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 devics.

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 accessor, 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 &&rhs);
T &operator=(const T &rhs);
T &operator=(T &&rhs);
~T();
friend bool operator==(const T &lhs, const T &rhs);
friend bool operator!=(const T &lhs, const T &rhs);
```

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

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: accessor, 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>
inline constexpr bool is_property_v = is_property<
    propertyT>::value;
template <typename propertyT, typename syclObjectT>
struct is_property_of;template <typename propertyT,
    typename syclObjectT>
inline constexpr bool is_property_of_v = is_property_of<
    propertyT, syclObjectT>::value;

class T {
    ...
    template <typename propertyT>
    bool has_property() const;

    template <typename propertyT>
    propertyT get_property() const;
    ...
};

class property_list {
   public:
     template <typename... propertyTN>
     property_list(propertyTN... props);
};
```

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::device_type::gpu
cpu_selector_v	Select a device according to device type info::device::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.

```
#include <iostream>
#include <sycl/sycl.hpp>
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, range<1> { 1024 } };

      // Create a command group to issue commands to the queue.
      myQueue.submit([&](handler & cgh) {

        // Request access to the buffer without initialization
        accessor writeResult { resultBuf, cgh, write_only, no_init };

        // Enqueue a parallel_for task with 1024 work-items.
        cgh.parallel_for(1024, [=](auto idx) {

          // Initialize each buffer element with its own rank number starting at 0
          writeResult[idx] = idx;

        }); // End of the kernel function

      });    // End of the queue commands

    }        // End of scope, so wait for the queued work to complete

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

    return 0;
}
```

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 data required 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.

Accessor
See accessor class functions in [4.7.6.x] on pages 4 and 5 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 accessors.

The ***application scope*** specifies all other code outside of a command group scope.

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.

```
platform();
template <typename DeviceSelector>
explicit platform(const DeviceSelector &deviceSelector);
backend get_backend() const noexcept;
std::vector<device> get_devices(
    info::device_type = info::device_type::all) const;
template <typename param>
    typename param::return_type get_info() const;
template <typename param>
    typename param::return_type get_backend_info() const;
bool has(aspect asp) const;
static std::vector<platform> get_platforms();
```

Platform information descriptors

version	Return type: std::string
info::platform::name	
info::platform::vendor	

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.

```
device();
template <typename DeviceSelector>
    explicit device(const DeviceSelector &deviceSelector);
backend get_backend() const noexcept;
platform get_platform() const;
bool is_cpu() const;
bool is_gpu() const;
bool is_accelerator() const;
template <typename param>
    typename param::return_type
        get_info() const;
template <typename param>
    typename param::return_type
        get_backend_info() const;
bool has(aspect asp) const;
template <info::partition_property prop> std::vector<device>
    create_sub_devices(size_t count) const;
```

(Continued on next page)

◀ Device class (cont.)

template <info::partition_property prop> std::vector<device>
create_sub_devices(const std::vector<size_t> &*counts*) const;

template <info::partition_property prop>std::vector<device>
create_sub_devices(info::affinity_domain *domain*) const;

static std::vector<device> **get_devices**(
info::device_type *deviceType* = info::device_type::all);

Device queries using get_info()

The following descriptor names are in the info::device namespace.

Descriptor	Return type
device_type	info::device_type
vendor_id	uint32_t
max_compute_units	uint32_t
max_work_item_dimensions	uint32_t
max_work_item_sizes<1>	id<1>
max_work_item_sizes<2>	id<2>
max_work_item_sizes<3>	id<3>
max_work_group_size	size_t
max_num_sub_groups	uint32_t
sub_group_independent_forward_progress	bool
sub_group_sizes	std::vector<size_t>
preferred_vector_width_char preferred_vector_width_short preferred_vector_width_int preferred_vector_width_long preferred_vector_width_float preferred_vector_width_double preferred_vector_width_half	uint32_t
native_vector_width_char native_vector_width_short native_vector_width_int native_vector_width_long native_vector_width_float native_vector_width_double native_vector_width_half	uint32_t
max_clock_frequency	uint32_t
address_bits	uint32_t
max_mem_alloc_size	uint64_t
max_read_image_args	uint32_t
max_write_image_args	uint32_t
image2d_max_width	size_t
image2d_max_height	size_t
image3d_max_width	size_t
image3d_max_height	size_t
image3d_max_depth	size_t
image_max_buffer_size	size_t
max_samplers	uint32_t
max_parameter_size	size_t
mem_base_addr_align	uint32_t
half_fp_config	std::vector<info::fp_config>
single_fp_config	std::vector<info::fp_config>
double_fp_config	std::vector<info::fp_config>
global_mem_cache_type	info::global_mem_cache_type
global_mem_cache_line_size	uint32_t
global_mem_cache_size	uint64_t
global_mem_size	uint64_t
local_mem_type	info::local_mem_type
local_mem_size	uint64_t
error_correction_support	bool
atomic_memory_order_capabilities	std::vector<memory_order>
atomic_fence_order_capabilities	std::vector<memory_order>
atomic_memory_scope_capabilities	std::vector<memory_scope>
atomic_fence_scope_capabilities	std::vector<memory_scope>
profiling_timer_resolution	size_t
is_available	bool
execution_capabilities	std::vector <info::execution_capability>
built_in_kernel_ids	std::vector<kernel_id>
built_in_kernels	std::vector<std::string>
platform	platform
name	std::string
vendor	std::string
driver_version	std::string
version	std::string
backend_version	std::string
aspects	std::vector<aspect>
printf_buffer_size	size_t
parent_device	device
partition_max_sub_devices	uint32_t
partition_properties	std::vector <info::partition_property>
partition_affinity_domains	std::vector <info::partition_affinity_domain>
partition_type_property	info::partition_property
partition_type_affinity_domain	info::partition_affinity_domain

Device aspects [4.6.4.3]

Device aspects are defined in enum class aspect. The core enumerants are shown below. Specific backends may define additional aspects.

cpu	online_compiler
gpu	online_linker
accelerator	queue_profiling
custom	usm_device_allocations
fp16, fp64	usm_host_allocations
emulated	usm_atomic_host_allocations
host_debuggable	usm_shared_allocations
atomic64	usm_atomic_shared_allocations
image	usm_system_allocations

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.

explicit **context**(const property_list &*propList* = {});

explicit **context**(async_handler *asyncHandler*,
const property_list &*propList* = {});

explicit **context**(const device &*dev*,
const property_list &*propList* = {});

explicit **context**(const device &*dev*, async_handler *asyncHandler*,
const property_list &*propList* = {});

explicit **context**(const std::vector<device> &*deviceList*,
const property_list &*propList* = {});

explicit **context**(const std::vector<device> &*deviceList*,
async_handler *asyncHandler*,
const property_list &*propList* = {});

backend **get_backend**() const noexcept;

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

platform **get_platform**() const;

std::vector<device> **get_devices**() const;

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

Context queries using get_info():

The following descriptor names are in the info::contexr namespace.

Descriptor	Return type
platform	platform
devices	std::vector<device>
atomic_memory_order_capabilities	std::vector<memory_order>
atomic_fence_order_capabilities	std::vector<memory_order>
atomic_memory_scope_capabilities	std::vector<memory_scope>
atomic_fence_scope_capabilities	std::vector<memory_scope>

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.

explicit **queue**(const property_list &*propList* = {});

explicit **queue**(const async_handler &*asyncHandler*,
const property_list &*propList* = {});

template <typename DeviceSelector>
explicit **queue**(const DeviceSelector &*deviceSelector*,
const property_list &*propList* = {});

template <typename DeviceSelector>
explicit **queue**(const DeviceSelector &*deviceSelector*,
const async_handler &*asyncHandler*,
const property_list &*propList* = {});

explicit **queue**(const device &*syclDevice*,
const property_list &*propList* = {});

explicit **queue**(const device &*syclDevice*,
const async_handler &*asyncHandler*,
const property_list &*propList* = {});

template <typename DeviceSelector>
explicit **queue**(const context &*syclContext*,
const DeviceSelector &*deviceSelector*,
const property_list &*propList* = {});

template <typename DeviceSelector>
explicit **queue**(const context &*syclContext*,
const DeviceSelector &*deviceSelector*,
const async_handler &*asyncHandler*,
const property_list &*propList* = {});

explicit **queue**(const context &*syclContext*,
const device &*syclDevice*,
const property_list &*propList* = {});

explicit **queue**(const context &*syclContext*,
const device &*syclDevice*,
const async_handler &*asyncHandler*,
const property_list &*propList* = {});

backend **get_backend**() const noexcept;

context **get_context**() const;

device **get_device**() const;

bool **is_in_order**() const;

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

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

template <typename T> event **submit**(T *cgf*);

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

void **wait**();

void **wait_and_throw**();

void **throw_asynchronous**();

Queue queries using get_info()

Descriptor	Return type
info::queue::context	context
info::queue::device	device

Convenience shortcuts

template <typename KernelName, typename KernelType>
event **single_task**(const KernelType &*kernelFunc*);

template <typename KernelName, typename KernelType>
event **single_task**(event *depEvent*,
const KernelType &*kernelFunc*);

template <typename KernelName, typename KernelType>
event **single_task**(const std::vector<event> &*depEvents*,
const KernelType &*kernelFunc*);

template <typename KernelName, int Dims,
typename... Rest>
event **parallel_for**(range<Dims> *numWorkItems*,
Rest&&... *rest*);

template <typename KernelName, int Dims,
typename... Rest>
event **parallel_for**(range<Dims> *numWorkItems*,
event *depEvent*, Rest&&... *rest*);

template <typename KernelName, int Dims,
typename... Rest>
event **parallel_for**(range<Dims> *numWorkItems*,
const std::vector<event> &*depEvents*,
Rest&&... *rest*);

template <typename KernelName, int Dims,
typename... Rest>
event **parallel_for**(nd_range<Dims> *executionRange*,
Rest&&... *rest*);

template <typename KernelName, int Dims,
typename... Rest>
event **parallel_for**(nd_range<Dims> *executionRange*,
event *depEvent*, Rest&&... *rest*);

template <typename KernelName, int Dims,
typename... *rest*>
event **parallel_for**(nd_range<Dims> *executionRange*,
const std::vector<event> &*depEvents*, Rest&&...*rest*);

(Continued on next page) ▶

◄ Queue class (cont.)

USM Functions

```
event memcpy(void* dest, const void* src, size_t numBytes);
event memcpy(void* dest, const void* src, size_t numBytes,
    event depEvent);
event memcpy(void* dest, const void* src, size_t numBytes,
    const std::vector<event> &depEvents);
template <typename T>
    event copy(const T *src, T *dest, size_t count);
template <typename T>
    event copy(const T *src, T *dest, size_t count,
        event depEvent);
template <typename T>
    event copy(const T *src, T *dest, size_t count,
        const std::vector<event> &depEvents);
event memset(void* ptr, int value, size_t numBytes);
event memset(void* ptr, int value, size_t numBytes,
    event depEvent);
event memset(void* ptr, int value, size_t numBytes,
    const std::vector<event> &depEvents);
template <typename T>
    event fill(void* ptr, const T& pattern, size_t count);
template <typename T>
    event fill(void* ptr, const T& pattern, size_t count,
        event depEvent);
template <typename T>
    event fill(void* ptr, const T& pattern, size_t count,
        const std::vector<event> &depEvents);
event prefetch(void* ptr, size_t numBytes);
event prefetch(void* ptr, size_t numBytes, event depEvent);
event prefetch(void* ptr, size_t numBytes,
    const std::vector<event> &depEvents);
event mem_advise(void *ptr, size_t numBytes, int advice);
event mem_advise(void *ptr, size_t numBytes, int advice,
    event depEvent);
event (void *ptr, size_t numBytes, int advice,
    const std::vector<event> &depEvents);
```

Explicit copy functions

```
template <typename T_src, int dim_src,
        access_mode mode_src, target tgt_src,
        access::placeholder isPlaceholder, typename T_dest>
    event copy(accessor<T_src, dim_src, mode_src, tgt_src,
        isPlaceholder> src, std::shared_ptr<T_dest> dest);
template <typename T_src, typename T_dest,
        int dim_dest, access_mode mode_dest,
        target tgt_dest, access::placeholder isPlaceholder>
    event copy(std::shared_ptr<T_src> src, accessor<T_dest,
        dim_dest, mode_dest, tgt_dest isPlaceholder> dest);
template <typename T_src, int dim_src,
        access_mode mode_src, target tgt_src,
        access::placeholder isPlaceholder, typename T_dest>
    event copy(accessor<T_src, dim_src, mode_src, tgt_src,
        isPlaceholder> src, T_dest *dest);
template <typename T_src, typename T_dest,
        int dim_dest, access_mode mode_dest,
        target tgt_dest, access::placeholder isPlaceholder>
    event copy(const T_src *src, accessor<T_dest, dim_dest,
        mode_dest, tgt_dest, isPlaceholder> dest);
template <typename T_src, int dim_src,
        access_mode mode_src, target tgt_src,
        access::placeholder isPlaceholder_src,
        typename T_dest, int dim_dest,
        access_mode mode_dest, target tgt_dest,
        access::placeholder isPlaceholder_dest>
    event copy(accessor<T_src, dim_src, mode_src, tgt_src,
        isPlaceholder_src> src, accessor<T_dest, dim_dest,
        mode_dest, tgt_dest, isPlaceholder_dest> dest);
template <typename T, int dim, access_mode mode,
        target tgt, access::placeholder isPlaceholder>
    event update_host(accessor<T, dim, mode, tgt,
        isPlaceholder> acc);
template <typename T, int dim, access_mode mode,
        target tgt, access::placeholder isPlaceholder>
    event fill(accessor<T, dim, mode, tgt, isPlaceholder> dest,
        const T &src);
```

Queue property class constructors:

```
property::queue::enable_profiling::enable_profiling();
property::queue::in_order::in_order();
```

Queries using get_info():

Descriptor	Return type
info::queue::context	context
info::queue::device	device

Event class [4.6.6]

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

```
event()
backend get_backend() const noexcept;
std::vector<event> get_wait_list();
void wait();
static void wait(const std::vector<event> &eventList);
void wait_and_throw();
static void wait_and_throw(
    const std::vector<event> &eventList);
template <typename param>
        typename param::return_type
    get_info() const;
template <typename param>
    typename param::return_type
    get_backend_info() const;
template <typename param>
    typename param::return_type
    get_profiling_info() const;
```

Event queries using get_info()

Descriptor	Return type
info::event::command_execution_status	info::event::command_status

Queries using get_profiling_info()

Descriptor	Return type
info::event_profiling::command_submit	uint64_t
info::event_profiling::command_start	uint64_t
info::event_profiling::command_end	uint64_t

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(d);
```

The default allocators are buffer_allocator for buffers and image_allocator for images.

Buffer 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

```
template <typename T, int dimensions = 1,
    typename AllocatorT =
        buffer_allocator<std::remove_const_t<T>>>
        class buffer;
```

Member functions

```
buffer(const range<dimensions> &bufferRange,
    const property_list &propList = {});
buffer(const range<dimensions> &bufferRange,
    AllocatorT allocator, const property_list &propList = {});
buffer(T *hostData, const range<dimensions> &bufferRange,
    const property_list &propList = {});
buffer(T *hostData, const range<dimensions> &bufferRange,
    AllocatorT allocator, const property_list &propList = {});
buffer(const T *hostData,
    const range<dimensions> &bufferRange,
    const property_list &propList = {});
buffer(const T *hostData,
    const range<dimensions> &bufferRange,
    AllocatorT allocator, const property_list &propList = {});
```

Available if dimensions == 1 and std::data(container) is convertible to T*

```
template <typename Container>
    buffer(Container &container, AllocatorT allocator,
        const property_list &propList = {});
template <typename Container>
    buffer(Container &container,
        const property_list &propList = {});
```

```
buffer(const std::shared_ptr<T> &hostData,
    const range<dimensions> &bufferRange,
    AllocatorT allocator, const property_list &propList = {});
buffer(const std::shared_ptr<T> &hostData,
    const range<dimensions> &bufferRange,
    const property_list &propList = {});
buffer(const std::shared_ptr<T[ ]> &hostData,
    const range<dimensions> &bufferRange,
    AllocatorT allocator, const property_list &propList = {});
buffer(const std::shared_ptr<T[ ]> &hostData,
    const range<dimensions> &bufferRange,
    const property_list &propList = {});
template <class InputIterator>
    buffer<T, 1>(InputIterator first, InputIterator last,
        AllocatorT allocator, const property_list &propList = {});
template <class InputIterator>
    buffer<T, 1>(InputIterator first, InputIterator last,
        const property_list &propList = {});
buffer(buffer &b,const id<dimensions> &baseIndex,
    const range<dimensions> &subRange);
get_range()
byte_size()
size_t size() const noexcept;
AllocatorT get_allocator() const;
template <access_mode mode = access_mode::read_write,
        target targ = target::device> accessor<T,
        dimensions, mode, targ>
    get_access(handler &commandGroupHandler);
template <access_mode mode = access_mode::read_write,
        target targ = target::device> accessor<T,
        dimensions, mode, targ>
    get_access(
        handler &commandGroupHandler, range<dimensions>
        accessRange, id<dimensions> accessOffset = {});
template<typename... Ts> auto get_access(Ts...);
template<typename... Ts> auto get_host_access(Ts...);
template <typename Destination = std::nullptr_t>
    void set_final_data(Destination finalData = nullptr);
void set_write_back(bool flag = true);
bool is_sub_buffer() const;
template <typename ReinterpretT, int ReinterpretDim>
    buffer<ReinterpretT, ReinterpretDim,
        typename std::allocator_traits<AllocatorT>::template
        rebind_alloc<ReinterpretT>>
    reinterpret(range<ReinterpretDim> reinterpretRange)
        const;
```

Available when ReinterpretDim == 1 or when (ReinterpretDim == dimensions) && (sizeof(ReinterpretT) == sizeof(T))

```
template <typename ReinterpretT,
        int ReinterpretDim = dimensions>
    buffer<ReinterpretT, ReinterpretDim,
        typename std::allocator_traits<
        AllocatorT>::template rebind_alloc<ReinterpretT>>
    reinterpret() const;
```

Buffer property class constructors:

```
property::buffer::use_host_ptr::use_host_ptr()
property::buffer::use_mutex::use_mutex(std::mutex &mutexRef)
property::buffer::context_bound::context_bound(context boundContext)
```

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.

enum class image_format values:	
r8g8b8a8_unorm	r16g16b16a16_uint
r16g16b16a16_unorm	r32b32g32a32_uint
r8g8b8a8_sint	r16b16g16a16_sfloat
r16g16b16a16_sint	r32g32b32a32_sfloat
r32b32g32a32_sint	b8g8r8a8_unorm
r8g8b8a8_uint	

Unsampled images [4.7.3.1]

Class declaration

```
template <int dimensions = 1,
        typename AllocatorT = sycl::image_allocator>
        class unsampled_image;
```

Constructors and members

```
unsampled_image(image_format format,
    const range<dimensions> &rangeRef,
    const property_list &propList = {});
unsampled_image(image_format format,
    const range<dimensions> &rangeRef, AllocatorT allocator,
    const property_list &propList = {});
unsampled_image(void *hostPointer, image_format format,
    const range<dimensions> &rangeRef,
    const property_list &propList = {});
unsampled_image(void *hostPointer,
    image_format format, const range<dimensions> &rangeRef,
    AllocatorT allocator, const property_list &propList = {});
unsampled_image(std::shared_ptr<void> &hostPointer,
    image_format format, const range<dimensions> &rangeRef,
    const property_list &propList = {});
unsampled_image(std::shared_ptr<void> &hostPointer,
    image_format format, const range<dimensions> &rangeRef,
    AllocatorT allocator, const property_list &propList = {});
```

Available when dimensions > 1

```
unsampled_image(image_format format,
    const range<dimensions> &rangeRef,
    const range<dimensions - 1> &pitch,
    const property_list &propList = {});
```

Available when dimensions > 1

```
unsampled_image(image_format format,
    const range<dimensions> &rangeRef,
    const range<dimensions - 1> &pitch,
    AllocatorT allocator, const property_list &propList = {});
unsampled_image(void *hostPointer,
    image_format format,
    const range<dimensions> &rangeRef,
    const range<dimensions - 1> &pitch,
    property_list &propList = {});
unsampled_image(void *hostPointer,
    image_format format,
    const range<dimensions> &rangeRef,
    const range<dimensions - 1> &pitch,
    AllocatorT allocator, const property_list &propList = {});
unsampled_image(std::shared_ptr<void> &hostPointer,
    image_format format,
    const range<dimensions> &rangeRef,
    const range<dimensions - 1> &pitch,
    AllocatorT allocator, const property_list &propList = {});
unsampled_image(std::shared_ptr<void> &hostPointer,
    image_format format,
    const range<dimensions> &rangeRef,
    const range<dimensions - 1> &pitch,
    const property_list &propList = {});
```

```
range<dimensions> get_range() const;
```

Available when dimensions > 1

```
range<dimensions-1> get_pitch() const;
```

```
size_t size() const noexcept;
size_t byte_size() const noexcept;
AllocatorT get_allocator() const;
template<typename... Ts>
    auto get_access(Ts... args);
template<typename... Ts>
    auto get_host_access(Ts... args);
template <typename Destination = std::nullptr_t>
    void set_final_data(Destination finalData = std::nullptr);
void set_write_back(bool flag = true);
```

Sampled images [4.7.3.2]

Class declaration

```
template <int dimensions = 1,
    typename AllocatorT = sycl::image_allocator>
    class sampled_image;
```

Constructors and members

```
sampled_image(const void *hostPointer,
    image_format format, image_sampler sampler,
    const range<dimensions> &rangeRef,
    const property_list &propList = {});
sampled_image(std::shared_ptr<const void> &hostPointer,
    image_format format, image_sampler sampler,
    const range<dimensions> &rangeRef,
    const property_list &propList = {});
```

Available when dimensions > 1

```
sampled_image(const void *hostPointer,
    image_format format, image_sampler sampler,
    const range<dimensions> &rangeRef,
    const range<dimensions - 1> &pitch,
    const property_list &propList = {});
sampled_image(std::shared_ptr<const void> &hostPointer,
    image_format format, image_sampler sampler,
    const range<dimensions> &rangeRef,
    const range<dimensions - 1> &pitch,
    const property_list &propList = {});
```

```
range<dimensions> get_range() const;
range<dimensions-1> get_pitch() const;
size_t byte_size() const noexcept;
size_t size() const noexcept;
template<typename... Ts>
    auto get_access(Ts... args);
template<typename... Ts>
    auto get_host_access(Ts... args);
```

Image property constructors and members [4.7.3.3]

```
property::image::use_host_ptr::use_host_ptr();
property::image::use_mutex::
    use_mutex(std::mutex &mutexRef);
property::image::context_bound::
    context_bound(context boundContext);
std::mutex *property::image::use_mutex::
    get_mutex_ptr() const;
context property::image::context_bound::get_context() const;
```

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).
- Unsampled image accessors of two kinds:
  - From within a kernel function or from within a host task (4.7.6.13, class unsampled_image_accessor).
  - From host code outside of a host task (4.7.6.13, class host_unsampled_image_accessor).
- Sampled image accessors of two kinds:
  - From within a kernel function or from within a host task (4.7.6.14, class sampled_image_accessor).
  - From host code outside of a host task (4.7.6.14, class host_sampled_image_accessor).

enum class access_mode [4.7.6.2]

read	write	read_write

Accessor property class constructor [4.7.6.4]

This is used in all accessor classes.

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

Access targets [4.7.6.9]

target::device	buffer access from kernel function via device global memory
target::host_task	buffer access from a host task

enum class access::address_space [4.7.7.1]

global_space	Accessible to all work-items in all work-groups
constant_space	Global space that is constant
local_space	Accessible to all work-items in a single work-group
private_space	Accessible to a single work-item
generic_space	Virtual address space overlapping global, local, and private

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;
```

Constructors and members

```
accessor();
```

Available when dimensions == 0

```
template <typename AllocatorT>
    accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
    const property_list &propList = {});
```

Available when dimensions == 0

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

Available when dimensions > 0

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

Available when dimensions > 0

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

template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &*bufferRef*, handler &*commandGroupHandlerRef*, range<dimensions> *accessRange*, const property_list &*propList* = {});

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

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

Available when dimensions > 0

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

id<dimensions> **get_offset**() const;

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

bool **is_placeholder**() const;

template <access::decorated IsDecorated>
accessor_ptr<IsDecorated> **get_multi_ptr**() const noexcept;

Common interface functions [Table 79]

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

size_type **byte_size**() const noexcept;

size_type **size**() const noexcept;

size_type **max_size**() const noexcept;

bool **empty**() const noexcept;

range<dimensions> **get_range**() const;

Available when dimensions == 0

operator **reference**() const;

Available when dimensions > 0

reference **operator[]**(id<dimensions> *index*) const;

Available when dimensions > 1

__unspecified__ &**operator[]**(size_t *index*) const;

Available when dimensions == 1

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

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

Property class constructor [4.7.3.3]

property::no_init::no_init()

Buffer accessor for host code outside of a command (class host_accessor) [4.7.6.10]

Class declaration

template <typename dataT, int dimensions,
access_mode accessMode =
(std::is_const_v<dataT> ? access_mode::read
: access_mode::read_write)>
class host_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>
host_accessor(buffer<dataT, 1, AllocatorT> &*bufferRef*, const property_list &*propList* = {});

Available when dimensions > 0

template <typename AllocatorT>
host_accessor(buffer<dataT, dimensions, AllocatorT> &*bufferRef*, const property_list &*propList* = {});

template <typename AllocatorT, typename TagT>
host_accessor(buffer<dataT, dimensions, AllocatorT> &*bufferRef*, TagT *tag*, const property_list &*propList* = {});

Available when dimensions > 0

template <typename AllocatorT>
host_accessor(buffer<dataT, dimensions, AllocatorT> &*bufferRef*, range<dimensions> *accessRange*, const property_list &*propList* = {});

template <typename AllocatorT, typename TagT>
host_accessor(buffer<dataT, dimensions, AllocatorT> &*bufferRef*, range<dimensions> *accessRange*, TagT *tag*, const property_list &*propList* = {});

template <typename AllocatorT>
host_accessor(buffer<dataT, dimensions, AllocatorT> &*bufferRef*, range<dimensions> *accessRange*, id<dimensions> *accessOffset*, const property_list &*propList* = {});

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

id<dimensions> **get_offset**() const;

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(), crbegin(), and crend().

size_t **byte_size**() const noexcept;

size_t **size**() const noexcept;

size_t **max_size**() const noexcept;

bool **empty**() const noexcept;

range<dimensions> **get_range**() const;

Available when dimensions == 0

operator **reference**() const;

Available when dimensions > 0

reference **operator[]**(id<dimensions> *index*) const;

Available when dimensions > 1

__unspecified__ &**operator[]**(size_t *index*) const;

Available when dimensions == 1)

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

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

Property class constructor [4.7.3.3]

property::no_init::no_init()

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

dataT can be any C++ type

Class declaration

template <typename dataT, int dimensions>
class local_accessor;

Constructors and members

local_accessor();

Available when dimensions == 0

local_accessor(handler &*commandGroupHandlerRef*, const property_list &*propList* = {});

Available when dimensions > 0

local_accessor(range<dimensions> *allocationSize*, handler &*commandGroupHandlerRef*, const property_list &*propList* = {});

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

template <access::decorated IsDecorated>
accessor_ptr<IsDecorated>
get_multi_ptr() const noexcept;

Common interface functions [Table 79]

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

size_t **byte_size**() const noexcept;

size_t **size**() const noexcept;

size_t **max_size**() const noexcept;

bool **empty**() const noexcept;

range<dimensions> **get_range**() const;

Available when dimensions == 0

operator **reference**() const;

Available when dimensions > 0

reference **operator[]**(id<dimensions> *index*) const;

Available when dimensions > 1

__unspecified__ &**operator[]**(size_t *index*) const;

Available when == 1

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

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

Property class constructor [4.7.3.3]

property::no_init::no_init()

Unsampled image accessors [4.7.6.13]

There are two kinds of unsampled image accessors:

- class 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

unsampled_image_accessor

Class declaration

```
template <typename dataT, int dimensions,
        access_mode accessMode,
        image_target accessTarget = image_target::device >
    class unsampled_image_accessor;
```

Constructors

```
template <typename AllocatorT>
    unsampled_image_accessor(
    unsampled_image<dimensions, AllocatorT> &imageRef,
    handler &commandGroupHandlerRef,
    const property_list &propList = {});
```

```
template <typename AllocatorT, typename TagT>
    unsampled_image_accessor(
    unsampled_image<dimensions, AllocatorT> &imageRef,
    handler &commandGroupHandlerRef,
    TagT tag, const property_list &propList = {});
```

host_unsampled_image_accessor

Class declaration

```
template <typename dataT, int dimensions,
    access_mode accessMode>
    class host_unsampled_image_accessor;
```

Constructors

```
template <typename AllocatorT>
    host_unsampled_image_accessor(
    unsampled_image<dimensions, AllocatorT> &imageRef,
    const property_list &propList = {});
```

```
template <typename AllocatorT, typename TagT>
    host_unsampled_image_accessor(
    unsampled_image<dimensions, AllocatorT> &imageRef,
    TagT tag, const property_list &propList = {});
```

Available to both unsampled image accessor types

```
size_t size() const noexcept;
```

Available when (accessMode == access_mode::read)

```
template <typename coordT>
    dataT read(const coordT &coords) const;
```

Available when (accessMode == access_mode::write)

```
template <typename coordT>
    void write(const coordT &coords,
    const dataT &color) const;
```

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

sampled_image_accessor

Class declaration

```
template <typename dataT, int dimensions,
        image_target accessTarget = image_target::device>
        class sampled_image_accessor;
```

Constructors

```
template <typename AllocatorT>
        sampled_image_accessor(
        sampled_image<dimensions, AllocatorT> &imageRef,
        handler &commandGroupHandlerRef,
        const property_list &propList = {});
```

```
template <typename AllocatorT, typename TagT>
        sampled_image_accessor(
        sampled_image<dimensions, AllocatorT> &imageRef,
        handler &commandGroupHandlerRef, TagT tag,
        const property_list &propList = {});
```

host_sampled_image_accessor

Class declaration

```
template <typename dataT, int dimensions>
        class host_sampled_image_accessor;
```

Constructor

```
template <typename AllocatorT>
        host_sampled_image_accessor(
        sampled_image<dimensions, AllocatorT> &imageRef,
        const property_list &propList = {});
```

Available to both sampled image accessor types

```
size_t size() const noexcept;
```

if dimensions == 1, coordT = float
if dimensions == 2, coordT = float2
if dimensions == 4, coordT = float4

```
template <typename coordT>
        dataT read(const coordT &coords) const;
```

Class multi_ptr [4.7.7.1]

The address spaces are global_space, local_space, private_space, and generic_space.

Class declaration

```
template <typename ElementType,
        access::address_space Space,
        access::decorated DecorateAddress>
    class multi_ptr;
```

Members: Constructors

```
multi_ptr();
multi_ptr(const multi_ptr&);
multi_ptr(multi_ptr&&);
explicit multi_ptr(multi_ptr<ElementType, Space,
    yes>::pointer);
multi_ptr(std::nullptr_t);
```

Available if Space == global_space or generic_space

```
template <int dimensions, access::mode Mode,
        access::placeholder isPlaceholder>
    multi_ptr(accessor<ElementType, dimensions, Mode,
        target::device, isPlaceholder>);
```

Available if Space == global_space or generic_space

```
template <int dimensions>
    multi_ptr(local_accessor<ElementType, dimensions>);
```

Members: Assignment and access operators

```
multi_ptr &operator=(const multi_ptr&);
multi_ptr &operator=(multi_ptr&&);
multi_ptr &operator=(std::nullptr_t);
```

Available if Space == address_space::generic_space && ASP != access::address_space::constant_space

```
template<access::address_space ASP,
        access::decorated IsDecorated>
    multi_ptr &operator=(
        const multi_ptr<value_type, ASP, IsDecorated>&);
template<access::address_space ASP,
        access::decorated IsDecorated>
    multi_ptr &operator=(
        multi_ptr<value_type, ASP, IsDecorated>&&);
```

```
reference operator*() const;
pointer operator->() const;
pointer get() const;
std::add_pointer_t<value_type> get_raw() const;
__unspecified__ * get_decorated() const;
```

Members: Conversions

Cast to private_ptr, available if Space == address_space::generic_space

```
explicit operator multi_ptr<value_type,
    access::address_space::private_space,
    DecorateAddress>();
explicit operator multi_ptr<const value_type,
    access::address_space::private_space,
    DecorateAddress>() const;
```

Cast to global_ptr, available if Space == address_space::generic_space

```
explicit operator multi_ptr<value_type,
    access::address_space::global_space, DecorateAddress>();
explicit operator multi_ptr<const value_type,
    access::address_space::global_space,
    DecorateAddress>() const;
explicit operator multi_ptr<value_type,
    access::address_space::local_space,
    DecorateAddress>();
```

Cast to local_ptr, available if Space == address_space::generic_space

```
explicit operator multi_ptr<const value_type,
    access::address_space::local_space,
    DecorateAddress>() const;
```

Implicit conversions to a multi_ptr

Implicit conversion to a multi_ptr<void>. Only available when value_type is not const-qualified.

```
template<access::decorated DecorateAddress>
    operator multi_ptr<void, Space, DecorateAddress>() const;
```

Implicit conversion to a multi_ptr<const void>. Only available when value_type is const-qualified.

```
template<access::decorated DecorateAddress>
    operator multi_ptr<const void, Space,
    DecorateAddress>() const;
```

Implicit conversion to multi_ptr<const value_type, Space>.

```
template<access::decorated DecorateAddress>
    operator multi_ptr<const value_type, Space,
    DecorateAddress>() const;
```

Implicit conversion to the non-decorated version of multi_ptr. Only available when is_decorated is true.

```
operator multi_ptr<value_type, Space,
    access::decorated::no>() const;
```

Implicit conversion to the decorated version of multi_ptr. Only available when is_decorated is false.

```
operator multi_ptr<value_type, Space,
    access::decorated::yes>() const;
```

```
void prefetch(size_t numElements) const;
```

Members: Arithmetic operators

The multi_ptr class supports the standard arithmetic and relational operators.

Class multi_ptr specialized for void and const void [4.7.7.1]

Class declaration

```
template <access::address_space Space,
        access::decorated DecorateAddress>
    class multi_ptr<VoidType, Space, DecorateAddress>
```

DecorateAddress: yes, no
VoidType: void or const void

Members: Constructors

```
multi_ptr();
multi_ptr(const multi_ptr&);
multi_ptr(multi_ptr&&);
explicit multi_ptr(multi_ptr<VoidType, Space, yes>::pointer);
multi_ptr(std::nullptr_t);
```

Available if Space == global_space

```
template <typename ElementType, int dimensions,
        access_mode Mode, access::placeholder isPlaceholder>
    multi_ptr(accessor<ElementType, dimensions, Mode,
        target::device, isPlaceholder>);
```

Available if Space == local_space

```
template <typename ElementType, int dimensions>
    multi_ptr(local_accessor<ElementType, dimensions>);
```

Assignment operators

```
multi_ptr &operator=(const multi_ptr&);
multi_ptr &operator=(multi_ptr&&);
multi_ptr &operator=(std::nullptr_t);
```

Members

```
pointer get() const;
explicit operator pointer() const;
template <typename ElementType>
    explicit operator multi_ptr<ElementType, Space,
    DecorateAddress>() const;
```

Only available when is_decorated is true.

```
operator multi_ptr<value_type, Space,
    access::decorated::no>() const;
```

(Continued on next page) ▶

◀ multi_ptr specialized (cont.)

Only available when is_decorated is false.

operator multi_ptr<value_type, Space, access::decorated::yes>() const;

operator multi_ptr<const void, Space, DecorateAddress>() const;

template <access::address_space Space, access::decorated DecorateAddress, typename ElementType> multi_ptr<ElementType, Space, DecorateAddress> **address_space_cast**(ElementType *);

Operators

The multi_ptr class supports the standard arithmetic and relational operators.

Explicit pointer aliases [4.7.7.2]

Aliases to class multi_ptr for each specialization of access::address_space:

global_ptr
local_ptr
private_ptr

Aliases for non-decorated pointers:

raw_global_ptr
raw_local_ptr
raw_private_ptr

Aliases for decorated pointers:

decorated_global_ptr
decorated_local_ptr
decorated_private_ptr

Sampler class enums [4.7.8]

The SYCL image_sampler struct contains a configuration for sampling a sampled_image.

```
struct image_sampler {
  addressing_mode addressing;
  coordinate_mode coordinate;
  filtering_mode filtering;
};
```

addressing
mirrored_repeat
repeat
clamp_to_edge
clamp
none

filtering
nearest
linear

coordinate
normalized
unnormalized

Unified Shared Memory [4.8]

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 [4.8.3]

Class declaration

template <typename T, usm::alloc AllocKind, size_t Alignment = 0> class usm_allocator;

Constructors 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;

template <class U> **usm_allocator**(usm_allocator<U, AllocKind, Alignment> const &) noexcept;

T ***allocate**(size_t *count*);

void **deallocate**(T **Ptr*, size_t *count*);

Allocators only compare equal if they are of the same USM kind, alignment, context, and device.

template <class U, usm::alloc AllocKindU, size_t AlignmentU> friend bool **operator==**(const usm_allocator<T, AllocKind, Alignment> &, const usm_allocator<U, AllocKindU, AlignmentU> &);

template <class U, usm::alloc AllocKindU, size_t AlignmentU> friend bool **operator!=**(const usm_allocator<T, AllocKind, Alignment> &, const usm_allocator<U, AllocKindU, AlignmentU> &);

malloc-style API [4.8.3]

Device allocation functions [4.8.3.2]

void* sycl::**malloc_device**(size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T> T* sycl::**malloc_device**(size_t *count*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**malloc_device**(size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T> T* sycl::**malloc_device**(size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

void* sycl::**aligned_alloc_device**(size_t *alignment*, size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T> T* sycl::**aligned_alloc_device**(size_t *alignment*, size_t *count*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**aligned_alloc_device**(size_t *alignment*, size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T> T* sycl::**aligned_alloc_device**(size_t *alignment*, size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

Host allocation functions [4.8.3.3]

void* sycl::**malloc_host**(size_t *numBytes*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T> T* sycl::**malloc_host**(size_t *count*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**malloc_host**(size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T> T* sycl::**malloc_host**(size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

void* sycl::**aligned_alloc_host**(size_t *alignment*, size_t *numBytes*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T> T* sycl::**aligned_alloc_host**(size_t *alignment*, size_t *count*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**aligned_alloc_host**(size_t *alignment*, size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T> T* sycl::**aligned_alloc_host**(size_t *alignment*, size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

Shared allocation functions [4.8.3.4]

void* sycl::**malloc_shared**(size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T> T* sycl::**malloc_shared**(size_t *count*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**malloc_shared**(size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T> T* sycl::**malloc_shared**(size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

void* sycl::**aligned_malloc_shared**(size_t *alignment*, size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

template <typename T> T* sycl::**aligned_malloc_shared**(size_t *alignment*, size_t *count*, const device& *syclDevice*, const context& *syclContext*, const property_list &*propList* = {});

void* sycl::**aligned_malloc_shared**(size_t *alignment*, size_t *numBytes*, const queue& *syclQueue*, const property_list &*propList* = {});

template <typename T> T* sycl::**aligned_malloc_shared**(size_t *alignment*, size_t *count*, const queue& *syclQueue*, const property_list &*propList* = {});

Parameterized allocation functions [4.8.3.5]

void* sycl::**malloc**(size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, usm::alloc *kind*, const property_list &*propList* = {});

template <typename T> T* sycl::**malloc**(size_t *count*, const device& *syclDevice*, const context& *syclContext*, usm::alloc *kind*, const property_list &*propList* = {});

void* sycl::**malloc**(size_t *numBytes*, const queue& *syclQueue*, usm::alloc *kind*, const property_list &*propList* = {});

template <typename T> T* sycl::**malloc**(size_t *count*, const queue& *syclQueue*, usm::alloc *kind*, const property_list &*propList* = {});

void* sycl::**aligned_alloc**(size_t *alignment*, size_t *numBytes*, const device& *syclDevice*, const context& *syclContext*, usm::alloc *kind*, const property_list &*propList* = {});

template <typename T> T* sycl::**aligned_alloc**(size_t *alignment*, size_t *count*, const device& *syclDevice*, const context& *syclContext*, usm::alloc *kind*, const property_list &*propList* = {});

void* sycl::**aligned_alloc**(size_t *alignment*, size_t *numBytes*, const queue& *syclQueue*, usm::alloc *kind*, const property_list &*propList* = {});

template <typename T> T* sycl::**aligned_alloc**(size_t *alignment*, size_t *count*, const queue& *syclQueue*, usm::alloc *kind*, const property_list &*propList* = {});

Memory deallocation functions [4.8.3.6]

void sycl::**free**(void* *ptr*, sycl::context& *syclContext*);

void sycl::**free**(void* *ptr*, sycl::queue& *syclQueue*);

USM pointer queries [4.8.4]

These queries are available only on the host.

usm::alloc **get_pointer_type**(const void **ptr*, const context &*ctxt*)

sycl::device **get_pointer_device**(const void **ptr*, const context &*ctxt*)

Ranges and index space identifiers [4.9.1]

Class range [4.9.1.1]

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 = 1> class range;

Constructors and members

range(size_t *dim0*);

range(size_t *dim0*, size_t *dim1*);

range(size_t *dim0*, size_t *dim1*, size_t *dim2*);

size_t **get**(int *dimension*) const;

size_t &**operator[]**(int *dimension*);

size_t **operator[]**(int *dimension*) const;

size_t **size**() const;

Class nd_range [4.9.1.2]

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 = 1> 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 [4.9.1.3]

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 = 1> class id;

Constructors and members

id();

id(size_t *dim0*);

id(size_t *dim0*, size_t *dim1*);

id(size_t *dim0*, size_t *dim1*, size_t *dim2*);

id(const range<dimensions> &*range*);

id(const item<dimensions> &*item*);

size_t **get**(int *dimension*) const;

size_t &**operator[]**(int *dimension*);

size_t **operator[]**(int *dimension*) const;

Class item [4.9.1.4]

Identifies an instance of the function object executing at each point in a range. It is passed to a **parallel_for** call or returned by member functions of h_item.

Class declaration

template <int dimensions = 1, bool with_offset = true>
class item;

Members

id<dimensions> **get_id**() const;

size_t **get_id**(int *dimension*) const;

size_t **operator[]**(int *dimension*) const;

range<dimensions> **get_range**() const;

size_t **get_range**(int *dimension*) const;

Available if with_offset is false

operator item<dimensions, true>() const;

Available if dimensions == 1

operator size_t() const;

size_t **get_linear_id**() const;

Class nd_item [4.9.1.5]

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 = 1> class nd_item;

Members

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

size_t **get_global_id**(int *dimension*) const;

size_t **get_global_linear_id**() const;

id<dimensions> **get_local_id**() const;

size_t **get_local_id**(int *dimension*) const;

size_t **get_local_linear_id**() const;

group<dimensions> **get_group**() const;

size_t **get_group**(int *dimension*) const;

size_t **get_group_linear_id**() const;

range<dimensions> **get_group_range**() const;

size_t **get_group_range**(int *dimension*) const;

range<dimensions> **get_global_range**() const;

size_t **get_global_range**(int *dimension*) const;

range<dimensions> **get_local_range**() const;

size_t **get_local_range**(int *dimension*) const;

nd_range<dimensions> **get_nd_range**() const;

template <typename dataT>
device_event **async_work_group_copy**(
decorated_local_ptr<dataT> *dest*,
decorated_global_ptr<dataT> *src*,
size_t *numElements*) const;

template <typename dataT>
device_event **async_work_group_copy**(
decorated_global_ptr<dataT> *dest*,
decorated_local_ptr<dataT> *src*,
size_t *numElements*) const;

template <typename dataT>
device_event **async_work_group_copy**(
decorated_local_ptr<dataT> *dest*,
decorated_global_ptr<dataT> *src*,
size_t *numElements*, size_t *srcStride*) const;

template <typename dataT>
device_event **async_work_group_copy**(
decorated_global_ptr<dataT> *dest*,
decorated_local_ptr<dataT> *src*,
size_t *numElements*, size_t *destStride*) const;

template <typename... eventTN>
void **wait_for**(eventTN... *events*) const;

Class h_item [4.9.1.6]

Identifies an instance of a group::**parallel_for_work_item** function object executing at each point in a local range<int dimensions> passed to a **parallel_for_work_item** call or to the corresponding **parallel_for_work_group** call if no range is passed to the **parallel_for_work_item** call.

Class declaration

template <int dimensions> class h_item;

Members

item<dimensions, false> **get_global**() const;

item<dimensions, false> **get_local**() const;

item<dimensions, false> **get_logical_local**() const;

item<dimensions, **false> get_physical_local**() const;

range<dimensions> **get_global_range**() const;

size_t **get_global_range**(int *dimension*) const;

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

size_t **get_global_id**(int *dimension*) const;

range<dimensions> **get_local_range**() const;

size_t **get_local_range**(int *dimension*) const;

id<dimensions> **get_local_id**() const;

size_t **get_local_id**(int *dimension*) const;

range<dimensions> **get_logical_local_range**() const;

size_t **get_logical_local_range**(int *dimension*) const;

id<dimensions> **get_logical_local_id**() const;

size_t **get_logical_local_id**(int *dimension*) const;

range<dimensions> **get_physical_local_range**() const;

size_t **get_physical_local_range**(int *dimension*) const;

id<dimensions> **get_physical_local_id**() const;

size_t **get_physical_local_id**(int *dimension*) const;

Class group [4.9.1.7]

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

Class declaration

template <int dimensions = 1> class group;

Members

id<Dimensions> **get_group_id**() const;

size_t **get_group_id**(int *dimension*) const;

id<Dimensions> **get_local_id**() const;

size_t **get_local_id**(int *dimension*) const;

range<Dimensions> **get_local_range**() const;

size_t **get_local_range**(int *dimension*) const;

range<Dimensions> **get_group_range**() const;

size_t **get_group_range**(int *dimension*) const;

range<Dimensions> **get_max_local_range**() const;

size_t **operator[]**(int *dimension*) const;

size_t **get_group_linear_id**() const;

size_t **get_local_linear_id**() const;

size_t **get_group_linear_range**() const;

size_t **get_local_linear_range**() const;

bool **leader**() const;

template<typename workItemFunctionT>
void **parallel_for_work_item**(
const workItemFunctionT &*func*) const;

template<typename workItemFunctionT>
void **parallel_for_work_item**(range<dimensions>
logicalRange, const workItemFunctionT &*func*) const;

template <typename dataT>
device_event **async_work_group_copy**(
decorated_local_ptr<dataT> *dest*,
decorated_global_ptr<dataT> *src*,
size_t *numElements*) const;

template <typename dataT>
device_event **async_work_group_copy**(
decorated_global_ptr<dataT> *dest*,
decorated_local_ptr<dataT> *src*,
size_t *numElements*) const;

template <typename dataT>
device_event **async_work_group_copy**(
decorated_local_ptr<dataT> *dest*,
decorated_global_ptr<dataT> *src*,
size_t *numElements*, size_t *srcStride*) const;

template <typename dataT>
device_event **async_work_group_copy**(
decorated_global_ptr<dataT> *dest*,
decorated_local_ptr<dataT> *src*, size_t *numElements*,
size_t *destStride*) const;

template <typename... eventTN>
void **wait_for**(eventTN... *events*) const;

Class sub_group [4.9.1.8]

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;

id<1> **get_local_id**() const;

range<1> **get_local_range**() const;

range<1> **get_group_range**() const;

range<1> **get_max_local_range**() const;

uint32_t **get_group_linear_id**() const;

uint32_t **get_local_linear_id**() const;

uint32_t **get_group_linear_range**() const;

uint32_t **get_local_linear_range**() const;

bool **leader**() const;

Reduction variables [4.9.2]

Reductions are supported for all SYCL copyable types.

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

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

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

Available if has_known_identity<BinaryOperation, BufferT::value_type>::value is false

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

Available if has_known_identity<BinaryOperation, T>::value is false

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

Available if has_known_identity<BinaryOperation, T>::value is false

```
template <typename T, typename Extent,
    typename BinaryOperation>
  __unspecified__ reduction(span<T, Extent> vars,
    const T& identity, BinaryOperation combiner);
```

Reduction property constructor [4.9.2.2]

```
property::reduction::initialize_to_identity::initialize_to_identity()
```

Reducer 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.

```
template <typename T>
  void operator+=(reducer<T,plus<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator*=(reducer<T,multiplies<T>,0>& accum,
    const T& partial);
```

Available only for integral types

```
template <typename T>
  void operator&=(reducer<T,bit_and<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator|=(reducer<T,bit_or<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator^=(reducer<T,bit_xor<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator++(reducer<T,plus<T>,0>& accum);
```

Member functions

```
void id combine(const T& partial) const;
__unspecified__ &operator[](size_t index) const;
T identity() const;
```

Operators

```
template <typename T>
  void operator+=(reducer<T, plus<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator*=(reducer<T, multiplies<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator|=(reducer<T, bit_or<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator&=(reducer<T, bit_and<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator^=(reducer<T, bit_xor<T>,0>& accum,
    const T& partial);

template <typename T>
  void operator++(reducer<T, plus<T>,0>& accum);
```

Reduction kernel example [4.9.2]

The following example shows how to write a reduction kernel that performs two reductions simultaneously on the same input values, computing both the sum of all values in a buffer and the maximum value in the buffer.

```
buffer<int> valuesBuf { 1024 };
{
  // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
  host_accessor a { valuesBuf };
  std::iota(a.begin(), a.end(), 0);
}

// Buffers with just 1 element to get the reduction results
int sumResult = 0;
buffer<int> sumBuf { &sumResult, 1 };
int maxResult = 0;
buffer<int> maxBuf { &maxResult, 1 };

myQueue.submit([&](handler& cgh) {

  // Input values to reductions are standard accessors
  auto inputValues = valuesBuf.get_access<access_mode::read>(cgh);

  // Create temporary objects describing variables with
  // reduction semantics
  auto sumReduction = reduction(sumBuf, cgh, plus<>());
  auto maxReduction = reduction(maxBuf, cgh, maximum<>());

  // parallel_for performs two reduction operations
  // For each reduction variable, the implementation:
  // - Creates a corresponding reducer
  // - Passes a reference to the reducer to the lambda as a parameter
  cgh.parallel_for(range<1>{1024},
    sumReduction, maxReduction,
    [=](id<1> idx, auto& sum, auto& max) {
      // plus<>() corresponds to += operator, so sum can be
      // updated via += or combine()
      sum += inputValues[idx];

      // maximum<>() has no shorthand operator, so max
      // can only be updated via combine()
      max.combine(inputValues[idx]);
  });
});

// sumBuf and maxBuf contain the reduction results once
// the kernel completes
assert(maxBuf.get_host_access()[0] == 1023
  && sumBuf.get_host_access()[0] == 523776);
```

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.

```
template <typename dataT, int dimensions,
    access_mode accessMode, access_target
    accessTarget, access::placeholder isPlaceholder>
  void require(accessor<dataT, dimensions, accessMode,
    accessTarget, placeholder> acc);

void depends_on(event depEvent);

void depends_on(const std::vector<event> &depEvents);
```

Backend interoperability interface

```
template <typename T>
  void set_arg(int argIndex, T && arg);

template <typename... Ts>
  void set_args(Ts &&... args);
```

Kernel dispatch API

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

template <typename KernelName, int dimensions,
    typename... Rest>
  void parallel_for(range<dimensions> numWorkItems,
    Rest&&... rest);

template <typename KernelName, int dimensions,
    typename... Rest>
  void parallel_for(nd_range<dimensions>
    executionRange,Rest&&... rest);

template <typename KernelName, typename
    WorkgroupFunctionType, int dimensions>
  void parallel_for_work_group(
    range<dimensions> numWorkGroups,
    const WorkgroupFunctionType &kernelFunc);

template <typename KernelName, typename
    WorkgroupFunctionType, int dimensions>
  void parallel_for_work_group(
    range<dimensions> numWorkGroups,
    range<dimensions> workGroupSize,
    const WorkgroupFunctionType &kernelFunc);

void single_task(const kernel_name &kernelObject);

template <int dimensions>
  void parallel_for(range<dimensions> numWorkItems,
    const kernel &kernelObject);

template <int dimensions>
  void parallel_for(range<dimensions> ndRange,
    const kernel &kernelObject);
```

USM functions

```
void memcpy(void *dest, const void *src, size_t numBytes);

template <typename T>
  void copy(const T *src, T *dest, size_t count);

void memset(void *ptr, int value, size_t numBytes);

template <typename T>
  void fill(void *ptr, const T &pattern, size_t count);

void prefetch(void *ptr, size_t numBytes);

void mem_advise(void *ptr, size_t numBytes, int advice);
```

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 copy API of the command group handler. Following are members of class handler.

```
template <typename T_src, int dim_src,
    access_mode mode_src, target tgt_src,
    access::placeholder isPlaceholder,
    typename T_dest>
  void copy(accessor<T_src, dim_src,
    mode_src, tgt_src, isPlaceholder> src,
    std::shared_ptr<T_dest> dest);

template <typename T_src, typename T_dest, int dim_dest,
    access_mode mode_dest, target tgt_dest,
    access::placeholder isPlaceholder>
  void copy(std::shared_ptr<T_src> src,
    accessor<T_dest, dim_dest, mode_dest,
    tgt_dest, isPlaceholder> dest);

template <typename T_src, int dim_src,
    access_mode mode_src, target tgt_src,
    access::placeholder isPlaceholder, typename T_dest>
  void copy(accessor<T_src, dim_src, mode_src, tgt_src,
    isPlaceholder> src, T_dest *dest);

template <typename T_src, typename T_dest,
    int dim_dest, access_mode mode_dest,
    target tgt_dest, access::placeholder isPlaceholder>
  void copy(const T_src *src, accessor<T_dest, dim_dest,
    mode_dest, tgt_dest, isPlaceholder> dest);

template <typename T_src, int dim_src,
    access_mode mode_src, target tgt_src,
    access::placeholder isPlaceholder_src, typename T_dest,
    int dim_dest, access::mode mode_dest,
    access::target tgt_dest,
    access::placeholder isPlaceholder_dest>
  void copy(
    accessor<T_src, dim_src, mode_src, tgt_src,
    isPlaceholder_src> src,
    accessor<T_dest, dim_dest, mode_dest, tgt_dest,
    isPlaceholder_dest> dest);

template <typename T, int dim, access_mode mode, target tgt,
    access::placeholder isPlaceholder>
  void update_host(
    accessor<T, dim, mode, tgt, isPlaceholder> acc);

template<typename T, int dim, access_mode mode, target tgt,
    access::placeholder isPlaceholder>
  void fill(accessor<T, dim, mode, tgt, isPlaceholder> dest,
    const T& src);

template<auto& S>
  typename std::remove_reference_t<decltype(S)>::type
  get_specialization_constant();
```

Member function for using a kernel bundle [4.9.4.4]

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

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_constant(
        typename std::remove_reference_t<decltype(
        SpecName)>::type value);
template<auto& SpecName>
        typename std::remove_reference_t<decltype(
        SpecName)>::type get_specialization_constant();
```

Member of class kernel_handler
```
template<auto& SpecName>
        typename std::remove_reference_t<decltype(
        SpecName)>::type get_specialization_constant();
```

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.
```
class private_memory {
    public:
        private_memory(const group<Dimensions> &);
        T &operator()(const h_item<Dimensions> &id);
};
```

Classes exception & 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(std::error_code ec, const char * what_arg);
exception(std::error_code ec);
exception(int ev, const std::error_category& ecat,
    const std::string& what_arg);
exception(int ev, const std::error_category& ecat,
    const char* what_arg);
exception(int ev, const std::error_category& ecat);
exception(context ctx, std::error_code ec,
    const std::string& what_arg);
exception(context ctx, std::error_code ec,
    const char* what_arg);
exception(context ctx, std::error_code ec);
exception(context ctx, int ev, const std::error_category& ecat,
    const std::string& what_arg);
exception(context ctx, int ev, const std::error_category& ecat,
    const char* what_arg);
exception(context ctx, int ev,
    const std::error_category& ecat);
const std::error_code& code() const noexcept;
const std::error_category& category() const noexcept;
bool has_context() const noexcept;
context get_context() const;
```

Members of class exception_list
```
size_type size() const;
iterator begin() const;
iterator end() const;
```

Helper functions
Free functions:
```
const std::error_category& sycl_category() noexcept;
template<backend b>
    const std::error_category& error_category_for() noexcept;
std::error_condition make_error_condition(errc e) noexcept;
std::error_code make_error_code(errc e) noexcept;
```

Standard error codes (enum errc)

runtime	invalid
kernel	memory_allocation
accessor	platform
nd_range	profiling
event	feature_not_supported
kernel_argument	kernel_not_supported
build	backend_mismatch

Host tasks [4.10]

Class interop_handle [4.10.1-2]
An abstraction over the queue which is being used to invoke the host task and its associated device and context.

Member functions
```
backend get_backend() const noexcept;
```

Available only if the optional interoperability function get_native taking a buffer is available and if accTarget is target::device.
```
template <backend Backend, typename dataT,
        int dims, access_mode accMode, target accTarget,
        access::placeholder isPlaceholder>
    backend_return_t<Backend, buffer<dataT, dims>>
    get_native_mem(const accessor<dataT, dims, accMode,
        accTarget, isPlaceholder> &bufferAcc) const;
```

Available only if the optional interoperability function get_native taking an unsampled_image is available.
```
template <backend Backend, typename dataT, int dims,
        access_mode accMode>
    backend_return_t<Backend, unsampled_image<dims>>
    get_native_mem(
        const unsampled_image_accessor<dataT, dims,
        accMode, image_target::device> &imageAcc) const;
```

Available only if the optional interoperability function get_native taking a queue is available.
```
template <backend Backend, typename dataT, int dims>
    backend_return_t<Backend, sampled_image<dims>>
    get_native_mem(
            const sampled_image_accessor<dataT, dims,
        image_target::device> &imageAcc) const;
```

Available only if the optional interoperability function get_native taking a queue is available.
```
template <backend Backend>
    backend_return_t<Backend, queue>
    get_native_queue() const;
```

Available only if the optional interoperability function get_native taking a device is available.
```
template <backend Backend>
    backend_return_t<Backend, device>
    get_native_device() const;
```

Available only if the optional interoperability function get_native taking a context is available.
```
template <backend Backend> backend_return_t<
    backend, context>
    get_native_context() const;
```

Addition to class handler [4.10.3]
```
template <typename T>
    void host_task(T &&hostTaskCallable);
```

Defining kernels [4.12]
Functions that are executed on a SYCL device are SYCL kernel functions. A kernel containing a SYCL kernel function is enqueued on a device queue in order to be executed on that device.
The return type of the SYCL kernel function is void.
There are two ways of defining kernels: as named function objects or as lambda functions.

Defining kernels as named function objects [4.12.1]
A kernel can be defined as a named function object type and provide the same functionality as any C++ function object. For example:
```
class RandomFiller {
    public:
        RandomFiller(accessor<int> ptr)
                : ptr_ { ptr } {
            std::random_device hwRand;
            std::uniform_int_distribution<> r { 1, 100 };
            randomNum_ = r(hwRand);
        }
        void operator()(item<1> item) const {
            ptr_[item.get_id()] = get_random(); }
        int get_random() { return randomNum_; }

    private:
        accessor<int> ptr_;
        int randomNum_;
};

void workFunction(buffer<int, 1>& b, queue& q,
            const range<1> r) {
        myQueue.submit([&](handler& cgh) {
            accessor ptr { buf, cgh };
            RandomFiller filler { ptr };

            cgh.parallel_for(r, filler);
        });
}
```

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:
```
// Explicit kernel names can be optionally forward declared
// at namespace scope
class MyKernel;

myQueue.submit([&](handler& h) {

   // Explicitly name kernel with previously forward
   // declared type
   h.single_task<MyKernel>([=]{
       // [kernel code]
   });

   // Explicitly name kernel without forward declaring type
at
   // namespace scope. Must still be forward declarable at
   // namespace scope, even if not declared at that scope
   h.single_task<class MyOtherKernel>([=]{
       // [kernel code]
   });
});
```

Synchronization and atomics [4.15]

Enums

class memory_order				
relaxed	acquire	release	acq_rel	seq_cst

class memory_scope				
work_item	sub_group	work_group	device	system

atomic_fence [4.15.1]
Free function:
```
void atomic_fence(memory_order order,
  memory_scope scope);
```

Class device_event [4.15.2]
Class device_event encapsulates a single SYCL device event which is available only within SYCL kernel functions and can be used to wait for asynchronous operations within a SYCL kernel function to complete. The class has an unspecified ctor and one other member:
```
void wait() noexcept;
```

class atomic_ref [4.15.3]

Class declaration
```
template <typename T, memory_order DefaultOrder,
        memory_scope DefaultScope, access::address_
        space Space = access::address_space::generic_space>
    class atomic_ref;
```

Constructors and members
```
explicit atomic_ref(T& ref);
atomic_ref(const atomic_ref&) noexcept;
bool is_lock_free() const noexcept;
void store(T operand,
    memory_order order = default_write_order,
    memory_scope scope = default_scope) const noexcept;
T operator=(T desired) const noexcept;
T load(memory_order order = default_read_order,
    memory_scope scope = default_scope) const noexcept;
```

(Continued on next page) ▶

◀ Synchronization and atomics (cont.)

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 &*expected*, T *desired*,
memory_order *success*, memory_order *failure*,
memory_scope *scope* = default_scope) const noexcept;

bool **compare_exchange_weak**(T &*expected*, T *desired*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

bool **compare_exchange_strong**(T &*expected*, T *desired*,
memory_order *success*, memory_order *failure*,
memory_scope *scope* = default_scope) const noexcept;

bool **compare_exchange_strong**(T &*expected*, T *desired*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

atomic_ref specialized for integral types

Class declaration

template <memory_order DefaultOrder, memory_scope
DefaultScope, access::address_space
Space = access::address_space::generic_space>
class atomic_ref <Integral, DefaultOrder, DefaultScope,
Space>;

Members

Integral **fetch_add**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

Integral **fetch_sub**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

Integral **fetch_and**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

Integral **fetch_or**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

Integral **fetch_min**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

Integral **fetch_max**(Integral *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

OP is ++, --

Integral **operatorOP**(int) const noexcept;

Integral **operatorOP**() const noexcept;

OP is +=, -=, &=, |=, ^=

Integral **operatorOP**(Integral) const noexcept;

atomic_ref specialized for floating point

Class declaration

template <memory_order DefaultOrder, memory_scope
DefaultScope, access::address_space
Space = access::address_space::generic_space>
class atomic_ref<Floating, DefaultOrder, DefaultScope,
Space>;

Members

Floating **fetch_add**(Floating *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

Floating **fetch_sub**(Floating *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

Floating **fetch_min**(Floating *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

Floating **fetch_max**(Floating *operand*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

OP is +=, -=

Floating **operatorOP**(Floating) const noexcept;

atomic_ref specialized for pointer types

Class declaration

template <typename T, memory_order DefaultOrder,
memory_scope DefaultScope, access::address_space
Space = access::address_space::generic_space>
class atomic_ref<T*, DefaultOrder, DefaultScope, Space>;

Constructors and members

explicit **atomic_ref**(T*&);

atomic_ref(const atomic_ref&) noexcept;

void **store**(T* *operand*,
memory_order *order* = default_write_order,
memory_scope *scope* = default_scope) const noexcept;

T* **operator=**(T* *desired*) const noexcept;

T* **load**(memory_order *order* = default_read_order,
memory_scope *scope* = default_scope) const noexcept;

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* &*expected*, T* *desired*,
memory_order *success*, memory_order *failure*,
memory_scope *scope* = default_scope) const noexcept;

bool **compare_exchange_weak**(T* &*expected*, T* *desired*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

bool **compare_exchange_strong**(T* &*expected*, T* *desired*,
memory_order *success*, memory_order *failure*,
memory_scope *scope* = *default_scope*) const noexcept;

bool **compare_exchange_strong**(T* &*expected*, T* *desired*,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

T* **fetch_add**(difference_type,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

T* **fetch_sub**(difference_type,
memory_order *order* = default_read_modify_write_order,
memory_scope *scope* = default_scope) const noexcept;

OP is ++, --

T* **operatorOP**(int) const noexcept;

T* **operatorOP**() const noexcept;

OP is +=, -=

T* **operatorOP**(difference_type) const noexcept;

Scalar data types [4.15]

SYCL supports the C++ fundamental data types (not within the sycl namespace) and the data types byte and half (in the sycl namespace).

Class device_event [4.15.2]

This class encapsulates a single SYCL device event which is available only within SYCL kernel functions and can be used to wait for asynchronous operations within a SYCL kernel function to complete. This class contains an unspecified ctor and one other member:

void **wait**() noexcept;

Class stream [4.16]

Enums

stream_manipulator				
dec	noshowbase	noshowpos	endl	hexfloat
hex	showbase	showpos	fixed	defaultfloat
oct			scientific	flush

Constructor and members

stream(size_t *totalBufferSize*, size_t *workItemBufferSize*,
handler& *cgh*, const property_list &*propList* = {});

size_t **size**() const noexcept;

size_t **get_work_item_buffer_size**() const;

Non-member function

template <typename T>
const stream&
operator<<(const stream& *os*, const T &*rhs*);

Function Objects [4.17.2]

SYCL provides a number of function objects in the sycl namespace on host and device that obey C++ conversion and promotion rules.

template <typename T=void>
struct plus {
T **operator()**(const T & *x*, const T & *y*) const;
};

template <typename T=void>
struct multiplies {
T **operator()**(const T & *x*, const T & *y*) const;
};

template <typename T=void>
struct bit_and {
T **operator()**(const T & *x*, const T & *y*) const;
};

template <typename T=void>
struct bit_or {
T **operator()**(const T & *x*, const T & *y*) const;
};

template <typename T=void>
struct bit_xor {
T **operator()**(const T & *x*, const T & *y*) const;
};

template <typename T=void>
struct logical_and {
T **operator()**(const T & *x*, const T & *y*) const;
};

template <typename T=void>
struct logical_or {
T **operator()**(const T & *x*, const T & *y*) const;
};

template <typename T=void>
struct minimum {
T **operator()**(const T & *x*, const T & *y*) const;
};

template <typename T=void>
struct maximum {
T **operator()**(const T & *x*, const T & *y*) const;
};

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_type *local_linear_id*);

template <typename Group, typename T>
T **group_broadcast**(Group *g*, T *x*, Group::id_type *local_id*);

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

Group algorithms [4.17.4]

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

template <typename Group, typename T,
typename Predicate>
bool **any_of_group**(Group *g*, T *x*, Predicate *pred*);

template <typename Group>
bool **any_of_group**(Group *g*, bool *pred*);

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

template <typename Group, typename T,
typename Predicate>
bool **all_of_group**(Group *g*, T *x*, Predicate *pred*);

template <typename Group>
bool **all_of_group**(Group *g*, bool *pred*);

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

template <typename Group, typename T,
typename Predicate>
bool **none_of_group**(Group *g*, T *x*, Predicate *pred*);

template <typename Group>
bool **none_of_group**(Group *g*, bool *pred*);

template <typename Group, typename T>
T **shift_group_left**(Group *g*, T *x*,
Group::linear_id_type *delta* = 1);

template <typename Group, typename T>
T **shift_group_right**(Group *g*, T *x*,
Group::linear_id_type *delta* = 1);

template <typename Group, typename T>
T **permute_group_by_xor**(Group *g*, T *x*,
Group::linear_id_type *mask*);

template <typename Group, typename T>
T **select_from_group**(Group *g*, T *x*,
Group::id_type *remote_local_id*);

template <typename Group, typename Ptr,
typename BinaryOperation>
std::iterator_traits<Ptr>::value_type **joint_reduce**(Group *g*,
Ptr *first*, Ptr *last*, BinaryOperation *binary_op*);

template <typename Group, typename Ptr, typename T,
typename BinaryOperation>
T **joint_reduce**(Group *g*, Ptr *first*, Ptr *last*, T *init*,
BinaryOperation *binary_op*);

(Continued on next page) ▶

◀ Group functions and algorithms (cont.)

template <typename Group, typename T, typename BinaryOperation>
T **reduce_over_group**(Group *g*, T *x*, BinaryOperation *binary_op*);

template <typename Group, typename V, typename T, typename BinaryOperation>
T **reduce_over_group**(Group *g*, V *x*, T *init*, BinaryOperation *binary_op*);

template <typename Group, typename InPtr, typename OutPtr, typename BinaryOperation>
OutPtr **joint_exclusive_scan**(Group *g*, InPtr *first*, InPtr *last*, OutPtr *result*, BinaryOperation *binary_op*);

template <typename Group, typename InPtr, typename OutPtr, typename T, typename BinaryOperation>
T **joint_exclusive_scan**(Group *g*, InPtr *first*, InPtr *last*, OutPtr *result*, T *init*, BinaryOperation *binary_op*);

template <typename Group, typename T, typename BinaryOperation>
T **exclusive_scan_over_group**(Group *g*, T *x*, BinaryOperation *binary_op*);

template <typename Group, typename V, typename T, typename BinaryOperation>
T **exclusive_scan_over_group**(Group *g*, V *x*, T *init*, BinaryOperation *binary_op*);

template <typename Group, typename InPtr, typename OutPtr, typename BinaryOperation>
OutPtr **joint_inclusive_scan**(Group *g*, InPtr *first*, InPtr *last*, OutPtr *result*, BinaryOperation *binary_op*);

template <typename Group, typename InPtr, typename OutPtr, typename T, typename BinaryOperation>
T **joint_inclusive_scan**(Group *g*, InPtr *first*, InPtr *last*, OutPtr *result*, BinaryOperation *binary_op*, T *init*);

template <typename Group, typename T, typename BinaryOperation>
T **inclusive_scan_over_group**(Group *g*, T *x*, BinaryOperation *binary_op*);

template <typename Group, typename V, typename T, typename BinaryOperation>
T **inclusive_scan_over_group**(Group *g*, V *x*, BinaryOperation *binary_op*, T *init*);

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.

Tf (genfloat in the spec) is type float[*n*], double[*n*], or half[*n*].
Tff (genfloatf) is type float[*n*].
Tfd (genfloatd) is type double[*n*].
Th (genfloath) is type half[*n*].
sTf (sgenfloat) is type float, double, or half.
Ti (genint) is type int[*n*].
uTi (ugenint) is type unsigned int or uint*n*.
uTli (ugenlonginteger) is unsigned long int, ulonglong*n*, ulong*n*, unsigned long long int.

N indicates **native variants**, available in sycl::native.
H indicates **half variants**, available in sycl::halfprecision, implemented with a minimum of 10 bits of accuracy.

Tf **acos** (*Tf x*)	Arc cosine
Tf **acosh** (*Tf x*)	Inverse hyperbolic cosine
Tf **acospi** (*Tf x*)	acos (*x*) / π
Tf **asin** (*Tf x*)	Arc sine
Tf **asinh** (*Tf x*)	Inverse hyperbolic sine
Tf **asinpi** (*Tf x*)	asin (*x*) / π
Tf **atan** (*Tf y_over_x*)	Arc tangent
Tf **atan2** (*Tf y*, *Tf x*)	Arc tangent of *y* / *x*
Tf **atanh** (*Tf x*)	Hyperbolic arc tangent
Tf **atanpi** (*Tf x*)	atan (*x*) / π
Tf **atan2pi** (*Tf y*, *Tf x*)	atan2 (*y*, *x*) / π
Tf **cbrt** (*Tf x*)	Cube root
Tf **ceil** (*Tf x*)	Round to integer toward + infinity
Tf **copysign** (*Tf x*, *Tf y*)	*x* with sign changed to sign of *y*
Tf **cos** (*Tf x*) *Tff* **cos** (*Tff x*) N H	Cosine
Tf **cosh** (*Tf x*)	Hyperbolic cosine
Tf **cospi** (*Tf x*)	cos (π *x*)
Tff **divide** (*Tff x*, *Tff y*) N H	*x* / *y* (Not available in cl::sycl.)
Tf **erfc** (*Tf x*)	Complementary error function
Tf **erf** (*Tf x*)	Calculates error function
Tf **exp** (*Tf x*) *Tff* **exp** (*Tff x*) N H	Exponential base e
Tf **exp2** (*Tf x*) *Tff* **exp2** (*Tff x*) N H	Exponential base 2
Tf **exp10** (*Tf x*) *Tff* **exp10** (*Tff x*) N H	Exponential base 10
Tf **expm1** (*Tf x*)	e^x -1.0
Tf **fabs** (*Tf x*)	Absolute value
Tf **fdim** (*Tf x*, *Tf y*)	Positive difference between *x* and *y*
Tf **floor** (*Tf x*)	Round to integer toward infinity
Tf **fma** (*Tf a*, *Tf b*, *Tf c*)	Multiply and add, then round
Tf **fmax** (*Tf x*, *Tf y*) *Tf* **fmax** (*Tf x*, *sTf y*)	Return *y* if *x* < *y*, otherwise it returns *x*
Tf **fmin** (*Tf x*, *Tf y*) *Tf* **fmin** (*Tf x*, *sTf y*)	Return *y* if *y* < *x*, otherwise it returns *x*
Tf **fmod** (*Tf x*, *Tf y*)	Modulus. Returns *x* – *y* * trunc (*x*/*y*)
Tf **fract** (*Tf x*, *Tf *iptr*)	Fractional value in *x*
Tf **frexp** (*Tf x*, *Ti *exp*)	Extract mantissa and exponent
Tf **hypot** (*Tf x*, *Tf y*)	Square root of $x^2 + y^2$
Ti **ilogb** (*Tf x*)	Return exponent as an integer value
Tf **ldexp** (*Tf x*, *Ti k*) *doublen* **ldexp** (*doublen x*, int *k*)	$x * 2^n$
Tf **lgamma** (*Tf x*)	Log gamma function
Tf **lgamma_r** (*Tf x*, *Ti *signp*)	Log gamma function
Tf **log** (*Tf x*) *Tff* **log** (*Tff x*) N H	Natural logarithm
Tf **log2** (*Tf x*) *Tff* **log2** (*Tff x*) N H	Base 2 logarithm
Tf **log10** (*Tf x*) *Tff* **log10** (*Tff x*) N H	Base 10 logarithm
Tf **log1p** (*Tf x*)	ln (1.0 + *x*)
Tf **logb** (*Tf x*)	Return exponent as an integer value
Tf **mad** (*Tf a*, *Tf b*, *Tf c*)	Approximates *a* * *b* + *c*
Tf **maxmag** (*Tf x*, *Tf y*)	Maximum magnitude of *x* and *y*
Tf **minmag** (*Tf x*, *Tf y*)	Minimum magnitude of *x* and *y*
Tf **modf** (*Tf x*, *Tf *iptr*)	Decompose floating-point number
Tff **nan** (*uTi nancode*) *Tfd* **nan** (*uTli nancode*)	Quiet NaN (Return is scalar when *nancode* is scalar)
Tf **nextafter** (*Tf x*, *Tf y*)	Next representable floating-point value after *x* in the direction of *y*
Tf **pow** (*Tf x*, *Tf y*)	Compute *x* to the power of *y*
Tf **pown** (*Tf x*, *Ti y*)	Compute x^y, where *y* is an integer
Tf **powr** (*Tf x*, *Tf y*) *Tff* **powr** (*Tff x*, *Tff y*) N *Tff* **powr** (*Tff x*, *Th y*) H	Compute x^y, where *x* is >= 0
Tff **recip** (*Tff x*) N H	1 / *x* (Not available in cl::sycl.)
Tf **remainder** (*Tf x*, *Tf y*)	Floating point remainder
Tf **remquo** (*Tf x*, *Tf y*, *Ti *quo*)	Remainder and quotient
Tf **rint** (*Tf x*)	Round to nearest even integer
Tf **rootn** (*Tf x*, *Ti y*)	Compute *x* to the power of 1/*y*
Tf **round** (*Tf x*)	Integral value nearest to *x* rounding
Tf **rsqrt** (*Tf x*) *Tff* **rsqrt** (*Tff x*) N H	Inverse square root
Tf **sin** (*Tf x*) *Tff* **sin** (*Tff x*) N H	Sine
Tf **sincos** (*Tf x*, *Tf *cosval*)	Sine and cosine of *x*
Tf **sinh** (*Tf x*)	Hyperbolic sine
Tf **sinpi** (*Tf x*)	sin (π *x*)
Tf **sqrt** (*Tf x*) *Tff* **sqrt** (*Tff x*) N H	Square root
Tf **tan** (*Tf x*) *Tff* **tan** (*Tff x*) N H	Tangent
Tf **tanh** (*Tf x*)	Hyperbolic tangent
Tf **tanpi** (*Tf x*)	tan (π *x*)
Tf **tgamma** (*Tf x*)	Gamma function
Tf **trunc** (*Tf x*)	Round to integer toward zero

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 [xbit] in its name, this indicates that the type is *x* bits in size. Parameter types may also be their vec and marray counterparts.

Tint (geninteger in the spec) is type int[*n*], uint[*n*], unsigned int, char, char[*n*], signed char, scchar*n*, uchar*n*, unsigned short[*n*], unsigned short, ushort[*n*], long*n*, ulong*n*, long int, unsigned long int, long long int, longlong*n*, ulonglong*n* unsigned long long int.

uTint (ugeninteger) is type unsigned char, uchar*n*, unsigned short, ushort*n*, unsigned int, uint*n*, unsigned long int, ulong*n*, ulonglong*n*, unsigned long long int.

iTint (igeninteger) is type signed char, scchar*n*, short[*n*], int[*n*], long int, long*n*, long long int, longlong*n*.

sTint (sgeninteger) is type char, signed char, unsigned char, short, unsigned short, int, unsigned int, long int, unsigned long int, long long int, unsigned long long int.

uTint **abs** (*Tint x*)	| *x* |
uTint **abs_diff** (*Tint x*, *Tint y*)	| *x* – *y* | without modulo overflow
Tint **add_sat** (*Tint x*, *Tint y*)	*x* + *y* and saturates the result
Tint **hadd** (*Tint x*, *Tint y*)	(*x* + *y*) >> 1 without mod. overflow
Tint **rhadd** (*Tint x*, *Tint y*)	(*x* + *y* + 1) >> 1
Tint **clamp** (*Tint x*, *Tint min*, *Tint max*) *Tint* **clamp** (*Tint x*, *sTint min*, *sTint max*)	min(max(*x*, *min*), *max*)
Tint **clz** (*Tint x*)	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.
Tint **ctz** (*Tint x*)	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.
Tint **mad_hi** (*Tint a*, *Tint b*, *Tint c*)	mul_hi(*a*, *b*) + *c*
Tint **mad_sat** (*Tint a*, *Tint b*, *Tint c*)	*a* * *b* + *c* and saturates the result

(Continued on next page) ▶

Integer functions (cont.)

Tint **max** (*Tint x*, *Tint y*) *Tint* **max** (*Tint x*, *sTint y*)	*y* if *x* < *y*, otherwise it returns *x*
Tint **min** (*Tint x*, *Tint y*) *Tint* **min** (*Tint x*, *sTint y*)	*y* if *y* < *x*, otherwise it returns *x*
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 v*, *Tint i*)	result[*indx*] = *v*[*indx*] << *i*[*indx*]
Tint **sub_sat** (*Tint x*, *Tint y*)	*x* - *y* and saturates the result
uTint16bit **upsample** (*uTint8bit hi*, *uTint8bit lo*)	result[*i*]= ((ushort)*hi*[*i*]<< 8) | *lo*[*i*]
iTint16bit **upsample** (*iTint8bit hi*, *uTint8bit lo*)	result[*i*]=((short)*hi*[*i*]<< 8) | *lo*[*i*]
uTint32bit **upsample** (*uTint16bit hi*, *uTint16bit lo*)	result[*i*]=((uint)*hi*[*i*]<< 16) | *lo*[*i*]
iTint32bit **upsample** (*iTint16bit hi*, *uTint16bit lo*)	result[*i*]=((int)*hi*[*i*]<< 16) | *lo*[*i*]
uTint64bit **upsample** (*uTint32bit hi*, *uTint32bit lo*)	result[*i*]=((ulonglong)*hi*[*i*]<< 32) | *lo*[*i*]
iTint64bit **upsample** (*iTint32bit hi*, *uTint32bit lo*)	result[*i*]=((longlong)*hi*[*i*]<< 32) | *lo*[*i*]
Tint32bit **mad24** (*Tint32bit x*, *Tint32bit y*, *Tint32bit z*) *Tint32bit* **mad24** (*Tint32bit x*, *Tint32bit y*, *Tint32bit z*)	Multiply 24-bit integer values *x*, *y*, add 32-bit integer result to 32-bit integer *z*
Tint32bit **mul24** (*Tint32bit x*, *Tint32bit y*)	Multiply 24-bit integer values *x* and *y*

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 (genfloat in the spec) is type float[*n*], double[*n*], or half[*n*].
Tff (genfloatf) is type float[*n*].
Tfd (genfloatd) is type double[*n*].

Tf **clamp** (*Tf x*, *Tf minval*, *Tf maxval*); *Tff* **clamp** (*Tff x*, *float minval*,float *maxval*); *Tfd* **clamp** (*Tfd x*, double *minval*, doublen *maxval*);	Clamp *x* to range given by *minval*, *maxval*
Tf **degrees** (*Tf radians*);	*radians* to degrees
Tf **abs** (*Tf x*, *Tf y*); *Tff* **abs** (*Tff x*, float *y*); *Tfd* **abs** (*Tfd x*, double *y*);	Max of *x* and *y*
Tf **max** (*Tf x*, *Tf y*); *Tff* **max** (*Tff x*, float *y*); *Tfd* **max** (*Tfd x*, double *y*);	Max of *x* and *y*
Tf **min** (*Tf x*, *Tf y*); *Tff* **min** (*Tff x*, float *y*); *Tfd* **min** (*Tfd x*, double *y*);	Min of *x* and *y*
Tf **mix** (*Tf x*, *Tf y*, *Tf a*); *Tff* **mix** (*Tff x*, *Tff y*, float *a*); *Tfd* **mix** (*Tfd x*, *Tfd y*, double *a*) ;	Linear blend of *x* and *y*
Tf **radians** (*Tf degrees*);	*degrees* to radians
Tf **step** (*Tf edge*, *Tf x*); *Tff* **step** (float *edge*, *Tff x*); *Tfd* **step** (double *edge*, *Tfd x*);	0.0 if *x* < *edge*, else 1.0
Tf **smoothstep** (*Tf edge0*, *Tf edge1*, *Tf x*); *Tff* **smoothstep** (float *edge0*, float *edge1*, *Tff x*); *Tfd* **smoothstep** (double *edge0*, double *edge1*, *Tfd x*);	Step and interpolate
Tf **sign** (*Tf x*);	Sign of *x*

Preprocessor directives and macros [5.6]

SYCL_LANGUAGE_VERSION	Integer version, e.g.: 202002
__SYCL_DEVICE_ONLY__	==1 when in device compilation
__SYCL_SINGLE_SOURCE__	== 1 when producing both host and device binary
SYCL_EXTERNAL	if defined, indicates support for external kernel linkage

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 type in the functions below is shown with [xbit] in its name, this indicates that the type is *x* bits in size.

Tint (geninteger in the spec) is type int[*n*], uint[*n*], unsigned int, char, char[*n*], signed char, scharn, ucharn, unsigned short[*n*], unsigned short, ushort[*n*], longn, ulongn, long int, unsigned long int, long long int, longlongn, ulonglongn unsigned long long int.

iTint (igeninteger) is type signed char, scharn, short[*n*], int[*n*], long int, longn, long long int, longlongn.

uTint (ugeninteger) is type unsigned char, ucharn, unsigned short, ushortn, unsigned int, uintn, unsigned long int, ulongn, ulonglongn, unsigned long long int.

Ti (genint) is type int[*n*].

uTi (ugenint) is type unsigned int or uintn.

Tff (genfloatf) is type float[*n*].

Tfd (genfloatd) is type double[*n*].

T (gentype) is type float[*n*], double[*n*], or half[*n*], or any type listed for above for *Tint*.

int **any** (*iTint x*)	1 if MSB in component of *x* is set; else 0
int **all** (*iTint x*)	1 if MSB in all components of *x* are set; else 0
T **bitselect** (*T a*, *T b*, *T c*)	Each bit of result is corresponding bit of *a* if corresponding bit of *c* is 0
Tint **select** (*Tint a*, *Tint b*, *iTint c*) *Tint* **select** (*Tint a*, *Tint b*, *uTint c*) *Tff* **select** (*Tff a*, *Tff b*, *Ti c*) *Tff* **select** (*Tff a*, *Tff b*, *uTi c*) *Tfd* **select** (*Tfd a*, *Tfd b*, *iTint*64bit *c*) *Tfd* **select** (*Tfd a*, *Tfd b*, *uTint*64bit *c*)	For each component of a vector type, result[i] = if MSB of c[i] is set ? *b*[i] : *a*[i] For scalar type, result = *c* ? *b* : *a*
*iTint*32bit ***function*** (*Tff* x, *Tff* y) *iTint*64bit ***function*** (*Tfd* x, *Tfd* y) ***function***: isequal, isnotequal, isgreater, isgreaterequal, isless, islessequal, islessgreater, isordered, isunordered.	This format is used for many relational functions. Replace ***function*** with the function name.
*iTint*32bit ***function*** (*Tff* x) *iTint*64bit ***function*** (*Tfd* x) ***function***: isfinite, isinf, isnan, isnormal, signbit.	

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.

Tgf (gengeofloat in the spec) is type float, float2, float3, float4.
Tgd (gengeodouble) is type double, double2, double3, double4.

float4 **cross** (float4 *p0*, float4 *p1*) float3 **cross** (float3 *p0*, float3 *p1*) double4 **cross** (double4 *p0*, double4 *p1*) double3 **cross** (double3 *p0*, double3 *p1*)	Cross product
float **distance** (*Tgf p0*, *Tgf p1*) double **distance** (*Tgd p0*, *Tgd p1*)	Vector distance
float **dot** (*Tgf p0*, *Tgf p1*) double **dot** (*Tgd p0*, *Tgd p1*)	Dot product
float **length** (*Tgf p*) double **length** (*Tgd p*)	Vector length
Tgf **normalize** (*Tgf p*) *Tgd* **normalize** (*Tgd p*)	Normal vector length 1
float **fast_distance** (*Tgf p0*, *Tgf p1*)	Vector distance
float **fast_length** (*Tgf p*)	Vector length
Tgf **fast_normalize** (*Tgf p*)	Normal vector length 1

Kernel attributes [5.8.1]

Attributes are applied as shown in the following examples.

```
[=] (item<1> it) [[sycl::reqd_work_group_size(16)]] {
    //[kernel code]
}
```

```
void operator()(item<1> it) const [[sycl::reqd_work_
    group_size(16)]] {
    //[kernel code]
};
```

Attributes

reqd_work_group_size(dim0)
reqd_work_group_size(dim0, dim1)
reqd_work_group_size(dim0, dim1, dim2)
work_group_size_hint(dim0)
work_group_size_hint(dim0, dim1)
work_group_size_hint(dim0, dim1, dim2)
vec_type_hint(<type>)
reqd_sub_group_size(dim)

Device function attributes [5.8.2]

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

sycl::requires(has(aspect,...))

Notes

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 using the enum class backend:

```
enum class backend {
    implementation-defined
};
```

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

```
template <backend Backend>
class backend_traits {
  public:
    template <class T>
    using input_type = backend-specific;

    template <class T>
    using return_type = backend-specific;

    using errc = backend-specific;
};

template <backend Backend, typename SyclType>
using backend_input_t =
    typename backend_traits<Backend>::template
      input_type<SyclType>;

template <backend Backend, typename SyclType>
using backend_return_t =
    typename backend_traits<Backend>::template
      return_type<SyclType>;
```

get_native (4.5.1.2)

Returns a SYCL application interoperability native backend object associated with syclObject, which can be used for SYCL application interoperability.

```
template <backend Backend, class T>
  backend_return_t<Backend, T>
  get_native(const T &syclObject);
```

Backend functions (4.5.1.3)

```
template <backend Backend>
  platform make_platform(const backend_input_t <Backend,
    platform> &backendObject);

template<backend Backend>
  device make_device(const backend_input_t<Backend,
    device> &backendObject);

template <backend Backend>
  context make_context(const backend_input_t<Backend,
    context> &backendObject,
    const async_handler asyncHandler = {});

template <backend Backend>
  queue make_queue(const backend_input_t<Backend,
    queue> &backendObject,
    const context &targetContext,
    const async_handler asyncHandler = {});

template <backend Backend>
  event make_event(const backend_input_t<Backend, event>
    &backendObject,
    const context &targetContext);

template <backend Backend, typename T, int dimensions = 1,
    typename AllocatorT = buffer_allocator<
    std::remove_const_t<T>>>
  buffer make_buffer(const backend_input_t<Backend,
    buffer<T, dimensions, AllocatorT>> &backendObject,
    const context &targetContext,
    event availableEvent = {});

template <backend Backend, typename T, int dimensions = 1,
    typename AllocatorT = buffer_allocator<
    std::remove_const_t<T>>>
  buffer<T, dimensions, AllocatorT> make_buffer(const
    backend_input_t<Backend, buffer<T, dimensions,
    AllocatorT>> &backendObject,
    const context &targetContext);

template <backend Backend, int dimensions = 1,
    typename AllocatorT = sycl::image_allocator>
  sampled_image<dimensions, AllocatorT>
    make_sampled_image(
    const backend_input_t<Backend, sampled_image
    <dimensions, AllocatorT>> &backendObject,
    const context &targetContext,
    image_sampler imageSampler,
    event availableEvent = {});

template <backend Backend, int dimensions = 1,
    typename AllocatorT = sycl::image_allocator>
  sampled_image<dimensions, AllocatorT>
    make_sampled_image(
    const backend_input_t<Backend, sampled_image
    <dimensions, AllocatorT>> &backendObject,
    const context &targetContext,
    image_sampler imageSampler);

template <backend Backend, int dimensions = 1,
    typename AllocatorT = sycl::image_allocator>
  unsampled_image<dimensions, AllocatorT>
    make_unsampled_image(
    const backend_input_t<Backend, unsampled_image
    <dimensions, AllocatorT>> &backendObject,
    const context &targetContext,
    event availableEvent);

template <backend Backend, int dimensions = 1,
    typename AllocatorT = sycl::image_allocator>
  unsampled_image<dimensions, AllocatorT>
    make_unsampled_image(
    const backend_input_t<Backend, unsampled_image
    <dimensions, AllocatorT>> &backendObject,
    const context &targetContext);

template <backend Backend, bundle_state State>
  kernel_bundle<State> make_kernel_bundle(
    const backend_input_t<Backend,
    kernel_bundle<State>> &backendObject,
    const context &targetContext);

template <backend Backend>
  kernel make_kernel(const backend_input_t<Backend,
    kernel> &backendObject,
    const context &targetContext);
```

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:

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

Obtaining a kernel identifier [4.11.6]

Free functions:

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

template <typename KernelName>
  kernel_id get_kernel_id();
```

Obtaining a kernel bundle [4.11.7]

Free functions:

```
template<bundle_state State>
  kernel_bundle<State>
  get_kernel_bundle(const context &ctxt,
    const std::vector<device> &devs);

template<bundle_state State>
  kernel_bundle<State>
  get_kernel_bundle(const context &ctxt,
    const std::vector<device> &devs,
    const std::vector<kernel_id> &kernelIds);

template<bundle_state State, typename Selector>
  kernel_bundle<State>
  get_kernel_bundle(const context &ctxt,
    const std::vector<device> &devs, Selector selector);

template<bundle_state State>
  kernel_bundle<State>
  get_kernel_bundle(const context &ctxt);

template<bundle_state State>
  kernel_bundle<State>
  get_kernel_bundle(const context &ctxt,
    const std::vector<kernel_id> &kernelIds);

template<bundle_state State, typename Selector>
  kernel_bundle<State>
  get_kernel_bundle(const context &ctxt, Selector selector);

template<typename KernelName, bundle_state State>
  kernel_bundle<State>
  get_kernel_bundle(const context &ctxt);

template<typename KernelName, bundle_state State>
  kernel_bundle<State>
  get_kernel_bundle(const context &ctxt,
    const std::vector<device> &devs);
```

Querying if a bundle exists [4.11.8]

Free functions:

```
template<bundle_state State>
  bool has_kernel_bundle(const context &ctxt,
    const std::vector<device> &devs);

template<bundle_state State>
  bool has_kernel_bundle(const context &ctxt,
    const std::vector<device> &devs,
    const std::vector<kernel_id> &kernelIds);

template<bundle_state State>
  bool has_kernel_bundle(const context &ctxt);

template<bundle_state State>
  bool has_kernel_bundle(const context &ctxt,
    const std::vector<kernel_id> &kernelIds);

template<typename KernelName, bundle_state State>
  bool has_kernel_bundle(const context &ctxt);

template<typename KernelName, bundle_state State>
  bool has_kernel_bundle(const context &ctxt,
    const std::vector<device> &devs);
```

Querying if kernel is compatible with a device [4.11.9]

Free functions:

```
bool is_compatible(const std::vector<kernel_id> &kernelIds,
  const device &dev);

template<typename KernelName>
  bool is_compatible(const device &dev);
```

Joining kernel bundles [4.11.10]

```
template<bundle_state State> kernel_bundle<State>
  join(const std::vector<kernel_bundle<State>> &bundles);
```

Online compiling and linking [4.11.11]

Free functions:

```
kernel_bundle<bundle_state::object>
  compile(const kernel_bundle <
    bundle_state::input> &inputBundle,
    const std::vector<device> &devs,
    const property_list &propList = {});

kernel_bundle<bundle_state::executable>
  link(const std::vector<kernel_bundle <
    bundle_state::object>> &objectBundles,
    const std::vector<device> &devs,
    const property_list &propList = {});

kernel_bundle<bundle_state::executable>
  build(const kernel_bundle <
    bundle_state::input> &inputBundle,
    const std::vector<device> &devs,
    const property_list &propList = {});

kernel_bundle<bundle_state::object>
  compile(const kernel_bundle <
    bundle_state::input> &inputBundle,
    const property_list &propList = {});

kernel_bundle<bundle_state::executable>
  link(const kernel_bundle <
    bundle_state::object> &objectBundle,
    const std::vector<device> &devs,
    const property_list &propList = {});

kernel_bundle<bundle_state::executable>
  link(const std::vector<kernel_bundle <
    bundle_state::object>> &objectBundles,
    const property_list &propList = {});

kernel_bundle<bundle_state::executable>
  link(const kernel_bundle <
    bundle_state::object> &objectBundle,
    const property_list &propList = {});

kernel_bundle<bundle_state::executable>
  build(const kernel_bundle <
    bundle_state::input> &inputBundle,
    const property_list &propList = {});
```

The kernel bundle class [4.11.12]

Class declaration

```
template<bundle_state State>class kernel_bundle;
```

Members

```
bool empty() const noexcept;
backend get_backend() const noexcept;
context get_context() const noexcept;
```

(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;

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

Available when State == bundle_state::executable
kernel **get_kernel**(const kernel_id &*kernelId*) const;

bool **contains_specialization_constants**() const noexcept;

bool **native_specialization_constant**() const noexcept;

template<auto& S>
  bool **has_specialization_constant**() const noexcept;

Available when State == bundle_state::input
template<auto& S>
  void **set_specialization_constant**(typename
    std::remove_reference_t<decltype(S)>::type *value*);

template<auto& S>
  typename std::remove_reference_t<decltype(S)>::type
  get_specialization_constant() const;

device_image_iterator **begin**() const;

device_image_iterator **end**() const;

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;

template <typename param>
  typename param::return_type
  get_info(const device &*dev*) const;

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

Queries using get_info():

Descriptor	Return type
info::kernel_device_specific::global_work_size	range<3>
info::kernel_device_specific::work_group_size	size_t
info::kernel_device_specific::compile_work_group_size	range<3>
info::kernel_device_specific:: preferred_work_ group_size_multiple	size_t
info::kernel_device_specific::private_mem_size	size_t
info::kernel_device_specific::max_num_sub_groups	uint_32
info::kernel_device_specific::compile_num_sub_groups	uint_32
info::kernel_device_specific:: max_sub_group_size	uint_32
info::kernel_device_specific:: compile_sub_group_size	uint_32

The device image class [4.11.14]

class declaration

template<bundle_state State>class device_image;

Members

bool **has_kernel**(const kernel_id &*kernelId*) const noexcept;

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

USM examples

Example with USM Shared Allocations

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
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 << "] = " << data[i] << std::endl;

  return 0;
}
```

Example with USM Device Allocations

```
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names
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 <<"data["<< i << "] = " << data[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.

```
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.

```
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.

```
myQueue.submit([&](handler & cgh) {
   accessor acc { myBuffer, cgh, write_only };

   cgh.parallel_for(range<1>(numWorkItems),
                    [=] (item<1> item) {
      // kernel argument type is item
      size_t index = item.get_linear_id();
      acc[index] = index;
   });
});
```

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.

```
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.

```
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.

```
myQueue.submit([&](handler & cgh) {
  // Issue 8 work-groups of 8 work-items each
  cgh.parallel_for_work_group(range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup) {

     // [workgroup code]
     int myLocal; // this variable is shared between workitems
     // This variable will be instantiated for each work-item separately
     private_memory<int> myPrivate(myGroup);

     // 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
     myGroup.parallel_for_work_item([&](h_item<3> myItem) {
        // [work-item code]
        output[myItem.get_global_id()] = myPrivate(myItem);
     });
     //[workgroup code]
  });
});
```