Runtime: Class device_selector [3.3.1]

device_selector();
device_selector(const device_selector &*selector*);
device **select_device**() const;
virtual int **operator()**(const device &*device*) const;

SYCL device selectors:

Device selectors	Description
default_selector	Devices selected by heuristics of the system.
gpu_selector	Select devices according to device type info::device::device_type::gpu
cpu_selector	Select devices according to device type info::device::device_type::cpu
host_selector	Selects the SYCL host CPU device that does not require an OpenCL runtime.

Runtime: Class platform [3.3.2]

platform();
explicit **platform**(cl_platform_id *platformId*);
explicit **platform**(device_selector &*devSelector*);
platform(const platform &*rhs*);
platform &**operator=**(const platform &*rhs*);
cl_platform_id **get**() const;
static vector_class<platform> **get_platforms**() const;
vector_class<device> **get_devices**(
info::device_type = info::device_type::all) const;
template <info::platform param>
typename info::param_traits<
info::platform, param>::type **get_info**() const;
bool **has_extension**(string_class *extension*) const;
bool **is_host**() const;

Platform information descriptors:

Platform descriptors	Return type
info::platform::profile	string_class
info::platform::version	string_class
info::platform::name	string_class
info::platform::vendor	string_class
info::platform::extensions	string_class

Runtime: Class context [3.3.3]

context();
explicit **context**(async_handler *asyncHandler* = nullptr);
context(const device_selector &*deviceSelector*,
info::gl_context_interop *interopFlag*,
async_handler *asyncHandler* = nullptr);
context(const device &*dev*,
info::gl_context_interop *interopFlag*,
async_handler *asyncHandler* = nullptr);
context(const platform &*plt*,
info::gl_context_interop *interopFlag*,
async_handler *asyncHandler* = nullptr);
context(vector_class<device> *deviceList*,
info::gl_context_interop *interopFlag*,
async_handler *asyncHandler* = nullptr);
context(cl_context *clContext*,
async_handler *asyncHandler* = nullptr);
context(const context &*rhs*);
cl_context **get**() const;
bool **is_host**() const;
template <info::context param> typename param_traits<
info::context, param>::type **get_info**() const;
platform **get_platform**();
vector_class<device> **get_devices**() const;

Optional context class members

The following members are also available in class context when enabled by extension cl_khr_gl_sharing. See OpenGL Interop [4.4.1] for more information.

device **get_gl_current_device**();
vector_class<device> **get_gl_context_devices**();

Context queries using get_info():

Descriptor	Return type
info::context::reference_count	cl_uint
info::context::num_devices	cl_uint
info::context::devices	vector_class<cl_device_id>
info::context::gl_interop	info::gl_context_interop

SYCL is a C++ programming model for OpenCL which builds on the underlying concepts, portability, and efficiency of OpenCL while adding much of the ease of use and flexibility of C++.

[n.n] refers to sections in the SYCL 1.2 specification available at khronos.org/registry/sycl

SYCL Example [2.6]

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

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

int main() {
    using namespace cl::sycl;
    int data[1024]; // Data to be worked on

    // Include all the SYCL work in a {} block to ensure all
    // SYCL tasks are completed before exiting the block.
    {
        // Create a queue to enqueue work to
        queue myQueue;

        // Wrap the 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
            auto writeResult = resultBuf.get_access<access::write>(cgh);

            // Enqueue a parallel_for task.
            cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx)
            {
                writeResult[idx[0]] = static_cast<int>(idx[0]);
            }); // 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 <CL/sycl.hpp> header file to provide all of the SYCL features.

Namespace
All SYCL names are defined in the cl::sycl namespace.

Queue
See queue class functions [3.3.5] on page 2 of this reference guide.

Buffer
See buffer class functions [3.4.2] on page 2 of this reference guide.

Accessor
See accessor class function [3.4.6] on page 3 of this reference guide.

Handler
See handler class functions [3.4.3.4] on page 5 of this reference guide.

Scopes
The ***kernel scope*** specifies a single kernel function that will be, or has been, compiled by a device compiler and executed on a device. See Invoking Kernels [3.5.4]

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.

Runtime: Class device [3.3.4]

device();
explicit **device**(device_selector &*deviceSelector*);
explicit **device**(cl_device_id *deviceId*);
device(const device &*rhs*);
device &**operator=**(const device &*rhs*);
cl_device_id **get**() const;
platform **get_platform**() const;
bool **is_host**() const;
bool **is_cpu**() const;
bool **is_gpu**() const;
bool **is_accelerator**() const;
template <info::device param> typename info::param_traits<
info::device, param>::type **get_info**() const;
bool **has_extension**(string_class *extension*) const;
vector_class<device> **create_sub_devices**(
info::device_partition_type *partitionType*,
info::device_partition_property *partitionProperty*,
info::device_affinity_domain *affinityDomain*);
static vector_class<device> **get_devices**(
info::device_type *deviceType* =
info::device_type::all);

Device queries using get_info():

Descriptor in info::device	Return type
device_type	info::device_type
vendor_id	cl_uint
max_compute_units	cl_uint
max_work_item_dimensions	cl_uint
max_work_item_sizes	id<3>
max_work_group_size	size_t
preferred_vector_width_char	cl_uint
preferred_vector_width_short	
preferred_vector_width_int	
preferred_vector_width_long_long	
preferred_vector_width_float	
preferred_vector_width_double	
preferred_vector_width_half	
native_vector_width_char	cl_uint
native_vector_width_short	
native_vector_width_int	
native_vector_width_long_long	
native_vector_width_float	
native_vector_width_double	
native_vector_width_half	
max_clock_frequency	cl_uint

(Continued on next page) ▶

◀ Runtime: Class device (continued)

Descriptor	Return type
address_bits	cl_uint
max_mem_alloc_size	cl_long
image_support	cl_bool
max_read_image_args	cl_uint
max_write_image_args	cl_uint
image2d_max_width	size_t
image_2d_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
image_max_array_size	size_t
max_samplers	cl_uint
max_parameter_size	size_t
mem_base_addr_align	cl_uint
single_fp_config	info::device_fp_config
double_fp_config	info::device_fp_config
global_mem_cache_type	info::device_mem_cache_type
global_mem_cacheline_size	cl_uint
global_mem_cache_size	cl_ulong
global_mem_size	cl_ulong
max_constant_buffer_size	cl_ulong
max_constant_args	cl_uint
local_mem_type	info::local_mem_type
local_mem_size	cl_ulong
error_correction_support	cl_bool

Descriptor	Return type
host_unified_memory	cl_bool
profiling_timer_resolution	size_t
is_endian_little	cl_bool
is_available	cl_bool
is_compiler_available	cl_bool
is_linker_available	cl_bool
execution_capabilities	info::device_exec_capabilities
queue_properties	vector_class<info ::device_queue_properties>
built_in_kernels	string_class
platform	cl_platform_id
name	string_class
vendor	string_class
version	string_class
profile	string_class
version	string_class
opencl_c_version	string_class
extensions	string_class
printf_buffer_size	size_t
preferred_interop_user_sync	cl_bool
parent_device	cl_device_id
partition_max_sub_devices	cl_uint
partition_properties	vector_class <info:: device_partition_property>
partition_affinity_domain	info::device_affinity_domain
partition_type	vector<info:: device_partition_property>
reference_count	cl_uint

Runtime: Class queue [3.3.5]

explicit **queue**(async_handler *asyncHandler* = nullptr);

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

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

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

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

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

queue(cl_command_queue *clQueue*, async_handler *asyncHandler* = nullptr);

queue(queue &*syclQueue*);

cl_command_queue **get**();

context **get_context**();

device **get_device**();

bool **is_host**();

void **wait**();

void **wait_and_throw**();

void **throw_asynchronous**();

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

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

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

Queue queries using get_info():

Descriptor	Return type
info::queue::context	cl_context
info::queue::device	cl_device_id
info::queue::reference_count	cl_uint
info::queue::properties	info::queue_profiling

Data Management: Class buffer [3.4.2]

Useful Typedefs

using value_type = T;

using reference = value_type&;

using const_reference = const value_type&;

Member Functions

template <typename T, int dimensions, typename AllocatorT = cl::sycl::buffer_allocator> **buffer**(const range<dimensions> &*bufferRange*);

template <typename T, int dimensions, typename AllocatorT = cl::sycl::buffer_allocator> **buffer**(const T* *hostData*, const range<dimensions> &*bufferRange*);

template <typename T, int dimensions, typename AllocatorT = cl::sycl::buffer_allocator> **buffer**(T* *hostData*, const range<dimensions> & *bufferRange*);

template <typename T, int dimensions, typename AllocatorT = cl::sycl::buffer_allocator> **buffer**(shared_ptr_class<T>& *hostData*, const range<dimensions> & *bufferRange*, cl::sycl::mutex_class **m*);

template <typename T, int dimensions, typename AllocatorT = cl::sycl::buffer_allocator> **buffer**(unique_ptr_class<void>&& *hostData*, const range<dimensions> & *bufferRange*);

template <typename T, int dimensions=1, typename AllocatorT = cl::sycl::buffer_allocator> **buffer**(Iterator *first*, Iterator *last*);

template <typename T, int dimensions, typename AllocatorT = cl::sycl::buffer_allocator> **buffer**<T, dimensions, AllocatorT> (const buffer <T, dimensions, AllocatorT> &*b*);

template <typename T, int dimensions, typename AllocatorT = cl::sycl::buffer_allocator> **buffer**(buffer<T, dimensions, AllocatorT> &*b*, index<dimensions> &*baseIndex*, range<dimensions> &*subRange*);

template <typename T, int dimensions, typename AllocatorT= cl::sycl::buffer_allocator> **buffer**(cl_mem *memObject*, queue & *fromQueue*, event *availableEvent* = {});

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

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

size_t **get_count**();

size_t **get_size**();

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

template <T> **set_final_data**(weak_ptr_class<T> &*finalData*);

Optional buffer class members

The following members are also available in class buffer when enabled by extension cl_khr_gl_sharing. See OpenGL Interop [4.4.1] for more information.

template <typename T, int dimensions = 1> **buffer**(context &*clGlContext*, GLuint *glBufferObj*);

cl_gl_object_type **get_gl_info**(GLuint *glBufferObj*);

Runtime: Class event [3.3.6]

event() = default;

explicit **event**(cl_event *clEvent*);

event(const event &*rhs*);

cl_event **get**();

vector_class<event> **get_wait_list**();

void **wait**();

void **wait_and_throw**();

static void **wait**(const vector_class<event> &*eventList*);

static void **wait_and_throw**(const vector_class<event> &*eventList*);

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

template <info::event_profiling param> typename param_traits<info::event_profiling, param> ::type **get_profiling_info**() const;

Event queries using get_info()

Descriptor	Return type
info::event::command_type	cl_command_type
info::event::command_execution_status	cl_int
info::event::reference_count	cl_uint
info::event_profiling::command_queued	cl_ulong
info::event_profiling::num_args	cl_ulong
info::event_profiling::reference_count	cl_ulong
info::event_profiling::attributes	cl_ulong

Optional event class members

The following members are also available in class event when enabled by extension cl_khr_gl_event. See OpenGL Interop [4.4.2.4] for more information.

event(context &*clGlContext*, GL_sync *syncObj*);

GL_sync **get_gl_info**();

Data Management: Host Allocation [3.4.1]

The default allocator for memory objects is implementation defined, but users can supply their own allocator class.

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

Default allocators

Allocators	Description
buffer_allocator	Default buffer allocator used by the runtime, when no allocator is defined by the user.
image_allocator	Default image allocator used by the runtime, when no allocator is defined by the user. Must be a byte-sized allocator.

Data Management: Class image [3.4.3]

enum class channel_order:

R,
Rx,
A,
INTENSITY,
LUMINANCE,
RG,
RGx,
RA,
RGB,
RGBx,
RGBA,
ARGB,
BGRA

enum class channel_type:

SNORM_INT8,
SNORM_INT16,
UNORM_INT8,
UNORM_INT16,
UNORM_SHORT_565,
UNORM_SHORT_555,
UNORM_INT_101010,
SIGNED_INT8,
SIGNED_INT16,
SIGNED_INT32,
UNSIGNED_INT8,
UNSIGNED_INT16,
UNSIGNED_INT32,
HALF_FLOAT,
FLOAT

template <int dimensions, typename AllocatorT= cl::sycl::image_allocator> **image**(void **hostPointer*, image_format::channel_order *order*, image_format::channel_type *type*, const range<dimensions> &*range*);

template <int dimensions, typename AllocatorT= cl::sycl::image_allocator> **image**<dimensions>(void **hostPointer*, image_format::channel_order *order*, image_format::channel_type *type*, const range<dimensions> &*range*, const range <dimensions-1> &*pitch*);

(Continued on next page) ▶

◀ Data Management: Class image (cont.)

```
template <int dimensions,
    typename AllocatorT= cl::sycl::image_allocator>
    image(shared_ptr_class<void> &hostPointer,
    image_format::channel_order order,
    image_format::channel_type type,
    const range<dimensions> &range,
    mutex_class *mutex = nullptr);
template <int dimensions,
    typename AllocatorT= cl::sycl::image_allocator>
    image(shared_ptr_class<void> &hostPointer,
    image_format::channel_order order,
    image_format::channel_type type,
    const range<dimensions> &range,
    const range <dimensions-1> &pitch,
    mutex_class *mutex = nullptr);
template <int dimensions,
    typename AllocatorT= cl::sycl::image_allocator>
    image(image<dimensions, AllocatorT> &rhs);
template<int dimensions,
    typename AllocatorT= cl::sycl::image_allocator>
    image<dimensions>(cl_mem memObject,
    queue fromQueue, event availableEvent = {});
image(const image<dimensions, AllocatorT> &rhs);
image<dimensions, AllocatorT> &operator=(
    const image<dimensions, AllocatorT> &rhs);
range<dimensions> get_range();
range<dimensions-1> get_pitch();
size_t get_count();
size_t get_size();
template <access::mode mode,
    access::target target=access::image> accessor<
    T, dimensions, mode, target> get_access();
template <T> set_final_data(
    weak_ptr_class<T> &finalPointer);
```

Optional image class members

The following members are also available in class image when enabled by extension cl_khr_gl_sharing. See OpenGL Interop [4.4.1] for more information.

```
template <int dimensions = 1> image(context &clGlContext,
    GLuint glBufferObj);
template <int dimensions = 2> image(context &clGlContext,
    GLuint glRenderbufferObj);
template <int dimensions = 1> image(context &clGlContext,
    GLenum textureTarget, GLuint glTexture, GLint glMiplevel);
template <int dimensions = 2> image(context &clGlContext,
    GLenum textureTarget, GLuint glTexture, GLint glMiplevel);
template <int dimensions = 3> image(context &clGlContext,
    GLenum textureTarget, GLuint glTexture, GLint glMiplevel);
GLenum get_gl_texture_target();
GLint get_gl_mipmap_level();
```

Data Management: Class accessor [3.4.6]

Member Functions

accessor(buffer< elementType, dimensions> &*bufferRef*, handler &*commandGroupHandler*);	Construct a buffer accessor from a buffer using a command group handler object from the command group scope. Only available for access modes global_buffer, host_buffer, or constant_buffer.
accessor(buffer< elementType, dimensions> &*bufferRef*, handler &*commandGroupHandler*, range<dimensions> *offset*, range<dimensions> *range*);	Construct a buffer accessor from a buffer given a specific range for access permissions and an offset that provides the starting point for the access range using a command group handler object from the command group scope. Only available for access modes global_buffer, host_buffer, or constant_buffer.
accessor(image<dimensions> &*imageRef*, handler &*commandGroupHandler*);	Construct an image accessor from an image using a command group handler object from the command group scope. Only available if accessMode is image or host_image.
accessor(range<dimensions> *allocationSize*, handler &*commandGroupHandler*);	Construct an accessor of dimensions dimensions with elements of type elementType using the passed range to specify the size in each dimension. Only available if accessMode is local.
size_t **get_size**();	Returns the size of the underlying buffer in number of elements.
elementType &**operator[]**(id<dimensions>); elementType &**operator[]**(int); accessor<dimensions-1> &**operator[]**(int);	Return a writeable reference to an element in or slice of the buffer. Available when mode includes non-atomic write permissions and dimensions > 0. An int parameter will return an element reference or a slice of the buffer depending on the value of dimensions.
const elementType &**operator[]** (id<dimensions>); const elementType &**operator[]**(int); accessor<dimensions-1> &**operator[]**(int);	Return the value an element in or slice of the buffer. Available when mode includes non-atomic write permissions and dimensions > 0. An int parameter will return an element reference or a slice of the buffer depending on the value of dimensions.
elementType &**operator[]**();	Return a writeable reference to the element in the buffer. Available when mode includes non-atomic write permissions and dimensions == 0.
const elementType &**operator[]**();	Return the value of the element in the buffer. Available when mode is read-only and dimensions == 0.
operator **elementType**();	Return the value of the element in the buffer. Available when mode is non-atomic and dimensions == 0.
accessor<elementType, 2, mode, image> **operator[]**(size_t *index*);	Returns an accessor to a particular plane of an image array. Available when accessor acts on an image array.
__undefined__ <dimensions-1> &**operator[]** (int)	Return an intermediate type with an additional subscript operator for each subsequent dimension of buffer where (dimensions ¿ 0). Available when mode non-atomic and for access mode read only. The return type is const.
__undefined__ &**operator**()(sampler *sample*) __undefined__ &**operator[]**(id < dimensions >)	Return the value of an element in the image. Available only for the case of an image accessor type.
atomic<elementType> &**operator[]** (id<dimensions>);	Returns a reference to an atomic object, when the accessor is of type access:: global_buffer, access::local_buffer, access::host_buffer, the target mode is access::mode::atomic and dimensions >0.
atomic<elementType> &**operator()**(); atomic<elementType> &**operator***();	Returns a reference to an atomic object, when the accessor is of type access:: global_buffer, access::local_buffer, access::host_buffer, the target mode is access::mode::atomic and dimensions ==0.
local_ptr<elementType> **get_pointer**();	Returns the accessor pointer, when the accessor is of type access::local and mode is non-atomic.
global_ptr<elementType> **get_pointer**();	Returns the accessor pointer, when the accessor is of type access::global_buffer and mode is non-atomic.
constant_ptr<elementType> **get_pointer**();	Returns the accessor pointer, when the accessor is of type access::constant_buffer.
elementType* **get_pointer**();	Returns the accessor pointer, when the accessor is of type access::host_buffer and mode is non-atomic.

Useful Typedefs

```
using value_type = T;
using reference = value_type&;
using const_reference = const value_type&;
```

Access Mode and Target Members [3.4.6.1-2]

enum class access:mode members	
read	Read-only access.
write	Write-only access. Previous contents not discarded.
read_write	Read and write access.
discard_write	Write-only access. Previous contents discarded.
discard_read_write	Read and write access. Previous contents discarded.
atomic	Atomic access.

enum class access:target members	
global_buffer	Access buffer via global memory.
constant_buffer	Access buffer via constant memory.
local	Access work-group-local memory.
image	Access an image.
host_buffer	Access a buffer immediately in host code.
host_image	Access an image immediately in host code.
image_array	Access an array of images on a device.

Optional event class members

The following members are also available in enum access::target when enabled by extension cl_khr_gl_sharing. See OpenGL Interop [4.4.1] for more information.

cl_gl_buffer cl_gl_image

Accessor Capabilities and Restrictions [3.4.6.8]

All accessor types and modes for buffers and local memory

Accessor Type	Access Target	Access Mode	Data Type	Description
Device	global_buffer	read write read_write discard_write discard_read_write	All available data types supported in SYCL.	Access a buffer allocated in global memory on the device.
Host	host_buffer			Access a host allocated buffer on host.
Device	constant_buffer	read		Access a buffer allocated in constant memory on the device.
Device	local	read, write read_write	All supported data types in local memory	Access work-group local buffer, which is not associated with a host buffer. Only accessible on device.

All accessor types and modes for images

Type	Access Target	Access Mode	Data Type	Description
Device	image	read write read_write discard_write discard_read_write	uint4 int4 float4 half4	Access an image on device.
Host	host_image			Access an image on the host.
Device	image_array			Access an array of images on device.

Allowed accessor to accessor conversions

Type	Access Target	Access Mode	Data Type	Description
Device	image	read write read_write discard_write discard_read_write	uint4 int4 float4 half4	Access an image on device.
Device	host_image			Access an image on the host.
Device	image_array			Access an array of images on device.

Data Management: Address Spaces [3.4.7]

Useful Typedefs

The following typedef is defined in each of the four pointer classes.

typedef __undefined__ pointer_t;

Pointer Classes

template <typename ElementType> **global_ptr**(pointer_t);

template <access::mode Mode> **global_ptr**(accessor< ElementType, 1, Mode, global_buffer);

template <typename ElementType> **global_ptr**(const global_ptr &);

template <typename ElementType> **constant_ptr**(pointer_t);

template <access::mode Mode> **constant_ptr**(accessor< ElementType, 1, Mode, constant_buffer);

template <typename ElementType> **constant_ptr**(const constant_ptr &);

template <typename ElementType> **local_ptr**(pointer_t);

template <access::mode Mode> **local_ptr**(accessor< ElementType, 1, Mode, local_buffer);

template <typename ElementType> **local_ptr**(const local_ptr &);

template <typename ElementType> **private_ptr**(pointer_t);

template <typename ElementType> **private_ptr**(const private_ptr &);

Operators

Following are the operators on the the global_ptr, local_ptr, constant_ptr, and private_ptr explicit pointer classes.

template <typename ElementType> ElementType &**operator***();

template <typename ElementType> ElementType &**operator[]** (size_t *i*);

template <typename ElementType> operator **pointer_t**();

Non-member explicit pointer class functions

For the functions below, ***OP*** is one of ==, !=, <, >, >=, <=

template <typename ElementType> bool **operator*OP***(const global_ptr<ElementType> &*lhs*, const global_ptr<ElementType> &*rhs*);

template <typename ElementType> bool **operatorOP**(const constant_ptr<ElementType> &*lhs*, const constant_ptr<ElementType> &*rhs*);

template <typename ElementType> bool **operatorOP**(const local_ptr<ElementType> &*lhs*, const local_ptr<ElementType> &*rhs*);

template <typename ElementType> bool **operatorOP**(const private_ptr<ElementType> &*lhs*, const private_ptr<ElementType> &*rhs*);

template <typename ElementType> bool **operator*OP***(const global_ptr<ElementType> &*lhs*, nullptr_t *rhs*);

template <typename ElementType> bool **operator*OP***(const constant_ptr<ElementType> &*lhs*, nullptr_t *rhs*);

template <typename ElementType> bool **operator*OP***(const local_ptr<ElementType> &*lhs*, nullptr_t *rhs*);

template <typename ElementType> bool **operator*OP***(const private_ptr<ElementType> &*lhs*, nullptr_t *rhs*);

template <typename ElementType> bool **operator*OP***(nullptr_t *lhs*, const global_ptr<ElementType> &*rhs*);

template <typename ElementType> bool **operator*OP***(nullptr_t *lhs*, const constant_ptr<ElementType> &*rhs*);

template <typename ElementType> bool **operator*OP***(nullptr_t *lhs*, const local_ptr<ElementType> &*rhs*);

template <typename ElementType> bool **operator*OP***(nullptr_t *lhs*, const private_ptr<ElementType> &*rhs*);

Explicit Pointer Classes [3.4.7.1]

Explicit pointer classes:	OpenCL address space:	Compatible accessor target:
global_ptr	__global	global_buffer
constant_ptr	__constant	constant_buffer
local_ptr	__local	local
private_ptr	__private	none

Multi-pointer class [3.4.7.2]

In selected functions below, the placeholder *addressSpace* may be one of global, constant, local, or private and must be consistently applied per function. ***OP*** is one of ==, !=, <, >, >=, <=

typedef __undefined__ pointer_t;

const address_space space;

template <typename ElementType, enum address_space Space> explicit **multi_ptr**(pointer_t);

template <typename ElementType, access:: address_space Space> **multi_ptr**(const multi_ptr &);

template <typename ElementType, access:: address_space Space>multi_ptr<ElementType, Space> **make_ptr**(pointer_t);

template <typename ElementType, access:: address_space Space> ElementType & **operator***();

template <typename ElementType, access:: address_space Space> ElementType & **operator[]**(size_t *i*);

template <typename ElementType, access::address_space Space = access::address_space:: global_space> **operator global_ptr**<ElementType>();

template <typename ElementType, access::address_space Space = access::address_space:: constant_space> **operator constant_ptr**<ElementType>();

template <typename ElementType, access::address_space Space = access::address_space:: local_space> **operator local_ptr**<ElementType>();

template <typename ElementType, access::address_space Space = access::address_space:: private_space> **operator private_ptr**<ElementType>();

template <typename ElementType, access::address_space Space = access::address_space:: global_space> global_ptr<ElementType>**pointer**();

template <typename ElementType, access::address_space Space = access::address_space:: constant_space> constant_ptr<ElementType>**pointer**();

template <typename ElementType, access::address_space Space = access::address_space:: local_space> local_ptr<ElementType>**pointer**();

template <typename ElementType, access::address_space Space = access::address_space:: private_space> private_ptr<ElementType>**pointer**();

template <typename ElementType, access::address_space Space> bool **operator*OP***(const multi_ptr<ElementType, Space> &*lhs*, const multi_ptr<ElementType, Space> &*rhs*);

template <typename ElementType, access::address_space Space> bool **operator*OP***(const multi_ptr< ElementType, Space> &*lhs*, nullptr_t *rhs*);

template <typename ElementType, access::address_space Space> bool **operator*OP***(nullptr_t *lhs*, const multi_ptr<ElementType, Space> &*rhs*);

Data Management: Class sampler [3.4.8]

enum class sampler_addressing_mode

SYCL_SAMPLER_ADDRESS_MIRRORED_REPEAT,
SYCL_SAMPLER_ADDRESS_REPEAT,
SYCL_SAMPLER_ADDRESS_CLAMP_TO_EDGE,
SYCL_SAMPLER_ADDRESS_CLAMP,
SYCL_SAMPLER_ADDRESS_NONE

enum class sampler_filter_mode

SYCL_SAMPLER_FILTER_NEAREST,
SYCL_SAMPLER_FILTER_LINEAR

Class sampler members

sampler(bool *normalized_coords*, sampler_addressing_mode *addressing_mode*, sampler_filter_mode *filter_mode*);

sampler(cl_sampler);

sampler_addressing_mode **get_address**() const;

sampler_filter_mode **get_filter**() const;

cl_sampler **get_sampler**() const;

Expressing Parallelism: Class kernel [3.5.4]

kernel(cl_kernel *openclKernelObj*);

kernel(const kernel &*rhs*);

cl_kernel **get**();

context **get_context**();

program **get_program**();

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

Kernel queries using get_info()

Descriptor	Return type
info::kernel::function_name	string_class
info::kernel::num_args	cl_uint
info::kernel::reference_count	cl_uint
info::kernel::attributes	string_class

Expressing Parallelism: Ranges and Index Space Identifiers [3.5.1]

Class range members [3.5.1.1]

range(const range <dimensions>&);

range(size_t *x*);

range(size_t *x*, size_t *y*);

range(size_t *x*, size_t *y*, size_t *z*);

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

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

range &**operator*OP***(const range &*rhs*);

where ***OP*** is one of [], =, +=, *=, /=, %=, >>=, <<=, &=, ^=, |=

size_t **size**() const;

template <size_t dimensions> bool **operator*OP***(const range<dimensions> &*a*, const range<dimensions> &*b*);

where ***OP*** is one of ==, !=, >, <, >=, <=

template <int dimensions> range <dimensions> **operatorOP**(range<dimensions> *a*, range<dimensions> *b*);

where ***OP*** is one of *, +, -

template <int dimensions> range <dimensions> **operator/**(range<dimensions> *dividend*, range<dimensions> *divisor*);

template <size_t dimensions> range <dimensions> **operator*OP***(const range<dimensions> &*a*, const range<dimensions> &*b*);

where ***OP*** is one of %, <<, >>, &, |, &&, ||, ^

template <size_t dimensions> range <dimensions> **operatorOP**(const size_t &*a*, const range<dimensions> &*b*);

where ***OP*** is one of *, /, +, -, %, <<, >>

template <size_t dimensions> range <dimensions> **operatorOP**(const range<dimensions> &*a*,const size_t &*b*);

where ***OP*** is one of *, /, +, -, %, <<, >>

Class nd_range members [3.5.1.2]

nd_range(const nd_range<dimensions> &)

nd_range<dimensions>(range<dimensions> *globalSize*, range<dimensions> *localSize*, id<dimensions> offset = id<dimensions>());

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

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

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

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

Class id members [3.5.1.3]

id(size_t *x*);

id(size_t *x*, size_t *y*);

id(size_t *x*, size_t *y*, size_t *z*);

id(const id &);

id(const range &*r*);

id(const item &*it*);

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

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

operator size_t();

(Continued on next page) ▶

◀Expressing Parallelism: Ranges and Index Space Identifiers (continued)

```
id &operatorOP(const id &rhs);
```
where *OP* is one of =, +=, *=, /=, %=, >>=, <<=, &=, ^=, |=

```
template <size_t dimensions> bool operatorOP(
    const id<dimensions> &a, const id<dimensions> &b);
```
where *OP* is one of ==, !=, >, <, >=, <=

```
template <size_t dimensions> id<dimensions>
    operatorOP(const id<dimensions> &a,
    const id<dimensions> &b);
```
where *OP* is one of *, +, -, %, <<, >>, &, |, ^, &&, ||template <size_t dimensions> id<dimensions> operator/(
const id<dimensions> &dividend,
const id<dimensions> &divisor);

```
template <size_t dimensions> id<dimensions>
    operatorOP(const size_t &a, const id<dimensions> &b);
```
where *OP* is one of *, +, -, /, %, <<, >>

```
template <size_t dimensions> id<dimensions>
    operatorOP(const id <dimensions> &a, const size_t &b);
```
where *OP* is one of *, +, -, /, %, <<, >>

Class item members [3.5.1.4]

```
id<dimensions> get() const;
size_t get(int dimension) const;
size_t &operator[](int dimension);
range<dimensions> get_range() const;
id<dimensions> get_offset() const;
size_t get_linear_id() const;
```

Class nd_item members [3.5.1.5]

```
id<dimensions> get_global() const;
size_t get_global(int dimension) const;
id<dimensions> get_global_linear_id() const;
id<dimensions> get_local() const;
size_t get_local(int dimension) const;
id<dimensions> get_local_linear_id() const;
group<dimensions> get_group() const;
size_t get_group(int dimension) const;
size_t get_group_linear_id() const;
int get_num_groups(int dimension) const;
id<dimensions> get_num_groups() const;
range<dimensions> get_global_range() const;
range<dimensions> get_local_range() const;
id<dimensions> get_offset() const;
nd_range<dimensions> get_nd_range() const;
void barrier(access::fence_space flag=
    access::global_and_local) const;
```

Class group members [3.5.1.6]

```
id<dimensions> get();
size_t get(int dimension) const;
range<dimensions> get_global_range();
size_t get_global_range(int dimension);
range<dimensions> get_group_range();
size_t get_group_range(int dimension);
size_t operator[](int dimension) const;
size_t get_linear() const;
```

Examples of How to Invoke Kernels

Example: single_task invoke [3.5.3.1]

SYCL provides a simple interface to enqueue a kernel that will be sequentially executed on an OpenCL device.

```
auto command_group = [&](handler & cgh)
{
cgh.single_task<class kernel_name>(
    [=] () {
        // [kernel code]
    }));
};
```

Examples: parallel_for invoke [3.5.3.2]

Example #1

Using a lambda function for a kernel invocation.

```
class MyKernel;
myQueue.submit( [&](handler & cmdgroup)
{
    auto acc=myBuffer.get_access<read_write>();
    cmdgroup.parallel_for<class MyKernel>(range<1>(
                workItemNo), [=] (id<1> index)
    {
        acc[index] = 42.0f;
    });
});
```

Example #2

Allowing the runtime to choose the index space best matching the range provided using information given the scheduled interface instead of the general interface.

```
class MyKernel;
myQueue.submit([&](handler & cmdgroup)
 {
    auto acc=myBuffer.get_access<read_write>();
    cmdgroup.parallel_for<class MyKernel>(range<1>(
      workItemNo), [=] (item<1> myItem)
    {
        size_t index = item.get_global();
        acc[index] = 42.0f;
    });
});
```

Example #3

Two examples of launching a kernel functor over a 3D grid. In the first eample, work_item IDs range from 0 to 2.

```
auto command_group = [&](handler & cgh)
{
    cgh.parallel_for<class example_kernel1>(
        range<3>(3,3,3), // global range
            [=] (item<3> it)
            {
                //[kernel code]
            });
};
```

In this second example of launching a kernel functor over a 3D grid, work_item IDs run from 1 to 3.

```
auto command_group2 = [&](handler & cgh)
{
    cgh.parallel_for<class example_kernel2>(
        range<3>(3,3,3), // global range
        id<3>(1,1,1), // offset
            [=] (item<3> it)
            {
                //[kernel code]
            });
};
```

Example #4

Launching sixty-four work-items in a three-dimensional grid with four in each dimension and divided into sixteen work-groups.

```
auto command_group = [&](handler& cgh)
{
    cgh.parallel_for<class example_kernel>(
        nd_range(range(4, 4, 4), range(2, 2, 2)), [=](
            nd_item<3> item)
        {
            //[kernel code]
            // Internal synchronization
            item.barrier(access::fence_space::global);
            //[kernel code]
        });
};
```

Parallel For hierarchical invoke [3.5.3.3]

```
auto command_group = [&](handler & cgh)
{
    // Issue 8 work-groups of 8 work-items each
    cgh.parallel_for_work_group<class example_kernel>(
        range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup)
    {
        //[workgroup code]
        int myLocal; // this variable shared between workitems
        // this variable instantiated for each work-item separately
        private_memory<int> myPrivate(myGroup);
        // Issue parallel sets of work-items each sized
        // using the runtime default
        parallel_for_work_item(myGroup, [=](item<3> myItem) {
            // [work-item code]
            myPrivate(myItem) = 0;
        });

        // Carry private value across loops
        parallel_for_work_item(myGroup, [=](item<3> myItem) {
            //[work-item code]
            output[myGroup.get_local_range()*
                    myGroup.get()+myItem] = myPrivate(myItem);
        });
        //[workgroup code]
    });
});
```

Expressing Parallelism: Command Group Class handler [3.5.3.4]

```
handler(const handler &rhs);

void set_arg(int index, accessor & accObj);

template <typename T> void set_arg(int index,
    accessor & accObj);

template <typename KernelName, class KernelType>
    void single_task(KernelType);

template <typename KernelName, class KernelType>
    void parallel_for(
    range<dimensions> numWorkItems, KernelType);

template <typename KernelName, class KernelType>
    void parallel_for(
    range<dimensions> numWorkItems,
    id<dimensions> workItemOffset, KernelType);

template <typename KernelName, class KernelType>
    void parallel_for(nd_range<dimensions> executionRange,
    KernelType);

template <typename KernelName, class KernelType>
    void parallel_for(nd_range<dimensions> numWorkItems,
    id<dimensions> workItemOffset, KernelType);

template <class KernelName, class WorkgroupFunctionType>
    void parallel_for_work_group(
    range<dimensions> numWorkGroups,
    WorkgroupFunctionType);

template <class KernelName, class WorkgroupFunctionType>
    void parallel_for_work_group(
    range<dimensions> numWorkGroups,
    range<dimensions> workGroupSize,
    WorkgroupFunctionType);

void single_task(kernel syclKernel);

void parallel_for(range<dimensions> numWorkItems,
    kernel syclKernel);

void parallel_for(nd_range<dimensions> ndRange,
    kernel syclKernel);
```

Non member function

```
template <class KernelType> void parallel_for_work_item(
    group numWorkItems,
    range<dimensions> localSize,  KernelType);
```

Expressing Parallelism: Class program [3.5.5]

```
explicit program(const context &context);
program(const context &context,
    vector_class<device> deviceList);
program(vector_class<program> programList,
    string_class linkOptions ="");
program(const context &context, cl_program clProgram);
program(const program &rhs);
template <typename kernelT>
    void compile_from_kernel_name(
    string_class compileOptions ="");
template <typename kernelT>
    void build_from_kernel_name(
    string_class compileOptions = "");
void link(string_class linking_options ="");
template <info::program param>
    typename info::param_traits<
    info::program, param>::type get_info();
vector_class<char*> get_binaries() const;
vector_class<device> get_devices() const;
string_class get_build_options() const;
cl_program get() const;
```

Information descriptors:

Descriptor	Return type
info::program::reference_count	cl_uint
info::program::context	cl_context
info::program::devices	vector_class<cl_device_id>

Data Types

SYCL supports the C++11 ISO standard data types.

Scalar data types [3.7.1]

SYCL integral data types:

char, unsigned char
short int, unsigned short int
int, unsigned int
long int, unsigned long int
long long int, unsigned long long int
size_t

SYCL floating point data types:

float
double
half

SYCL scalar data types:

The following data types are defined in the cl::sycl namespace:

cl_bool
cl_char, cl_uchar
cl_short, cl_ushort
cl_int, cl_uint
cl_long, cl_ulong
cl_float
cl_double
cl_half

Vector data types [3.7.2]

genvector = all possible vector types.

```
typedef dataT element_type;
typedef __undefined__ vector_t;
vec<T, dims>();
explicit vec<T, dims>(const T &arg);
vec<T, dims> (const T &element_0, const T &element_1,
    . . . , const T &element_dims-1);
vec<T, dims>(const &vec<T, dims>);
size_t get_count();
size_t get_size();
template<typename asDataT, int width>
    vec<asDataT, width> as() const;
operator genvector() const;
operator vec<dataT, numElements>(genvector clVector) const;
vec<dataT, numElements> operatorOP(const vec<dataT,
    numElements> &rhs);
```

OP is one of +, -, *, /, %, |, ^, &&, ||, >>, <<, +=, -=, *=, /=, |=, ^=, <<=, >>=, &=, %=, =

```
vec<dataT, numElements> operatorOP(const dataT, &rhs);
```

OP is one of +, -, *, /, %, |, ^, &&, ||, >>, <<, +=, -=, *=, /=, |=, ^=, <<=, >>=, &=, %=, =

```
vec<dataT, numElements> operatorOP();
```

OP is one of ++, --, ~, !

```
vec<dataT, numElements> operatorOP(int);
```

OP is one of ++, --

```
bool operatorOP(const vec<dataT, numElements> &rhs)
    const;
```

OP is one of ==, !=

```
swizzled_vec<T, out_dims> swizzle<elem s1, ...>();

#ifdef SYCL_SIMPLE_SWIZZLES
    swizzled_vec<T, 4> xyzw();
    ...
#endif    // #ifdef SYCL_SIMPLE_SWIZZLES
```

Kernel functor example [3.5.6.1]

In the following example the kernel is executed on a unary index space for the specific example, since it's using single_ask as its invocation.

```
class MyFunctor
{
    float m_parameter;
public:
    MyFunctor(float parameter):
        m_parameter(parameter)
    {
    }
    void operator() ()
    {
        // [kernel code]
    }
};
void workFunction(float scalarValue)
{
    MyFunctor myKernel(scalarValue);
    queue.submit([&] (handler & cmdGroup)
    {
        cmdgroup.single_task(myKernel);
    });
}
```

Kernel lambda example [3.5.6.2]

To invoke a C++11 lambda, the kernel name must be included explicitly by the user as a template parameter tothe kernel invoke function.

```
class MyKernel;
command_queue.submit([&](handler& cmdGroup)
{
    cmdgroup.single_task<class MyKernel>([=]()
    {
        // [kernel code]
    });
});
```

Kernel program object example [3.5.6.3]

In the following example, the kernel is defined as a lambda function. The example obtains the program object for the lambda function kernel and then passes it to the parallel_for.

```
class MyKernel; // Forward declaration of the
                // name of the lambda functor
cl::sycl::queue myQueue;
cl::sycl::program MyProgram(myQueue.get_context());

/* use the kernel name to obtain the associated program */
MyProgram.build_from_name<MyKernel>();

myQueue.submit([&](handler& commandGroup)
{
    commandgroup.parallel_for<class MyKernel>(
        cl::sycl::nd_range<2>(4, 4),
        // execute the kernel as compiled in MyProgram
        MyProgram, get_kernel<MyKernel>(),
        ([=](cl::sycl::nd_item<2> index)
        {
            // [kernel code]
        }));
});
```

Kernel program object example

In the next example, a SYCL kernel is linked with an existing pre-compiled OpenCL C program object to create a combined program object, which is then called in a parallel_for.

```
class MyKernel;  // Forward declaration of the name
                 // of the lambda functor
cl::sycl::queue myQueue;
// obtain an existing OpenCL C program object
cl_program myClProgram = ...;

// Create a SYCL program object from a cl_program object
cl::sycl::program myExternProgram(myQueue.get_context(),
myClProgram);

// Release the program if we no longer need it as
// SYCL program retained a reference to it
clReleaseProgram(myClProgram);

// Add in the SYCL program object for our kernel
cl::sycl::program mySyclProgram(myQueue.get_context());
mySyclProgram.compile_from_kernel_name<MyKernel>("-
my-compile-options");

// Link myClProgram with the SYCL program object
mySyclProgram.link(myExternProgram,"-my-link-options");

myQueue.submit([&](handler& commandgroup)
{
    commandgroup.parallel_for<class MyKernel>(
        cl::sycl::nd_range<2>(4, 4),
        // execute the kernel as compiled in MyProgram
        myLinkedProgram.get_kernel<MyKernel>(),
        ([=](cl::sycl::nd_item<2> index)
        {
            //[kernel code]
        }));
});
```

Kernels from OpenCL C kernel objects [3.5.6.4]

The following creates a SYCL kernel object from an OpenCL kernel object.

```
kernel::kernel(cl_kernel kernel)
```

Stream Class [3.9]

SYCL stream class is available in SYCL instead of the printf() function. The cl::sycl::stream object can be used to output a sequence of characters on standard output. Operators are in the cl::sycl namespace.

Stream operator	Description
stream=	Add one stream to another
operator<<	Output characters or structs on standard output
endl	Add an end of line character
hex	Display output in hexadecimal base

Stream operator	Description
oct	Display output in octal base
setw	Set field width
precision	Set decimal precision
scientific	Set float value notation to C++ scientific
fixed	Set float value notation to C++ fixed
hexfloat	Display float values in hexadecimal format
defaultfloat	Display float values in default notation

Exception Classes [3.6.2]

Class: exception

string_class **what**();
context **get_context**();

Class: cl_exception

cl_int **get_cl_code**();

Class: exception_list

size_t **size**() const;
iterator **begin**() const;
iterator **end**() const;

Exception types

Derived from cl::sycl::runtime_error class
kernel_error
nd_range_error
accessor_error
event_error
invalid_parameter_error

Derived from cl::sycl::device_error class
compile_program_error
link_program_error
invalid_object_error
memory_allocation_error
device_error
platform_error
profiling_error
feature_not_supported

Synchronization & Atomics [3.8]

void **store**(T *operand*,
std::memory_order = std::memory_order_relaxed);

T **load**(memory_order = std::memory_order_relaxed) const;

T **exchange**(T *operand*,
std::memory_order = std::memory_order_relaxed);

T **compare_exchange_strong**(T& *expected*, T *desired*,
std::memory_order *success*, std::memory_order *fail*);

T **fetch_add**(T *operand*,
std::memory_order = std::memory_order_relaxed);

T ***F***(T *operand*, std::memory_order *order*);
F may be one of fetch_sub, fetch_and, fetch_or, fetch_xor, fetch_min, fetch_max

template <class T> T **atomic_load_explicit**(
atomic<T>* *object*, std::memory_order *order*);

template <class T> void **atomic_store_explicit**(
atomic<T>* *object*, T *operand*, std::memory_order *order*);

template <class T> T **atomic_exchange_explicit**(
atomic<T>* *object*, T *operand*, std::memory_order *order*);

template <class T>
bool **atomic_compare_exchange_strong_explicit**(
atomic<T>* *object*, T* *expected*, T *desired*,
std::memory_order *success*,
std::memory_order *fail*);

template <class T> T ***F***(atomic<T>* *object*, T *operand*,
std::memory_order *order*);
F may be one of atomic_exchange_explicit, atomic_fetch_add_explicit, atomic_fetch_sub_explicit, atomic_fetch_and_explicit, atomic_fetch_or_explicit, atomic_fetch_xor_explicit, atomic_fetch_min_explicit, atomic_fetch_max_explicit

Math Functions [3.10.3]

Tf (genfloat in the spec) is type float, float*n*, double, or double*n*.
sTf (sgenfloat in the spec) is type float or double.
Ti (genint in the spec) is type int or int*n*. *n* is 2, 3, 4, 8, or 16.
N indicates that native variants are available.
Functions shown in brown text are available in native forms only.

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 x*, *Tf y*)		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*)		Cosine
Tf **cosh** (*Tf x*)	N	Hyperbolic cosine
Tf **cospi** (*Tf x*)		cos (π *x*)
Tf **divide** (*Tf x*, *Tf y*)		*x* / *y*
Tf **erfc** (*Tf x*)		Complementary error function
Tf **erf** (*Tf x*)		Calculates error function
Tf **exp** (*Tf x*)	N	Exponential base e
Tf **exp2** (*Tf x*)	N	Exponential base 2
Tf **exp10** (*Tf x*)	N	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*)
float*n* **fract** (float*n* *x*, int*n* **iptr*) float **fract** (float *x*, int **iptr*)		Fractional value in *x*
double*n* **frexp** (double*n* *x*, int*n* **exp*) double **frexp** (double *x*, int **exp*)		Extract mantissa and exponent
Tf **hypot** (*Tf x*, *Tf y*)		Square root of $x^2 + y^2$
int **logb** (float *x*) int*n* **ilogb** (*Tf x*) int **logb** (double *x*) int*n* **logb** (double*n* *x*)		Return exponent as an integer value
Tf **ldexp** (*Tf x*, *Ti k*) float*n* **ldexp** (float*n* *x*, int *k*) double*n* **ldexp** (double*n* *x*, int *k*)		$x * 2^n$
Tf **lgamma** (*Tf x*) *Ts* **lgamma_r** (*Tf x*, *Ti* **signp*)		Log gamma function
Tf **log** (*Tf*)	N	Natural logarithm
Tf **log2** (*Tf*)	N	Base 2 logarithm
Tf **log10** (*Tf*)	N	Base 10 logarithm
Tf **log1p** (*Tf x*)		ln (1.0 + *x*)
Tf **logb** (*Tf x*)		Exponent of *x*
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
float*n* **nan** (uint*n* *nancode*) float **nan** (unsigned int *nancode*) double*n* **nan** (ulonglong*n* *nancode*) double **nan** (unsigned long long int *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*)	N	Compute x^y, where *x* is >= 0
Tf **recip** (*Tf x*)		1 / *x*
Tf **remainder** (*Tf x*, *Tf y*)		Floating point remainder
Tf **remquo** (*Tf x*, *Tf y*, *Ti* **q*)		Remainder and quotient
Tf **rint** (*Tf*)		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*)	N	Inverse square root
Tf **sin** (*Tf*)	N	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*)	N	Square root
Tf **tan** (*Tf x*)	N	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

Common Functions [3.10.5]

Tf (genfloat in the spec) is float, float*n*, double or double*n* types.
Tff (genfloatf in the spec) is float or float*n* types.
Tfd (genfloatd in the spec) is double or double*n* types.
n is 2, 3, 4, 8, or 16.

Tf **clamp** (*Tf x*, *Tf minval*, *Tf maxval*); float*n* **clamp** (float*n* *x*, float *minval*, float *maxval*); double*n* **clamp** (double*n* *x*, double *minval*, double*n* *maxval*);	Clamp *x* to range given by *minval*, *maxval*

Tf **degrees** (*Tf radians*);	*radians* to degrees
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*

Integer Functions [3.10.4]

T (gentype in the spec) is all integer and float types.
Ti (geninteger in spec) is all signed and unsigned integer types.
Tsi (sgeninteger in the spec) is all scalar integer types.
Tui (ugeninteger in the spec) is all unsigned integer types.

Tui **abs** (*Ti x*)	| *x* |
Tui **abs_diff** (*Ti x*, *Ti y*)	| *x* – *y* | without modulo overflow
Ti **add_sat** (*Ti x*, *Ti y*)	*x* + *y* and saturates the result
Ti **hadd** (*Ti x*, *Ti y*)	(*x* + *y*) >> 1 without mod. overflow
Ti **rhadd** (*Ti x*, *Ti y*)	(*x* + *y* + 1) >> 1
Ti **clamp** (*Ti x*, *Ti min*, *Ti max*) *Ti* **clamp** (*Ti x*, *Tsi min*, *Tsi max*)	min(max(*x*, *min*), *max*)
Ti **clz** (*Ti x*)	number of leading 0-bits in *x*
Ti **mad_hi** (*Ti a*, *Ti b*, *Ti c*)	mul_hi(*a*, *b*) + *c*
Ti **mad_sat** (*Ti a*, *Ti b*, *Ti c*)	*a* * *b* + *c* and saturates the result
Ti **max** (*Ti x*, *Ti y*) *Ti* **max** (*Ti x*, *Tsi y*)	*y* if *x* < *y*, otherwise it returns *x*
Ti **min** (*Ti x*, *Ti y*) *Ti* **min** (*Ti x*, *Tsi y*)	*y* if *y* < *x*, otherwise it returns *x*
Ti **mul_hi** (*Ti x*, *Ti y*)	high half of the product of *x* and *y*
Ti **popcount** (*Ti x*)	Number of non-zero bits in *x*
Ti **rotate** (*Ti v*, *Ti i*)	result[*indx*] = *v*[*indx*] << *i*[*indx*]
Ti **sub_sat** (*Ti x*, *Ti y*)	*x* - *y* and saturates the result
T **popcount** (*T x*)	Number of non-zero bits in *x*
short*n* **upsample** (char*n hi*, uchar*n lo*)	result[*i*]= ((short)*hi*[*i*]<< 8) | *lo*[*i*]
ushort*n* **upsample** (uchar*n hi*, uchar*n lo*)	result[*i*]=((ushort)*hi*[*i*]<< 8) | *lo*[*i*]
int*n* **upsample** (short*n hi*, ushort*n lo*)	result[*i*]=((int)*hi*[*i*]<< 16) | *lo*[*i*]
uint*n* **upsample** (ushort*n hi*, ushort*n lo*)	result[*i*]=((uint)*hi*[*i*]<< 16) | *lo*[*i*]
longlong*n* **upsample** (int*n hi*, uint*n lo*)	result[*i*]=((long)*hi*[*i*]<< 32) | *lo*[*i*]
ulonglong*n* **upsample** (uint*n hi*, uint*n lo*)	result[*i*]=((ulong)*hi*[*i*]<< 32) | *lo*[*i*]
int*n* **mad24** (int*n x*, int*n y*, int*n z*) uint*n* **mad24** (uint*n x*, uint*n y*, uint*n z*)	Multiply 24-bit integer values *x, y,* add 32-bit int. result to 32-bit integer *z*
int*n* **mul24** (int*n x*, int*n y*) uint*n* **mul24** (uint*n x*, uint*n y*)	Multiply 24-bit integer values *x* and *y*

Geometric Functions [3.10.6]

T is type float or double, and must be consistently applied per function. *n* is 2, 3, 4, 8, or 16.

T{3,4} **cross** (*T*{3,4} *p0*, *T*{3,4} *p1*)	Cross product
T **distance** (*Tn p0*, *Tn p1*)	Vector distance
T **dot** (*Tn p0*, *Tn p1*)	Dot product
T **length** (*Tn p*)	Vector length
Tn **normalize** (*Tn p*)	Normal vector length 1
float **fast_distance** (float*n p0*, float*n p1*)	Vector distance
float **fast_length** (float*n p*)	Vector length
float*n* **fast_normalize** (float*n p*)	Normal vector length 1

Relational Built-in Functions [3.10.7]

T (gentype in the spec) is all signed , unsigned, float, double, scalar and vector types.
Ti (geninteger in spec) is signed and unsigned integer types.
Tui (ugeninteger in the spec) is all unsigned integer types.

In the prototypes shown below, ***F*** may be one of the following: isequal, isnotequal, isgreater, isgreaterequal, isless, islessequal, islessgreater, isordered, isunordered

int ***F*** (float *x*, float *y*)
int*n* ***F*** (float*n x*, float*n y*)
longlong ***F*** (double *x*, double *y*)
longlong*n* ***F*** (double*n x*, double*n y*)

In the prototypes shown below, ***F*** may be one of the following: isfinite, isinf, isnan, isnormal, signbit

int ***F*** (float)
int*n* ***F*** (float*n*)
longlong ***F*** (double)
longlong*n* ***F*** (double*n*)

int **any** (*Ti x*)	1 if MSB in component of *x* is set; else 0
int **all** (*Ti 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
T **select** (*T a*, *T b*, *Ti c*) *T* **select** (*T a*, *T b*, *Tui 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*

Half precision floating-point [4.2]

The following functions are available when extension cl_khr_fp16 is enabled. *Th* (genhalf in the spec) is half or half*n* types.
n is 2, 3, 4, 8, or 16.

Th ***F*** (*Th x*, *Th y*)	***F*** may be one of: divide, powr
Th ***F*** (*Th x*)	***F*** may be one of the following: cos, exp, exp2, exp10, log, log2, log10, recip, rsqr, sin, sqrt, tan

OpenGL Interop [4.4.1]

Enable OpenGL interoperability using the extension cl_khr_gl_sharing. When enabled, this extension allows for the following.

cl_context_properties:

cl_context_properties flag	Description
CL_GL_CONTEXT_KHR	OpenGL context handle (default: 0)
CL_EGL_DISPLAY_KHR	CGL share group handle (default: 0)
CL_GLX_DISPLAY_KHR	EGLDisplay handle (default: EGL_NO_ DISPLAY)
CL_WGL_HDC_KHR	X handle (default: None)
CL_CGL_SHAREGROUP_KHR	HDC handle (default: 0)

Class members

Additional members are added to the classes context [3.3.3], buffer [3.4.2], image [3.4.3], and event [3.3.6].

Mapping of GL internal format to CL image format

The compatible formats are specified in Table 9.4 of the OpenCL 1.2 Extension Specification document and are also included in the table below.

OpenGL internal format	Corresponding OpenCL image format (channel order, channel data type)
GL_RGBA8	CL_RGBA, CL_UNORM_INT8, CL_BGRA, CL_UNORM_INT8
GL_RGBA, GL_UNSIGNED_INT_8_8_8_8_REV	CL_RGBA, CL_UNORM_INT8
GL_BGRA, GL_UNSIGNED_INT_8_8_8_8_REV	CL_BGRA, CL_UNORM_INT8
GL_RGBA16	CL_RGBA, CL_UNORM_INT16
GL_RGBA8I, GL_RGBA8I_EXT	CL_RGBA, CL_SIGNED_INT8
GL_RGBA16I, GL_RGBA16I_EXT	CL_RGBA, CL_SIGNED_INT16
GL_RGBA32I, GL_RGBA32I_EXT	CL_RGBA, CL_SIGNED_INT32
GL_RGBA8UI, GL_RGBA8UI_EXT	CL_RGBA, CL_UNSIGNED_INT8
GL_RGBA16UI, GL_RGBA16UI_EXT	CL_RGBA, CL_UNSIGNED_INT16
GL_RGBA32UI, GL_RGBA32UI_EXT	CL_RGBA, CL_UNSIGNED_INT32
GL_RGBA16F, GL_RGBA16F_ARB	CL_RGBA, CL_HALF_FLOAT
GL_RGBA32F, GL_RGBA32F_ARB	CL_RGBA, CL_FLOAT

Preprocessor Directives & Macros [5.6]

CL_SYCL_LANGUAGE_VERSION	Integer version, e.g.: 120
__FAST_RELAXED_MATH__	-cl-fast-relaxed-math
__SYCL_DEVICE_ONLY__	produce no host binary
__SYCL_SINGLE_SOURCE__	produce host and device binary
__SYCL_TARGET_SPIR__	produce SPIR binary
SYCL_EXTERNAL	allow kernel external linkage

Enable OpenCL Extensions in SYCL [4.1]

To enable an optional extension:
#pragma OPENCL EXTENSION <*extension_name*> : <*behavior*>
extension_name: all, or an extension name from the table below
behavior: enable, disable

cl_apple_gl_sharing
cl_khr_3d_image_writes
cl_khr_d3d10_sharing
cl_khr_d3d11_sharing
cl_khr_dx9_media_sharing
cl_khr_fp16
cl_khr_gl_sharing
cl_khr_int64_base_atomics
cl_khr_int64_extended_atomics