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

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

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

Common interfaces

Common reference semantics [4.5.2]

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

friend bool operator== (const & T & lhs, const & T & rhs);
friend bool operator== (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== (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_unmanaged_image_accessor, context, local_accessor, queue, [un]sampled_image, [un]sampled_image_accessor, stream, and usm_allocator.

template<typename propertyT>
struct is-property {
  static int value;
};

template<typename propertyT>
inline constexpr bool is_property_v = is_property<
  propertyT>::value;

template<typename propertyT>
class property_list {
public:
  property_list();
  template<typename... propertyTN>
  property_list(property_list & list);
private:
  template<typename propertyT>
  class T {
  public:
    T();
    T(const T & rhs);
    T & operator=(const T & rhs);
  };?
};

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(property_list & list);
};

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

<table>
<thead>
<tr>
<th>default_selector_v</th>
<th>Selects device by system heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu_selector_v</td>
<td>Select a device according to device type info::device_type::gpu</td>
</tr>
<tr>
<td>cpu_selector_v</td>
<td>Select a device according to device type info::device_type::cpu</td>
</tr>
<tr>
<td>accelerator_selector_v</td>
<td>Select an accelerator device.</td>
</tr>
</tbody>
</table>

Anatomy of a SYCL application [3.2]

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

```cpp
#include <sycl.hpp>
using namespace sycl;

// 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, range<1024>> data{0};
  buffer<int, 1, range<1024>> resultBuf{0};

  // Create a command group to issue commands to the queue.
  myQueue.submit([&](handler & h)
  {
    // Request access to the buffer without initialization
    accessor<buffer<int, 1, range<1024>>, write_access, no_init> dataAccess{h, data, 0, 1024};
    accessor<buffer<int, 1, range<1024>>, write_access, no_init> resultAccess{h, resultBuf, 0, 1024};
    accessor<buffer<int, 1, range<1024>>, read_access> tempAccess{h, data, 0, 1024};

    // Divide the problem into 1024 work-items
    // and execute each work-item
    parallel_for(range<1024>, h, dataAccess);
    parallel_for(range<1024>, h, resultAccess);

    // End of the kernel function
  });

  // End of the kernel 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] << std::endl;
}
```

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

Platform class [4.6.2]

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

```cpp
platform();
```
Device class (cont.)

```cpp
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 device_type = info::device_type::all);
```

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

```cpp
descriptor Return type
device_type info::device_type
to id<2>
```

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

```cpp
explicit context(std::vector<info::fp_config>&& propList) const;
explicit context(std::vector<info::fp_config> const& propList) const;
explicit context(devices const& deviceList, async_handler& asyncHandler, const std::vector<info::fp_config> & propList = {});
```

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

```cpp
descriptor Return type
platform std::vector<info::platform>
devices std::vector<info::device>
atomic_memory_order_capabilities std::vector<info::memory_order>
atomic_fence_order_capabilities std::vector<info::memory_order>
atomic_memory_scope_capabilities std::vector<info::memory_scope>
atomic_fence_scope_capabilities std::vector<info::memory_scope>
```

Device aspects [4.6.4.3]
Device aspects are defined in enum class aspect. The core enumerators are shown below. Specific backends may define additional aspects.

```cpp
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
```

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

```cpp
explicit queue(const device& device, const context& context = context());
```

Queue queries using get_info():

```cpp
descriptor Return type
info::queue::context context
info::queue::device device
```

Convenience shortcuts

```cpp
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, 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(range<Dims> numWorkitems, const std::vector<event>& depEvents, 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, event depEvent, Rest&... rest);
```

(Continued on next page)
USM Functions

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

event memcpyy(void* dest, const void* src, size_t numBytes, event depEvent);

event memcpyy(void* dest, const void* src, size_t numBytes, std::vector<event> &depEvents);

template<typename T>
event copy(const T* src, T* dest, size_t size, event depEvent);

template<typename T>
event copy(const T* src, T* dest, size_t size, std::vector<event> &depEvents);

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

Explicit copy functions

template<typename T_src, int dim_src, access_mode src_tgt, target tgt_src, size_t numBytes, accessor<T_src, dim_src, src_tgt, tgt_src, isPlaceholder> src, std::shared_ptr<T_dest> dest>

template<typename T_src, typename T_dest, int dim_dest, target tgt_dest, accessor<T_src, dim_dest, src_tgt, tgt_src, isPlaceholder> src, std::shared_ptr<T_dest> dest>

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, access_mode mode = access::placeholder isPlaceholder>
buffer<T, dimensions> const & first;
buffer<T, dimensions> const & last;

Member functions

buffer<T, dimensions> & bufferRange;

Buffer property class constructors:

property::buffer::use_host_ptr<const T*, use_host_ptr>

Buffer class [4.7.2]

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

Event class [4.6.6]

Event queries using get_info()

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>info::event::command_execution_status</td>
<td>info::event::command_status</td>
</tr>
</tbody>
</table>

Queries using get_profiling_info()

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>info::profiling::command_submit</td>
<td>int64_t</td>
</tr>
<tr>
<td>info::profiling::command_start</td>
<td>int64_t</td>
</tr>
<tr>
<td>info::profiling::command_end</td>
<td>int64_t</td>
</tr>
</tbody>
</table>

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(b);

The default allocators are buffer_allocator for buffers and image_allocator for images.
Images, unsampled and sampled [4.7.3]

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


class accessor;

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

Class declaration
template<typename dataT, int dimensions, access_mode accessMode>

Available when dimensions = 0

Available when dimensions > 0

Constructors and members
sampled_image(const void *hostPointer,
image_format format, image_sampler sampler,
custom_ranges dimensions &rangeRef,
custom_property_list &propList = {});
sampled_image(const void *hostPointer,
image_format format, image_sampler sampler,
custom_ranges dimensions &rangeRef,
custom_property_list &propList = {});

Available when dimensions = 1

Available when dimensions > 1

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)
- Accessor for host code outside of a command (4.7.6.10, class accessor)
- Local accessor from within kernel functions (4.7.6.11, class local_accessor)

Available when dimensions = 0

Available when dimensions > 0

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 memory
- target::host_task to access a buffer from a host task

Class declaration
template<typename dataT, int dimensions, access_mode accessMode>

Constructors and members
accessor();

Available when dimensions = 0

Available when dimensions > 0

Access targets [4.7.6.9]

target::device buffer access from kernel function via device memory

target::host_task buffer access from a host task

enum class 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

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

Available when dimensions > 0

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

Available when dimensions == 0

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

Available when dimensions > 1

```cpp
__unspecified__: handler &commandGroupHandlerRef
```

### Common interface functions [Table 79]

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

- 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;

### Property class constructor [4.7.3.3]

```cpp
_property::no_init::no_init();
```

### Common interface functions [Table 79]

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

- 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;

### Property class constructor [4.7.3.3]

```cpp
_property::no_init::no_init();
```

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

#### Class declaration

```cpp
template<typename dataT, int dimensions, AllocatorT>
class host_accessor;
```

#### Constructors and members

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

```cpp
host_accessor();
```

#### Property class constructor [4.7.3.3]

```cpp
_property::no_init::no_init();
```

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

#### Class declaration

```cpp
template<typename dataT, int dimensions>
class local_accessor;
```

#### Constructors and members

```cpp
local_accessor();
```

#### Property class constructor [4.7.3.3]

```cpp
_property::no_init::no_init();
```
Unsampled image accessors [4.7.6.13]
There are two kinds of unsampled image accessors:
- class host_unsampled_image_accessor: From host code outside of a host task
- class unsampled_image_accessor: From within a kernel function or from within a host task

Unsampled image accessor
Class declaration
template <typename dataT, int dimensions, access::mode Mode, access::address_space::local_space
image_target accessTarget = image_target::device >
class host_unsampled_image_accessor;

Constructor
template <typename AllocatorT>
host_unsampled_image_accessor
unsampled_image_accessor(dimensions, AllocatorT)> &imageRef,
handler &CommandGroupHandlerRef, Tag1 Tag, const property_list &propList = ());

Available at both unsampled image accessor types size_1 size() const noexcept;
Available when (accessMode == access_mode::read) template <typename coordT>
data1 read(const coordT &coords) const;

Available when (accessMode == access_mode::write) template <typename coordT>
void write(const coordT &coords, const data1 &color) const;

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

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

Sampled image accessor
Class declaration
template <typename dataT, int dimensions, image_target accessTarget = image_target::device>
class sampled_image_accessor;

Constructor
template <typename AllocatorT>
host_sampled_image_accessor
sampled_image_accessor(dimensions, AllocatorT)> &imageRef,
const property_list &propList = ());

Available to both sampled image accessor types size_1 size() const noexcept;

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::host_space
image_target accessTarget = image_target::device >
class multi_ptr;

Members: Constructors
multi_ptr() (multi_ptr<VoidType, access::address_space::global_space, isPlaceholder>
= {});

Available if Space == address_space::generic_space &
ASP != access::address_space::constant_space
template <typename ElementType, int dimensions, access::mode Mode,
access::address_space::local_space, isPlaceholder>
multi_ptr operator()(const coordT &coord);

Available if Space == address_space::generic_space
template <typename ElementType, int dimensions, Mode, access::address_space::local_space
imageTarget accessTarget = image_target::device >
class multi_ptr;

Members: Constructors
multi_ptr() (multi_ptr<VoidType, access::address_space::global_space, isPlaceholder>
= {});

Available if Space == address_space::generic_space &
ASP != access::address_space::constant_space
template <typename ElementType, int dimensions, access::mode Mode,
access::address_space::local_space, isPlaceholder>
multi_ptr operator()(const coordT &coord);

Available if Space == address_space::generic_space

Class multi_ptr specialized for void and const void [4.7.7.1]
Class declaration
template <typename ElementType = void, access::address_space::host_space
image_target accessTarget = image_target::device >
class multi_ptr;

Members: Constructors
multi_ptr() (multi_ptr<VoidType, access::address_space::global_space, isPlaceholder>
= {});

Available if Space == address_space::generic_space &
ASP != access::address_space::constant_space
template <typename ElementType, int dimensions, access::mode Mode,
access::address_space::local_space, isPlaceholder>
multi_ptr operator()(const coordT &coord);

Available if Space == address_space::generic_space

Assignment operators
multi_ptr &operator=(multi_ptr &);
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 &ctx, 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> &a);

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_allocator<U, size_t AlignmentU>;
friend bool operator==(const usm_allocator<T, AllocKind, Alignment> &a);
template <class U> usm_allocator(usm_allocator<U, AllocKind, Alignment> &a);
friend bool operator==(const usm_allocator<T, AllocKind, Alignment> &a);

mallocre-style API [4.8.3]

Device allocation functions [4.8.3.2]

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

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

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

Shared allocation functions [4.8.3.4]

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

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

Sampler class enums [4.8.5]

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
filtered repeat
clamp_to_edge
clamp

coordinate
normalized
unnormalized

©2021 Khronos Group - Rev. 0721
www.khronos.org/sycl
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, bool with_offset = true>
    class range;
```

Constructors and members

```
size_t
    range<dimensions> get_id() const;
size_t
    range<dimensions> get_range() const;
```
Reduction variables [4.9.2]

Reductions are supported for all SYCL copyable types.

```cpp
template<typename BufferT, typename BinaryOperation>
    unspecified_reduction(BufferT* vars, handler& cgh, BinaryOperation combiner, const property_list& propsList = {});
template<typename T, typename BinaryOperation>
    unspecified_reduction(T* var, BinaryOperation combiner, const property_list& propsList = {});
```

Reduction class functions [4.9.2.3]

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

```cpp
template<typename T>
    void operator[](reducer<T, plus<T>, 0>&, BufferT* vars, const T& partial);
```

Member functions

```cpp
Member function for using a kernel bundle
void use_kernel_bundle(const kernel_bundle< bundle_state::executable>& bundle);
```

USM functions

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

Explicit memory operation APIs

```cpp
template<typename T_src, int dim, access_mode mode, target tgt, access::target tgt_src, access::target tgt_dest, isPlaceholder src, isPlaceholder dest, isPlaceholder& isPlaceholderDest>
    void copy(const T src, const T dest, size_t nCount, const target& tgt, const target& tgt_dest, isPlaceholder& isPlaceholderDest);
```

Kernel dispatch API

```cpp
template<typename KernelName, int dim, access_mode mode, target tgt, access::target tgt_src, access::target tgt_dest, isPlaceholder src, isPlaceholder dest, isPlaceholder& isPlaceholderDest>
    void copy(const T src, const T dest, size_t nCount, const target& tgt, const target& tgt_dest, isPlaceholder& isPlaceholderDest);
```

Command group handler class [4.9.4]

This class provides the ability to manage a group of kernels, as well as the ability to share data between kernels.

```cpp
class CommandGroupHandler {
  // Properties
  buffer<nd_range<1>, T> cgh;

  // Member functions
  void parallel_for(workgroup<1>{1024}, int idx, auto& sum, auto& max);

  // Buffers and maxBuf contain the reduction results once the kernel completes
  assert(maxBuf.get_host_access()[0] == 1023 && sumBuf.get_host_access()[0] == 523776);
};
```
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>& g);  
    &operator[](const h_item<Dimensions>& h);  
};

Classes exception & exception_list [4.13.2]  
Class exception is derived from std::exception.

Class_kernel_handler [4.10.5]  
Member functions

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>& g);  
    &operator[](const h_item<Dimensions>& h);  
};

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:

```cpp  
class RandomFiller {  
  public:  
    RandomFiller(accessor<int> ptr)  
      : ptr_{ ptr }  
    {  
      RandomFiller(int, ptr)  
    }  
    RandomFiller(int, ptr)  
    {  
      RandomFiller(accessor<int>& ptr)  
    }  
  private:  
    int ptr_;  
    RandomFiller(int, ptr);  
    RandomFiller(int, ptr);  
  };  
```

Defining kernels as lambda functions [4.12.2]  
Kernels may be defined as lambda functions. The name of a lambda function in SYCL may optionally be specified by passing it as a template parameter to the invoking member function. For example:

```cpp  
// Explicit kernel names can be optionally forward declared  
// at namespace scope  
class MyKernel  
{  
  myQueue.submit([](handler& h) {  
    // Explicitly name kernel with previously forward declared  
    // declared type  
    MyKernel k;  
    // [kernel code]  
    });  
};  
```

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:

```cpp  
void wait() noexcept;  
```

Class_atomic_ref [4.15.3]  
Class declaration  

Synchronization and atomics [4.15]  

Enums  

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class_memory_order</td>
<td>relaxed</td>
</tr>
<tr>
<td>class_memory_scope</td>
<td>work item</td>
</tr>
<tr>
<td>atomic_fence [4.15.1]</td>
<td>Free function: void atomic_fence(...)</td>
</tr>
</tbody>
</table>

(Continued on next page)
Synchronization and atomic operations

### Class declaration

**template** <typename Group, typename T>

```cpp
Group::id_type
```

**members**

- **operator()**
  - `first` (Group &)
  - `last` (Group &)
  - `pred` (Predicate)

- **operator<<**
  - `binary_op` (const T &, const T &)

### Class functions

- **get_work_item_buffer_size**: `size_t` (const group &)
- **get_local_id**: `Group::id_type` (const group &)

### Group functions

- **select_from_group**: `void` (Group &)
- **parallel_for**: `void` (Group &)
- **dot_product**: `T` (Group &)

### Scalar and algorithm libraries

- **operator+=**: `T` (Group &)
- **operator-**: `T` (Group &)
- **operator&**: `T` (Group &)
- **operator|**: `T` (Group &)
- **operator^**: `T` (Group &)

### Function objects

- **maximum**, **minimum**, **logical_or**, **bit_xor**, **bit_or**, **bit_and**

### References

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

### Device event

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()** 

### Stream

- **stream**: `stream` (typename T)
- **constructors**: `stream` (typename T)
- **members**: `operator<<` (const stream &)
- **functions**: `template<typename T>` (typename T)
- **non-member functions**: `template<typename T>` (typename T)

### Function objects

- **maximum**, **minimum**, **logical_or**, **bit_xor**, **bit_or**, **bit_and**

### Scalar data types

- **data types**: `int`, `long`, `long long`, `float`, `double`, `half`

### Enum

- **stream_manipulator**: `stream` (typename T)
- **constructors**: `stream` (typename T)
- **members**: `operator<<` (const stream &)
- **functions**: `template<typename T>` (typename T)
- **non-member functions**: `template<typename T>` (typename T)
### Math functions [4.1.7.5]

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]. `Tf` (genfloat) is type `float[n]`. `Tf` (geninteger) is type `signed char, schar, int[n]`. `Tf` (geninteger) is type `signed int or uint, uint[n]`. `Tf` (geninteger) is type `signed long int, longlong, long int`.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acos</code></td>
<td>Arc cosine</td>
</tr>
<tr>
<td><code>acosh</code></td>
<td>Inverse hyperbolic cosine</td>
</tr>
<tr>
<td><code>acosf</code></td>
<td>Arcos <code>x</code> ( / \pi )</td>
</tr>
<tr>
<td><code>asin</code></td>
<td>Arc sine</td>
</tr>
<tr>
<td><code>asinh</code></td>
<td>Inverse hyperbolic sine</td>
</tr>
<tr>
<td><code>asinf</code></td>
<td>Arcsin <code>x</code> ( / \pi )</td>
</tr>
<tr>
<td><code>atan</code></td>
<td>Arc tangent</td>
</tr>
<tr>
<td><code>atan2</code></td>
<td>Arc tangent of <code>y / x</code></td>
</tr>
<tr>
<td><code>atanh</code></td>
<td>Hyperbolic arc tangent</td>
</tr>
<tr>
<td><code>atand</code></td>
<td>Arc tangent of <code>x</code></td>
</tr>
<tr>
<td><code>atan2d</code></td>
<td>Arc tangent of <code>y / x</code></td>
</tr>
<tr>
<td><code>cbrt</code></td>
<td>Cube root</td>
</tr>
<tr>
<td><code>ceil</code></td>
<td>Round to integer toward + infinity</td>
</tr>
<tr>
<td><code>copysign</code></td>
<td><code>x</code> with sign changed to sign of <code>y</code></td>
</tr>
<tr>
<td><code>cos</code></td>
<td>Cosine</td>
</tr>
<tr>
<td><code>cosh</code></td>
<td>Hyperbolic cosine</td>
</tr>
<tr>
<td><code>cosf</code></td>
<td>Cosx <code>x</code> ( / \pi )</td>
</tr>
<tr>
<td><code>dive</code></td>
<td>Divide <code>x</code> by <code>y</code></td>
</tr>
<tr>
<td><code>erf</code></td>
<td>Complementary error function</td>
</tr>
<tr>
<td><code>erff</code></td>
<td>Calculates error function</td>
</tr>
</tbody>
</table>

### Integer functions [4.1.7.6]

Integer functions are available in the namespace sycl. In all cases below, `n` may be `2, 3, 4, 8, or 16`. If a `T` in the functions below is shown with `|bit|` 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], int[n]`. `Tint` (geninteger) is type `signed char, schar, char, short[n], unsigned short[n]`. `Tint` (geninteger) is type `signed int, uint[n], int[n]`. `Tint` (geninteger) is type `signed long int, longlong, long int`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Tint abs</code></td>
<td><code>Tint x</code></td>
</tr>
<tr>
<td><code>Tint abs_diff</code></td>
<td><code>Tint x</code>, <code>Tint y</code></td>
</tr>
<tr>
<td><code>Tint add_sat</code></td>
<td><code>Tint x</code>, <code>Tint y</code></td>
</tr>
<tr>
<td><code>Tint hadd</code></td>
<td><code>Tint x</code>, <code>Tint y</code></td>
</tr>
<tr>
<td><code>Tint hadd_sat</code></td>
<td><code>Tint x</code>, <code>Tint y</code></td>
</tr>
<tr>
<td><code>Tint clamp</code></td>
<td><code>Tint x</code>, <code>Tint min</code>, <code>Tint max</code></td>
</tr>
<tr>
<td><code>Tint clamp_sat</code></td>
<td><code>Tint x</code>, <code>Tint min</code>, <code>Tint max</code></td>
</tr>
</tbody>
</table>

(Continued on next page)
Integer functions (cont.)

- \( \text{Tint max} (\text{Tint } x, \text{Tint } y) \) - if \( x < y \), otherwise it returns \( x \)
- \( \text{Tint min} (\text{Tint } x, \text{Tint } y) \) - if \( x < y \), otherwise it returns \( y \)
- \( \text{Tint sub_sat} (\text{Tint } x, \text{Tint } y) \) - \( x - y \) and saturates the result

Common functions [4.17.7]

Common functions are available in the namespace sycl on host and device. On the host the vector types use the vec class and on
an OpenCL device use the corresponding OpenCL vector types. In all cases below, \( n \) may be 2, 3, 4, 8, or 16. The built-in functions can take as input float or optionally double and their vec and
marray counterparts. \( \text{Tf} \) (genfloat in the spec) is type float[n], double[n], or half[n]. \( \text{Tfd} \) (genfloatd) is type double[n].

- \( \text{fold} (\text{Tf } x, \text{double} x) \) - \( \text{Tf} \) (genfloat) is type double[n].

- \( \text{fld} \) - \( x \) to range given by \( \text{minval} \) and \( \text{maxval} \)
- \( \text{fold} (\text{Tf } x, \text{double} x) \) - \( \text{Tf} \) (genfloat) is type double[n].

Relational built-in functions [4.17.9]

Relational functions are available in the namespace sycl on host and device. In all cases below, \( n \) may be 2, 3, 4, 8, or 16. If a
function returns a boolean vector, it is true for any \( (X_i)_n \) where \( X_i \) is \( \text{true} \)
for at least one \( n \). If a relational function returns a boolean, \( \text{true} \) if \( X_i \) is \( \text{true} \) for all \( n \).

- \( \text{Tf} \) (genfloat in the spec) is type float[n], double[n], or half[n]. \( \text{Tff} \) (genfloatd) is type double[n].
- \( \text{fmax} (\text{Tf } x, \text{Tf } y) \) - \( \text{Tf} \) (genfloat) is type float[n].
- \( \text{fmin} (\text{Tf } x, \text{Tf } y) \) - \( \text{Tf} \) (genfloat) is type double[n].

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.

- \( \text{fgeq}(\text{gengeq in the spec}) \) is type float, float2, float3, float4,
double, double2, double3, double4.

 Kernel attributes [5.8.1]

Attributes are applied as shown in the following examples.

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

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

Attributes

- \( \text{reqd_work_group_size(dim0)} \)
- \( \text{reqd_work_group_size(dim0, dim1)} \)
- \( \text{reqd_work_group_size(dim0, dim1, dim2)} \)
- \( \text{work_group_size_hint(dim0)} \)
- \( \text{work_group_size_dim0} \)
- \( \text{work_group_size_dim1} \)
- \( \text{work_group_size_dim0, dim1} \)
- \( \text{vec_type Hint} \)
- \( \text{reqd_sub_group_size(dim)} \)

Device function attributes [5.8.2]

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

```
sycl::requires(has_aspect,...)
```
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 name 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, typename SyclType>
{ boolean hasKernelBundle();
  template <typename KernelName> bool hasKernelBundle<KernelName>();
};

Obtaining a kernel identifier [4.11.6]
Some of the functions related to kernel bundles take an input parameter of type kernel_id. It is a class with members:
const char *get_name() const noexcept;

Obtaining a kernel bundle [4.11.7]
Free functions:
template <backend Backend, typename KernelName> kernel_bundle<KernelName> get_kernel_bundle(const std::vector<device>& devs, const kernel_id& kernelsId);

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 members:
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 <backend Backend, typename KernelName> kernel_bundle<KernelName> get_kernel_bundle(const std::vector<device>& devs, const kernel_id& kernelsId);

Online compiling and linking [4.11.11]
Free functions:
kernel_bundle<bundle_state::object> compile(const kernel_bundle & bundleObject, const std::vector<device>& devs, const kernel_id& kernelsId);

The kernel bundle class [4.11.12]
Class declaration
template <backend Backend, typename AllocatorT> kernel_bundle<Backend, AllocatorT>;
The kernel class [4.11.13]

- **get_backend()** const noexcept;
- **get_context()** const;
- **get_kernel_bundle()** const;
- **get_info()** const;
- **get_backend_info()** const;
- **get_info()** (const device & dev) const;
- **get_info()** (const device & dev) const;
- **get_info()** (const device & dev) const;
- **get_info()** (const device & dev) const;

Queries using **get_info()**:

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

The device image class [4.11.14]

**class declaration**

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

**Example with USM Shared Allocations**

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

int main() {
    queue myQueue;

    // Allocate shared memory bound to the device and context associated to the queue
    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;
    });
    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**

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

int main() {
    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;
    });
    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.

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

Examples: parallel_for invoke [4.9.4.2.2]

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

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

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

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

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

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

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

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

Parallel for hierarchical invoke [4.9.4.2.3]
In the following example we issue 8 work-groups but let the runtime choose their size, by not passing a work-group size to the parallel_for work_group call. The parallel_for work_item loops may also vary in size, with their execution ranges unrelated to the dimensions of the work-group, and the compiler generating an appropriate iteration space to fill the gap. In this case, the h_item provides access to local ids and ranges that reflect both kernel and parallel_for work_item invocation ranges.

```cpp
myQueue.submit([&](handler & cgh) {
    // issue 8 work-groups of 8 work-items each
cgh.parallel_for_work_group(range<3>(2, 2, 2), 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]
});
```

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

syclcon.org

© 2021 Khronos Group. All rights reserved. SYCL is a trademark of the Khronos Group. The Khronos Group is an industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, and more on a wide variety of platforms and devices. See www.khronos.org to learn more about the Khronos Group. See www.khronos.org/sycl to learn more about SYCL.