>aspects</td>
<td>std::vector&lt;aspect&gt;</td>
</tr>
<tr>
<td>print_buffer_size</td>
<td>size_t</td>
</tr>
<tr>
<td>parent_device</td>
<td>device</td>
</tr>
<tr>
<td>partition_max_sub_devices</td>
<td>uint32_t</td>
</tr>
<tr>
<td>partition_properties</td>
<td>std::vector<a href="">info::partition_property</a></td>
</tr>
<tr>
<td>partition_affinity_domain</td>
<td>std::vector<a href="">info::partition_affinity_domain</a></td>
</tr>
<tr>
<td>partition_type</td>
<td>info::partition_type</td>
</tr>
<tr>
<td>partition_affinity_domain</td>
<td>info::partition_affinity_domain</td>
</tr>
</tbody>
</table>

Platform details:

```cpp
template <typename KernelName, int Dims, int Rest>:
```

Queue class [4.6.5]
The queue class encapsulates a single queue which schedules kernels on a device. A queue can be used to submit command groups to be executed by the runtime using the submit member function. Note that the destructor does not block.

```cpp
template <typename QueueSelector >:
```

Explicit queue:

```cpp
template <typename QueueSelector >
explicit queue() const;
```

Queue queries using get_info():

```cpp
template <typename param
```

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>info::queue:context</td>
<td>context</td>
</tr>
<tr>
<td>info::queue:device</td>
<td>device</td>
</tr>
</tbody>
</table>

Convenience shortcuts

```cpp
template <typename KernelName, typename KernelType>
event single_task(event depEvent, const KernelType &KernelName);
```

Platform details:

```cpp
template <typename param
```
USM Functions

Event class [4.6.6]

An event is an object that represents the status of an operation that is being executed by the runtime.

event
backend get_backend() const noexcept;
void wait();
void wait_and_throw();

Event queries using get_info()

Buffers class [4.7.2]

The buffer class defines a shared array of one, two, or three dimensions that can be used by the kernel and has to be accessed using accessor classes. Note that the destructor does block.

Class declaration

Event queries using get_profiling_info()

Buffer property class constructors:

Queue property class constructors:

Host allocation [4.7.1]

The default allocator for memory objects is implementation defined, but users can supply their own allocator class, e.g.: buffer<int, 1, UserDefinedAllocator<int>> b; The default allocators are buffer::allocator for buffers and image::allocator for images.
Images, unsampled and sampled [4.7.3]

Buffers and images define storage and ownership. Images are of type unsampled image or sampled image. Their constructors take an image_format parameter from enum class image_format.

e num class image_format values:

r8g8b8a8_unorm
r16g16b16a16_unorm
r8g8b8_sint
r16g16b16a16_sint
s32b32g32a32_sint
r16b16g16a16_sfloat
r32b32g32a32_sfloat
r16b16g16a16_sfloat
b8g8r8a8_unorm

Unsampled images [4.7.3.1]

Class declaration

template<typename dimensions = 1, typename AllocatorT = std::nullptr_t>
class unsampled_image_accessor;

Constructors and members

- unsampled_image_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, const property_list &propList = {});
- unsampled_image_accessor(const void *hostPointer, image_format format, const range<dimensions> &pitch, AllocatorT allocator, const property_list &propList = {});
- unsampled_image_accessor(void *hostPointer, image_format format, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {});

Available when dimensions > 1

- unsampled_image_accessor(const void *hostPointer, image_format format, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {});
- unsampled_image_accessor(void *hostPointer, image_format format, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {});

Available when dimensions == 0

- unsampled_image_accessor(const void *hostPointer, image_format format, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {});

Available when dimensions < 1

- unsampled_image_accessor(void *hostPointer, image_format format, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {});

Sampled images [4.7.3.2]

Class declaration

template<typename dimensions = 1, typename AllocatorT = std::nullptr_t>
class sampled_image_accessor;

Constructors and members

- sampled_image_accessor(commandGroupHandlerRef &commandGroupHandlerRef, buffer<dataT, dimensions, AllocatorT> &bufferRef, const property_list &propList = {});
- sampled_image_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, const property_list &propList = {});
- sampled_image_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, const property_list &propList = {});

Available when dimensions == 1

- sampled_image_accessor(const void *hostPointer, image_format format, image_sampler sampler, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {});
- sampled_image_accessor(const void *hostPointer, image_format format, image_sampler sampler, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {});

Available when dimensions == 0

- sampled_image_accessor(const void *hostPointer, image_format format, image_sampler sampler, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {});

Available when dimensions < 0

- sampled_image_accessor(const void *hostPointer, image_format format, image_sampler sampler, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {});

Data access and storage [4.7]

Buffers and images define storage and ownership. Accessors provide access to the data.

Accessors [4.7.6]

Accessor classes and the objects they access:
- Buffer accessor for commands (4.7.6.9, class accessor) with two uses:
  - access a buffer from a kernel function via device global memory
  - access a buffer from a host task
- Buffer accessor for host code outside of a command (4.7.6.10, class host_accessor)
- Local accessor from within kernel functions (4.7.6.11, class local_accessor)

Buffer accessor for commands (class accessor) [4.7.6.9]

This one class provides two kinds of accessors depending on accessTarget:
- target: device to access a buffer from a kernel function via device global memory
- target: host_task to access a buffer from a host task

Class declaration

template<typename dataT, int dimensions, access_mode accessMode = (std::is_const_v<dataT> ? access_mode::read : access_mode::read_write), target accessTarget = target::device, class accessor>
class accessor;

Constructors and members

- accessor(buffer<dataT, 1, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, const property_list &propList = {});
- accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, const property_list &propList = {});

Available when dimensions == 0

- accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, const property_list &propList = {});
- accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, const property_list &propList = {});

Available when dimensions > 0

- accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, const property_list &propList = {});
- accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, range<dimensions> accessRange, id<dimensions> accessOffset, const property_list &propList = {});

(Continued on next page)
**Buffer accessor for commands (cont.)**

Available when dimensions > 0

```
template <typename AllocatorT, typename TagT>
  accessor<buffer<
    T, dimensions, AllocatorT>& bufferRef,
    range<
      accessRange,
      id<
        accessOffset, TagT tag,
        const property_list &propList = {}>
  > accessor
  (buffer<
    T, dimensions, AllocatorT>& bufferRef,
    handler &commandGroupHandlerRef,
    range<
      accessRange,
      id<
        accessOffset, TagT tag,
        const property_list &propList = {}>
  );
```

Available when dimensions > 0

```
template <typename AllocatorT, typename TagT>
  accessor<
    buffer<
      T, dimensions, AllocatorT>& bufferRef,
      handler &commandGroupHandlerRef,
      range<
        accessRange,
        id<
          accessOffset, TagT tag,
          const property_list &propList = {}>
      > accessor
      (buffer<
        T, dimensions, AllocatorT>& bufferRef,
        handler &commandGroupHandlerRef,
        range<
          accessRange,
          id<
            accessOffset, TagT tag,
            const property_list &propList = {}>
        );
```

Available when dimensions > 0

```
void swap(accessor &other);
```

Common interface functions [Table 79]

This class supports the following functions in addition to begin(), end(), cbegin(), cend(), rbegin(), rend() and crbegin(), and crend(),

- size() const noexcept;
- size_type size() const noexcept;
- size_type max_size() const noexcept;
- bool empty() const noexcept;
- range<dimensions> get_range() const;

**Local accessor from within kernel functions (class local_accessor) [4.7.6.11]**

data_T can be any C++ type

**Class declaration**

```
template <typename data_T, int dimensions, int access_mode = std::is_const_v<data_T> ? access_mode::read : access_mode::read_write>
class local_accessor;
```

**Constructors and members**

All constructors block until data is available from kernels that access the same underlying buffer.

```
host_accessor();
```

Available when dimensions == 0

```
template <typename AllocatorT, typename TagT>
  host_accessor<
    buffer<
      T, dimensions, AllocatorT>& bufferRef,
      handler &commandGroupHandlerRef,
      range<
        accessRange,
        id<
          accessOffset, TagT tag,
          const property_list &propList = {}>
      > host_accessor
      (buffer<
        T, dimensions, AllocatorT>& bufferRef,
        handler &commandGroupHandlerRef,
        range<
          accessRange,
          id<
            accessOffset, TagT tag,
            const property_list &propList = {}>
        );
```

Available when dimensions > 0

```
template <typename AllocatorT, typename TagT>
  host_accessor<
    buffer<
      T, dimensions, AllocatorT>& bufferRef,
      handler &commandGroupHandlerRef,
      range<
        accessRange,
        id<
          accessOffset, TagT tag,
          const property_list &propList = {}>
    > host_accessor
    (buffer<
      T, dimensions, AllocatorT>& bufferRef,
      handler &commandGroupHandlerRef,
      range<
        accessRange,
        id<
          accessOffset, TagT tag,
          const property_list &propList = {}>
      );
```

Available when dimensions > 0

```
void swap(host_accessor &other);
```

Common interface functions [Table 79]

This class supports the following functions in addition to begin(), end(), cbegin(), cend(), rbegin(), rend() and crbegin(), and crend(),

- size() const noexcept;
- size_type size() const noexcept;
- size_type max_size() const noexcept;
- bool empty() const noexcept;
- range<dimensions> get_range() const;

**Property class constructor [4.7.3.3]**

```
property::no_init::no_init()
```

Available when dimensions > 1

```
__unspecified__ operator[](size_t index) const;
```

Available when dimensions == 1

```
reference operator[](size_t index) const;
```

Available when dimensions > 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

Available when dimensions == 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

Available when dimensions > 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

Available when dimensions == 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

Available when dimensions > 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

Available when dimensions == 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

Available when dimensions > 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

Available when dimensions == 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

Available when dimensions > 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```

Available when dimensions == 1

```
std::add_pointer_t<value_type> get_pointer() const noexcept;
```
Unsampled image accessors [4.7.6.13]
There are two kinds of unsampled image accessors:
- class host_unsampled_image_accessor: From within a kernel function or from within a host task
- class host_unsampled_image_accessor: From host code outside of a host task

Sampled image accessors [4.7.6.14]
There are two kinds of sampled image accessors:
- class sampled_image_accessor: From within a kernel function or from within a host task
- class host_sampled_image_accessor: From host code outside of a host task

Class multi_ptr [4.7.7.1]
The address spaces are global_space, local_space, private_space, and generic_space.

Available if Space == global_space or generic_space
- templated constructor
  `template<typename ElementType, int dimensions, access::address_space Space, access::decorated DecorateAddress> multi_ptr
    (std::nullptr_t);`

Available if Space == local_space
- templated constructor
  `template<typename ElementType, int dimensions, access::address_space::local_space, access::decorated DecorateAddress> multi_ptr
    (std::nullptr_t);`

Available if Space == private_space
-(sampled_image_accessor)
  `template<typename AllocatorT, typename TagT> class sampled_image_accessor
    (sampled_image<dimensions, AllocatorT>& imageRef, const property_list& propList = {});`

Available if Space == address_space::generic_space
- templated constructor
  `template<typename AllocatorT, typename TagT> class sampled_image_accessor
    (sampled_image<dimensions, AllocatorT>& imageRef, commandGroupHandlerRef tag, const property_list& propList = {});`
Unified Shared Memory

Unified Shared Memory is an optional addressing model providing an alternative to the buffer model. See examples on page 15 of this reference guide.

There are three kinds of USM allocations (enum class alloc):
- **host**: in host memory accessible by a device
- **device**: in device memory not accessible by the host
- **shared**: in shared memory accessible by host and device

Class usm_allocator

**Class declaration**

```cpp
template <typename T, usm::alloc AllocKind, size_t Alignment = 0>
struct usm_allocator {
    // Constructor and members
    usm_allocator(const context& ctxt, const device& dev, const property_list& propList = {}) noexcept;
    usm_allocator(const queue& q, const property_list& propList = {}) noexcept;
    // Parameterized allocation functions
    T* allocate(size_t count, const property_list& propList = {}) noexcept;
    T* allocate(size_t count, const queue& q, const property_list& propList = {}) noexcept;
    void dealloc(size_t count, const property_list& propList = {});
    // Class members
    // usm_allocator(U& u, const context& ctxt); // Implicit copy constructor
};
```

**Constructors and members**

- **allocate**: Allocate memory
- **dealloc**: Free memory

**Aliases for decorated pointers**

- `raw_global_ptr`
- `global_ptr`
- `device_ptr`

**Aliases for non-decorated pointers**

- `raw_local_ptr`
- `local_ptr`
- `malloc_device`

**Explicit pointer aliases**

- `malloc_shared`
- `malloc_host`
- `aligned_malloc_shared`
- `aligned_malloc_host`

**Sampler class enums**

The SYCL image_sampler struct contains a configuration for sampling a sampled image.

```cpp
struct image_sampler {
    enum { none, linear, nearest, clamp, linear_unnormalized, clamp_to_edge, unnormalized };
};
```

**Device allocation functions**

- `malloc_device`
- `malloc_host`
- `aligned_malloc_device`
- `aligned_malloc_host`

**Shared allocation functions**

- `malloc_shared`
- `aligned_malloc_shared`
- `malloc_device`
- `aligned_malloc_device`

**Memory deallocation functions**

- `free`
- `free_in_device`

**USM pointer queries**

These queries are available only on the host.

- `usm::get_pointer_type`
- `devices::get_device_pointer_type`

---

**Device allocation functions**

- `malloc_device`
- `malloc_host`
- `aligned_malloc_device`
- `aligned_malloc_host`

**Shared allocation functions**

- `malloc_shared`
- `aligned_malloc_shared`
- `malloc_device`
- `aligned_malloc_device`

**Memory deallocation functions**

- `free`
- `free_in_device`

**USM pointer queries**

These queries are available only on the host.

- `usm::get_pointer_type`
- `devices::get_device_pointer_type`
Ranges and index space identifiers [4.9.1.3]

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. This class supports the standard arithmetic, logical, and relational operators.

Class declaration

```
template <int dimensions > class range;

Constructors and members

range(size_t t_dim0);
range(size_t t_dim0, size_t t_dim1);
range(size_t t_dim0, size_t t_dim1, size_t t_dim2);
size_t t_get(int dimension) const;
size_t &operator[](int dimension);
size_t type operator[](int dimension);
size_t type size_t t_size() const;
```

Class nd_range [4.9.1.4]

This 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.

Class declaration

```
template <int dimensions > class nd_range;

Constructors and members

nd_range(range<dimensions> globalSize, range<dimensions> localSize);
range<dimensions> get_global_range() const;
range<dimensions> get_local_range() const;
range<dimensions> get_group_range() const;
```

Class id [4.9.1.5]

A vector of dimensions that is used to represent an id into a global or local range. It can be used as an index in an accessor of the same rank. This class supports the standard arithmetic, logical, and relational operators.

Class declaration

```
template <int dimensions > class id;

Constructors and members

id(size_t t_dim0);
id(size_t t_dim0, size_t t_dim1);
id(size_t t_dim0, size_t t_dim1, size_t t_dim2);
const range<dimensions> &range();
const range<dimensions> &range(const range<dimensions> &&range);
const id & operator[](int dimension);
const size_t & operator[](int dimension);
size_t type size_t t_size() const;
```

Class h_item [4.9.1.6]

Identifies an instance of the function object executing at each point in a range<int dimensions> passed to a parallel_for call.

Class declaration

```
template <int dimensions > class h_item;

Members

id<dimensions> get_global_id() const;
id<dimensions> get_local_id() const;
id<dimensions> get_group_id() const;
```

Class nd_item [4.9.1.7]

Identifies an instance of the function object executing at each point in an nd_range<int dimensions> passed to a parallel_for call.

Class declaration

```
template <int dimensions > class nd_item;

Members

id<dimensions> get_global_id() const;
```

Class group [4.9.1.8]

Encapsulates all functionality required to represent a particular work-group within a parallel execution. It is not user-constructible.

Class declaration

```
template <int dimensions > class group;
```

Members

id<Dimensions> get_group_id() const;
size_t get_max_local_range() const;
range<Dimensions> get_max_local_range() const;
range<Dimensions> get_group_range() const;
size_t get_group_id() const;
size_t get_max_local_range() const;
size_t t_size() const;
```

Class parallel_for_work_item [4.9.1.9]

Encapsulates all functionality required to represent a parallel for work_item within a parallel execution. It is not user-constructible.

Class declaration

```
template <int dimensions > class parallel_for_work_item;
```

Members

device_event<decorated_global_ptr<dataT>> src;
decorated_local_ptr<dataT> dest;
t_size<1> numElements;
```

Class sub_group [4.9.1.10]

Encapsulates all functionality required to represent a particular sub-group within a parallel execution. It is not user-constructible.

Members

id<1> get_group_id() const;
size_t get_max_local_range() const;
range<1> get_group_range() const;
range<1> get_local_range() const;
uint32_t get_max_local_range() const;
uint32_t get_local_range() const;
bool leader() const;
**Reduction variables [4.9.2]**

Reductions are supported for all SYCL copyable types.

```cpp
template <typename BufferT, typename BinaryOperation>
__unspecified__ reduction(BufferT & vars, handler & cgh, BinaryOperation combiner, const property_list & propList = {});
```

**Reduction class functions [4.9.2.3]**

Defines the interface between a work-item and a reduction variable during the execution of a SYCL kernel, restricting access to the underlying reduction variable.

```cpp
template <typename T, typename BinaryOperation>
__unspecified__ reduction(T * vars, const BinaryOperation & combiner, const property_list & propList = {});
```

**Reduction property constructor [4.9.2.2]**

```cpp
property::reduction::initialize_to_identity::initialize_to_identity() {
    reduction property;
    property.initialize_to_identity();
}
```

**Reduction kernel example [4.9.2]**

```cpp
void use_kernel_bundle(const kernel_bundle< bundle_state::executable > & execkBundle);
```

---

**Command group handler class [4.9.4]**

**Class handler**

A command group handler object can only be constructed by the SYCL runtime. All of the accessors defined in command group scope take as a parameter an instance of the command group handler, and all the kernel invocation functions are member functions of this class.

```cpp
template <typename dataT, int dimensions, access::mode mode, access::target target, access::placeholder isPlaceholder>
void require(accessor<dataT, dimensions, accessMode, accessTarget, placeholder> & accessor);
```

**USM functions**

```cpp
const T& use_kernel_bundle(const kernel_bundle< bundle_state::executable > & execkBundle);
```

---

**Kernel dispatch API**

```cpp
template <typename KernelName, typename KernelType>
void single_task(const KernelType & kernelFunc);
```

---

**Explicit memory operation APIs**

In addition to kernels, command group objects can also be used to perform manual operations on host and device memory by using the API of the command group handler. Following are members of this class.

```cpp
template <typename T_src, int dim_src, access::mode mode_src, target tgt_src, access::placeholder isPlaceholder>
void copy(const T_src & src, T_dest & dest, const Tdest & dim_dest, const Tdest & mode_dest, isPlaceholder & dest);
```

---

**Member function for using a kernel bundle [4.9.4.4]**

```cpp
void use_kernel_bundle(const kernel_bundle< bundle_state::executable > & execkBundle);
```
### Specialization constants [4.9.5]

**Class specialization_id declaration**

```
template<typename T>
class specialization_id;
```

**Class specialization_id constructor**

```
template<class... Args>
explicit constexpr specialization_id(Args&&... args);
```

**Members of class handler**

```
template/auto& SpecName>
void set_specialization_id(
  typename std::remove_reference_t<decltype(SpecName)>::type value;
)
```

**Member of class kernel_handler**

```
typename std::remove_reference_t<decltype(SpecName)>::type
```

---

### Class private_memory [4.10.4.2.3]

To guarantee use of private per-work-item memory, the `private_memory` class can be used to wrap the data.

```cpp
class private_memory {
  public:
    private_memory(const std::vector<Dimensions>& dimensions);
    &operator[(const h_item<Dimensions>&)]
};
```

---

### Exceptions and exception list [4.13.2]

**Class exception is derived from std::exception.**

**Members of class exception**

```
exception(std::error_code ec, const std::string& what_arg);
```

**exception() const noexcept**

```
const std::error_category& error_category() const noexcept;
```

**begin() const**

```
iterator() const;
```

**end() const**

```
() const;
```

**Members of class exception_list**

```
context has_context() const noexcept;
```

**Members of class handler**

```
context get_specialization_id() const;
```

**Members of class kernel_handler**

```
typename std::remove_reference_t<decltype(SpecName)>::type
```

---

### Defining kernels [4.12]

**Defining kernels as named function objects [4.12.1]**

A kernel can be defined as a named function object to provide the same functionality as any C++ function object. For example:

```cpp
class RandomFiller {
  private:
    std::uniform_int_distribution<> r { 1, 100 };
    std::random_device hwRand;
  public:
    RandomFiller(const size_t num) : hwRand() {
      for (size_t i = 0; i < num; ++i) {
        hwRand(gen); // Generate random number
      }
    }
    void operator()(item<1> item) const {
      item[0] = r(hwRand);
    }
}
```

**Defining kernels as lambda functions [4.12.2]**

Kernels may be defined as lambda functions. The name of a lambda function in SYCL may optionally be specified by passing it as a template parameter to the invoking member function. For example:

```cpp
defining kernels as lambda functions:
```
operator T() const noexcept;
  T exchange(T operand,
             memory_order order = default_read_modify_write_order,
             memory_scope scope = default_scope) const noexcept;
bool compare_exchange_weak(T &x, T y,
                            memory_order order = default_read_modify_write_order,
                            memory_scope scope = default_scope) const noexcept;
bool compare_exchange_strong(T &x, T y,
                            memory_order order = default_read_modify_write_order,
                            memory_scope scope = default_scope) const noexcept;

atomic_ref specialization for integral types
Class declaration
template <typename Group, typename T,
         memory_order DEFAULT_ORDER = default_read_order,
         memory_scope DEFAULT_SCOPE = default_scope>
        class atomic_ref
        {...

atomic_ref specialization for floating point
Class declaration
template <typename Group, typename T,
         memory_order DEFAULT_ORDER = default_read_order,
         memory_scope DEFAULT_SCOPE = default_scope>
        class atomic_ref
        {...

Group functions and algorithms

Group functions [4.17.3]
template <typename Group, typename T>
    bool group_broadcast(Group g, T x);
template <typename Group, typename T>
    T group_broadcast(Group g, T x,
                      Group:linear_id:local_linear_id);
template <typename Group, typename T>
    void group_barrier(Group g,
                       memory_scope fence_scope = Group::fence_scope);

template <typename Group, typename T, typename Predicate>
    bool joint_all_off(Group g, Ptr first, Ptr last, Predicate pred);

template <typename Group, typename T, typename Predicate>
    bool all_of(Group g, Ptr first, Predicate pred);

template <typename Group, typename T, typename Predicate>
    bool none_of(Group g, Ptr first, Predicate pred);

template <typename Group, typename T, typename Predicate>
    void group_barrier(Group g,
                       memory_scope fence_scope = Group::fence_scope);

template <typename Group, typename T, typename Predicate>
    T shift_group_left(Group g, T x);

template <typename Group, typename T, typename Predicate>
    T shift_group_right(Group g, T x);

template <typename Group, typename T, typename Predicate>
    void permute_group_by_xor(Group g, T x,
                            Group:linear_id:local_linear_id);

(Continued on next page)
Math functions [4.17.5]
Math functions are available in the namespace sycl for host and device. In all cases below, \( n \) may be 2, 3, 4, 8, or 16. 
\( \text{tf} \) (genfloat in the spec) is type float[\( n \)], double[\( n \)], or half[\( n \)]. \( \text{tf} \) (genfloat) is type float[\( n \)].

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{acos}(\text{f}(x)) )</td>
<td>Arc cosine</td>
</tr>
<tr>
<td>( \text{acosh}(\text{f}(x)) )</td>
<td>Inverse hyperbolic cosine</td>
</tr>
<tr>
<td>( \text{acosi}(\text{f}(x)) )</td>
<td>( \text{acos}(\text{x}) / \pi )</td>
</tr>
<tr>
<td>( \text{asin}(\text{f}(x)) )</td>
<td>Arc sine</td>
</tr>
<tr>
<td>( \text{asinh}(\text{f}(x)) )</td>
<td>Inverse hyperbolic sine</td>
</tr>
<tr>
<td>( \text{asini}(\text{f}(x)) )</td>
<td>( \text{asin}(\text{x}) / \pi )</td>
</tr>
<tr>
<td>( \text{atan}(\text{f}(y, x)) )</td>
<td>Arc tangent</td>
</tr>
<tr>
<td>( \text{atan2}(y, x) )</td>
<td>Arc tangent of ( y / x )</td>
</tr>
<tr>
<td>( \text{atanh}(\text{f}(x)) )</td>
<td>Hyperbolic arc tangent</td>
</tr>
<tr>
<td>( \text{atanpi}(\text{f}(x)) )</td>
<td>( \text{atan}(\text{x}) / \pi )</td>
</tr>
<tr>
<td>( \text{atan2pi}(\text{f}(y, x)) )</td>
<td>( \text{atan}(\text{y}, \text{x}) / \pi )</td>
</tr>
<tr>
<td>( \text{cbrt}(\text{f}(x)) )</td>
<td>Cube root</td>
</tr>
<tr>
<td>( \text{ceil}(\text{f}(x)) )</td>
<td>Round to integer toward + infinity</td>
</tr>
<tr>
<td>( \text{copysign}(\text{f}(x, y)) )</td>
<td>( x ) with sign changed to ( y )</td>
</tr>
<tr>
<td>( \text{cos}(\text{f}(x)) )</td>
<td>Cosine</td>
</tr>
<tr>
<td>( \text{cosh}(\text{f}(x)) )</td>
<td>Hyperbolic cosine</td>
</tr>
<tr>
<td>( \text{cosi}(\text{f}(x)) )</td>
<td>( \text{cos}(\text{x}) )</td>
</tr>
<tr>
<td>( \text{divide}(\text{f}(x), \text{f}(y)) )</td>
<td>( x / y ) (Not available in cl::sycl.)</td>
</tr>
<tr>
<td>( \text{erfc}(\text{f}(x)) )</td>
<td>Complementary error function</td>
</tr>
<tr>
<td>( \text{erf}(\text{f}(x)) )</td>
<td>Calculates error function</td>
</tr>
<tr>
<td>( \text{exp}(\text{f}(x)) )</td>
<td>Exponential base ( e )</td>
</tr>
<tr>
<td>( \text{exp2}(\text{f}(x)) )</td>
<td>Exponential base ( 2 )</td>
</tr>
<tr>
<td>( \text{exp10}(\text{f}(x)) )</td>
<td>Exponential base ( 10 )</td>
</tr>
<tr>
<td>( \text{fabs}(\text{f}(x)) )</td>
<td>Absolute value</td>
</tr>
<tr>
<td>( \text{fdim}(\text{f}(x), y) )</td>
<td>Positive difference between ( x ) and ( y )</td>
</tr>
<tr>
<td>( \text{floor}(\text{f}(x)) )</td>
<td>Round to integer toward infinity</td>
</tr>
<tr>
<td>( \text{fma}(a, b, c) )</td>
<td>Multiply add and then, round</td>
</tr>
<tr>
<td>( \text{fmax}(a, b) )</td>
<td>Return ( y ) if ( y &gt; x ), otherwise ( x )</td>
</tr>
<tr>
<td>( \text{fmin}(a, b) )</td>
<td>Return ( y ) if ( y &lt; x ), otherwise ( x )</td>
</tr>
<tr>
<td>( \text{fmod}(a, b) )</td>
<td>Modulus. Returns ( a - y ) * trunc(( b/y ))</td>
</tr>
<tr>
<td>( \text{fmod2}(a, \text{f}(x)) )</td>
<td>Fractional value in ( x )</td>
</tr>
<tr>
<td>( \text{fround}(\text{f}(x)) )</td>
<td>Extract mantissa and exponent</td>
</tr>
<tr>
<td>( \text{hypot}(\text{f}(x), y) )</td>
<td>Square root of ( x^2 + y^2 )</td>
</tr>
<tr>
<td>( \text{lloqb}(\text{f}(x)) )</td>
<td>Return exponent as an integer value</td>
</tr>
<tr>
<td>( \text{ldexp}(\text{f}(x), y) )</td>
<td>( x ) * ( \text{f}(2^y) )</td>
</tr>
<tr>
<td>( \text{lgamma}(\text{f}(x)) )</td>
<td>Log gamma function</td>
</tr>
<tr>
<td>( \text{lgamma}(\text{f}(x), \text{T}(\pm\text{sign})) )</td>
<td>Log gamma function</td>
</tr>
<tr>
<td>( \text{log}(\text{f}(x)) )</td>
<td>Natural logarithm</td>
</tr>
<tr>
<td>( \text{log2}(\text{f}(x)) )</td>
<td>Base 2 logarithm</td>
</tr>
<tr>
<td>( \text{log10}(\text{f}(x)) )</td>
<td>Base 10 logarithm</td>
</tr>
<tr>
<td>( \text{log1p}(\text{f}(x)) )</td>
<td>( \ln(1 + x) )</td>
</tr>
<tr>
<td>( \text{logb}(\text{f}(x)) )</td>
<td>Return exponent as an integer value</td>
</tr>
<tr>
<td>( \text{maxmag}(\text{f}(x), \text{f}(y)) )</td>
<td>Approximates ( a \times b + c )</td>
</tr>
<tr>
<td>( \text{maxmag}(\text{f}(x), \text{f}(y)) )</td>
<td>Maximum magnitude of ( x ) and ( y )</td>
</tr>
</tbody>
</table>

Integer functions [4.17.6]
Integer functions are available in the namespace sycl. In all cases below, \( n \) may be 2, 3, 4, 8, or 16. If a type in the functions below is shown with \( \text{xbit} \) in its name, this indicates that the type is \( x \) bits in size. Parameter types may also be their vec and marr counterparts.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{abs}(\text{T}(\text{x})) )</td>
<td>Count of leading 0-bits in ( x ) starting at the most significant bit position. If ( x ) is 0, returns the size in bits of the type of ( x ) or component type of ( x ), if ( x ) is a vector type.</td>
</tr>
<tr>
<td>( \text{add}(\text{T}(\text{x}), \text{T}(\text{y})) )</td>
<td>Count of trailing 0-bits in ( x ) if ( x ) is 0, returns the size in bits of the type of ( x ) or component type of ( x ), if ( x ) is a vector type.</td>
</tr>
<tr>
<td>( \text{add}(\text{T}(\text{x}), \text{T}(\text{y})) )</td>
<td>Count of leading 0-bits in ( x ) starting at the most significant bit position. If ( x ) is 0, returns the size in bits of the type of ( x ) or component type of ( x ), if ( x ) is a vector type.</td>
</tr>
<tr>
<td>( \text{add}(\text{T}(\text{x}), \text{T}(\text{y})) )</td>
<td>Count of trailing 0-bits in ( x ) if ( x ) is 0, returns the size in bits of the type of ( x ) or component type of ( x ), if ( x ) is a vector type.</td>
</tr>
<tr>
<td>( \text{add}(\text{T}(\text{x}), \text{T}(\text{y})) )</td>
<td>Count of leading 0-bits in ( x ) starting at the most significant bit position. If ( x ) is 0, returns the size in bits of the type of ( x ) or component type of ( x ), if ( x ) is a vector type.</td>
</tr>
<tr>
<td>( \text{add}(\text{T}(\text{x}), \text{T}(\text{y})) )</td>
<td>Count of trailing 0-bits in ( x ) if ( x ) is 0, returns the size in bits of the type of ( x ) or component type of ( x ), if ( x ) is a vector type.</td>
</tr>
</tbody>
</table>
### Integer functions (cont.)

- `Tint max (Tint x, Tint y)`; returns `x` if `y < x`, otherwise it returns `y`.
- `Tint min (Tint x, Tint y)`; same as `Tint max`.
- `Tint mul_hi (Tint x, Tint y)`; high half of the product of `x` and `y`.
- `Tint popcount (Tint x)`; number of non-zero bits in `x`.
- `Tint rotate (Tint x, Tint l)`; result = `x` rotated left by `l`.
- `Tint sub_sat (Tint x, Tint y)`; `x - y` and saturates the result.

### Common functions [4.17.7]

Common functions are available in the namespace `sycl on host` and device. On the host the vector types use the vec class and on an OpenCL device use the corresponding OpenCL vector types. In all cases below, `n` may be 2, 3, 4, 8, or 16. The built-in functions can take as input float or optionally double and their vec and marray counterparts.

- `Tf (Tf x)`; type `float[n]`.
- `Tf (Tf x, float)`; type `float[n]`.
- `Tf clamp (Tf x, Tf minval, Tf maxval)`; clamp `x` to range given by `minval`, `maxval`.
- `Tf degrees (Tf radians)`; radians to degrees.
- `Tf abs (Tf x)`; max of `x` and `0`.
- `Tf max (Tf x, Tf y)`; max of `x` and `y`.
- `Tf min (Tf x, Tf y)`; min of `x` and `y`.
- `Tf mix (Tf x, Tf y, Tf a)`; linear blend of `x` and `y`.
- `Tf radians (Tf degrees)`; degrees to radians.
- `Tf step (Tf edge, Tf x)`; if `x < edge`, return `0`.
- `Tf smoothstep (Tf edge0, Tf edge1, Tf x)`; Step and interpolate.
- `Tf sign (Tf x)`; Sign of `x`.

### Relational built-in functions [4.17.9]

Relational functions are available in the namespace `sycl on host` and device. In all cases below, `n` may be 2, 3, 4, 8, or 16. If a function is used in the context below with `[bit]` in its name, this indicates that the type is `x` bits in size.

- `Tf (geninteger in the spec)`; type `float[n]`.
- `Tf clamp (Tf x, Tf minval, Tf maxval)`; clamp `x` to range given by `minval`, `maxval`.
- `Tf degrees (Tf radians)`; radians to degrees.
- `Tf abs (Tf x)`; max of `x` and `0`.
- `Tf max (Tf x, Tf y)`; max of `x` and `y`.
- `Tf min (Tf x, Tf y)`; min of `x` and `y`.
- `Tf mix (Tf x, Tf y, Tf a)`; linear blend of `x` and `y`.
- `Tf radians (Tf degrees)`; degrees to radians.
- `Tf step (Tf edge, Tf x)`; if `x < edge`, return `0`.
- `Tf smoothstep (Tf edge0, Tf edge1, Tf x)`; Step and interpolate.
- `Tf sign (Tf x)`; Sign of `x`.

### Geometric Functions [4.17.8]

Geometric functions are available in the namespace `sycl on host` and device. The built-in functions can take as input float or optionally double and their vec and marray counterparts, for dimensions 2, 3, and 4. On the host the vector types use the vec class and on a SYCL device use the corresponding native SYCL backend vector types.

- `Tf (gegenfloat in the spec)`; type `float`, `float2`, `float3`, `float4`, `Tfp (gegendouble in the spec)`; type `double`, `double2`, `double3`, `double4`.

### Kernel attributes [5.8.1]

Attributes are applied as shown in the following examples.

- `=1 [item1 < r]` if `[sycl: reqd_work_group_size(16)]` [kernel code]

### Device attribute functions [5.8.2]

The attribute below is applied to the declaration of a non-kernel device function.

- `sycl: requires(has_aspect...))`
Backends [4.1]
Each Khronos-defined backend is associated with a macro of the form SYCL_BACKEND_BACKEND_NAME. The SYCL backends that are available can be identified by using the enum class backend:

```cpp
enum class backend {
    opencl, vulkan, direct3d, metal, openvx, ...
};
```

**Backend interoperability [4.5.1]**
SYCL applications that rely on SYCL backend-specific behavior must include the SYCL backend-specific header in addition to the sycl/sycl.hpp header. Support for SYCL backend interoperability is optional. A SYCL application using SYCL backend interoperability is considered to be non-generic SYCL.

**Backend type traits, template function**

```cpp
template<backend Backend, typename SyclType>
using backend_traits = ...
```

**Kernel bundles [4.11]**
A kernel bundle is a high-level abstraction which represents a set of kernels that are associated with a context and can be executed on a number of devices, where each device is associated with that same context.

### Bundle states

- **Bundle state:** The device images in the kernel bundle have a format that...
- **bundle state:input** Must be compiled and linked before their kernels can be invoked.
- **bundle state:object** Must be linked before their kernels can be invoked.
- **bundle state:executable** Allows them to be invoked on a device.

**Kernel identifiers [4.11.6]**
Some of the functions related to kernel bundles take an input parameter of type kernel id. It is a class with member:

```cpp
const char *get_name() const noexcept;
```

**Obtaining a kernel identifier [4.11.6]**
Free functions:

```cpp
std::vector<kernel_id> get_kernel_id();
```

**Obtaining a kernel bundle [4.11.7]**
Free functions:

```cpp
template<backend Backend, typename SyclType>
using kernel_bundle = ...
```

**Kernel bundles [4.11.8]**
Returns a SYCL application interoperability native backend object associated with syclObject, which can be used for SYCL application interoperability.

```cpp
template<backend Backend, class Class>
backend_return_t make_device_return_t(backendObject, Class);  // (where Class is one of the classes defined above)
```

**Bundle states [4.11.9]**
Free functions:

```cpp
bool is_compatible(const kernel_bundle &kernel, kernel_device<device &dev, Selector selector>);  // (where device &dev is a device on which the kernel can be executed)
```

**Joining kernel bundles [4.11.10]**
Template metafunction:

```cpp
template<class State>
kernel_bundle<State> join(bundle_state::executable kernel_bundle<State> &bundle1, bundle_state::executable kernel_bundle<State> &bundle2);  // (where bundle1 and bundle2 are kernel bundles)
```

**Online compiling and linking [4.11.11]**
Free functions:

```cpp
kernel_bundle<bundle_state::object> compile(const kernel_bundle<targetContext::kernel_device<device &dev, &propList = {}>> &kernel);  // (where targetContext::kernel_device<device &dev, &propList = {}>> is a device on which the kernel can be executed)
```

**The kernel bundle class [4.11.12]**
Class declaration:

```cpp
template<class State> class kernel_bundle;  // (where State is a bundle state)
```

**Members**
- **bool empty()**
- **backend get_backend()**
- **context get contexto**

(Continued on next page)
Kernel bundles (cont.)

std::vector<device> get_devices() const noexcept;
bool has_kernel(const kernel_id & kernelId) const noexcept;
bool has_kernel(const kernel_id & kernelId, const device & dev) const noexcept;

Available when State == bundle_state::executable

kernel get_kernel(const kernel_id & kernelId) const;

bool has_kernel(const kernel_id & kernelId) const noexcept;
bool has_kernel(const kernel_id & kernelId, const device & dev) const noexcept;

Available when State == bundle_state::executable

std::vector<kernel_id> get_kernel_ids() const;

Available when State == bundle_state::executable

template<typename param> typename param::return_type get_info() const;

Queries using get_info():

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>info::kernel_device_specific::global_work_size</td>
<td>range&lt;3&gt;</td>
</tr>
<tr>
<td>info::kernel_device_specific::work_group_size</td>
<td>size_t</td>
</tr>
<tr>
<td>info::kernel_device_specific::compile_work_group_size</td>
<td>range&lt;3&gt;</td>
</tr>
<tr>
<td>info::kernel_device_specific::preferred_work_group_size_multiple</td>
<td>size_t</td>
</tr>
<tr>
<td>info::kernel_device_specific::private_mem_size</td>
<td>size_t</td>
</tr>
<tr>
<td>info::kernel_device_specific::max_num_sub_groups</td>
<td>uint_32</td>
</tr>
<tr>
<td>info::kernel_device_specific::compile_num_sub_groups</td>
<td>uint_32</td>
</tr>
<tr>
<td>info::kernel_device_specific::max_sub_group_size</td>
<td>uint_32</td>
</tr>
<tr>
<td>info::kernel_device_specific::compile_sub_group_size</td>
<td>uint_32</td>
</tr>
</tbody>
</table>

The kernel class [4.11.13]

backend get_backend() const noexcept;
context get_context() const;

kernel_bundle<bundle_state::executable> get_kernel_bundle() const;
template<typename param> typename param::return_type get_info() const;

Example with USM Shared Allocations

#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  // Create default queue to enqueue work
  queue myQueue;

  // Allocate shared memory bound to the device and context associated to the queue
  // Replacing malloc_shared with malloc_host would yield a correct program that
  // allocated device-visible memory on the host.

  int *data = sycl::malloc_shared<int>(1024, myQueue);

  myQueue.parallel_for(1024, [=](id<1> idx) {
    // Initialize each buffer element with its own rank number starting at 0
    data[idx] = idx;
  }); // End of the kernel function

  myQueue.wait();

  // Print result
  for (int i = 0; i < 1024; i++)
    std::cout << data[i] << std::endl;

  return 0;
}

Example with USM Device Allocations

#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  // Create default queue to enqueue work
  queue myQueue;

  // Allocate shared memory bound to the device and context associated to the queue
  int *data = sycl::malloc_device<int>(1024, myQueue);

  myQueue.parallel_for(1024, [=](id<1> idx) {
    // Initialize each buffer element with its own rank number starting at 0
    data[idx] = idx;
  }); // End of the kernel function

  myQueue.wait();

  int hostData[1024];
  myQueue.memcpy(hostData, data, 1024*sizeof(int));
  myQueue.wait();

  // Print result
  for (int i = 0; i < 1024; i++)
    std::cout << hostData[i] << std::endl;

  return 0;
}
Examples of how to invoke kernels

Example: single_task invoke [4.9.4.2.1]
SYCL provides a simple interface to enqueue a kernel that will be sequentially executed on an OpenCL device.

```cpp
myQueue.submit([&](handler & cgh) {
    cgh.single_task(
        [=] () {
            // [kernel code]
        });
});
```

Examples: parallel_for invoke [4.9.4.2.2]

Example #1
Using a lambda function for a kernel invocation. This variant of parallel_for is designed for when it is not necessary to query the global range of the index space being executed across.

```cpp
myQueue.submit([&](handler & cgh) {
    accessor acc { myBuffer, cgh, write_only);
    cgh.parallel_for(range <1>(numWorkItems),
        [=] (id <1> index) {
            acc[index] = 42.0f;
        });
});
```

Example #2
Invoking a SYCL kernel function with parallel_for using a lambda function and passing an item parameter. This variant of parallel_for is designed for when it is necessary to query the global range of the index space being executed across.

```cpp
myQueue.submit([&](handler & cgh) {
    accessor acc { myBuffer, cgh, write_only};
    cgh.parallel_for(range <1>(numWorkItems),
        [=] (item <1> item) {
            size_t index = item.get_linear_id();
            acc[index] = item.get_global_id();
        });
});
```

Example #3
The following two examples show how a kernel function object can be launched over a 3D grid, with 3 elements in each dimension. In the first case work-item ids range from 0 to 2 inclusive, and in the second case work-item ids run from 1 to 3.

```cpp
myQueue.submit([&](handler & cgh) {
    cgh.parallel_for(
        range <3>(3,3,3), // global range
        [=] (item <3> it) {
            // [kernel code]
        });
});
```

Example #4
Launching sixty-four work-items in a three-dimensional grid with four in each dimension and divided into eight work-groups.

```cpp
myQueue.submit([&](handler & cgh) {
    cgh.parallel_for(
        nd_range <3>(range <3>(4, 4, 4), range <3>(2, 2, 2)), [=] (nd_item <3> item) {
            // [kernel code]
            // internal synchronization
            group_barrier(item.get_group());
            // [kernel code]
        });
});
```

Parallel for hierarchical invoke [4.9.4.2.3]
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. In this case, the h_item provides access to local ids and ranges that reflect both kernel and parallel_for_work_item invocation ranges.

```cpp
myQueue.submit([&](handler & cgh) {
    // issue 8 work-groups of 8 work-items each
cgh.parallel_for_work_group(
        range <3>(2, 2, 2),
        [=] (group <3> g) {
            // [workgroup code]
            int myLocal; // this variable is shared between workitems
            // This variable will be instantiated for each work-item separately
            private_memory<int> myPrivate(g);

            // Issue parallel work-items. The number issued per work-group is determined
            // by the work-group size range of parallel_for_work_group. In this case, 8 work-items
            // will execute the parallel_for_work_item body for each of the 8 work-groups,
            // resulting in 64 executions globally/total.
            myGroup.parallel_for_work_item(&[](h_item <3> myItem) {
                // [work-item code]
                myPrivate(myItem) = 0;
            });

            // Implicit work-group barrier
            // Carry private value across loops
            output[myItem.get_global_id()] = myPrivate(myItem);
        });
    });
});
```

The annual gathering of the international community of SYCL developers, researchers, suppliers, and Khronos SYCL Working Group members to share best practices and to advance the use and evolution of the SYCL standard for C++ programming of heterogeneous platforms.

syclcon.org