KHRONOS GROUP

The **OpenCL** C++ Wrapper API

Version: 1.2.6

Document Revision: 01

Khronos OpenCL Working Group

Authors: Benedict R. Gaster and Lee Howes

Contributers: Simon McIntosh-Smith, Tom Deakin and Bruce Merry

1. Introduction

This specification describes the OpenCL C++ wrapper API (Version 1.2). It should be used in conjunction with the *OpenCL Specification,* Version 1.2. The C++ wrapper is built on top of the OpenCL C API and is not a replacement for it. An implementation of the C++ Wrapper API calls the underlying C API, which is assumed to be a compliant implementation of the *OpenCL Specification* platform and runtime API at version 1.2 or below. The C++ API corresponds closely to the underlying C API and introduces no additional execution overhead. C++ header macros adapt to the underlying C API version against which the header is compiled.

The wrapper interface is defined within a single C++ header file **cl.hpp**. All its definitions are contained within namespace **cl**. There is no additional requirement to include **cl.h**; to use either the C++ wrapper or the original C API, simply include **cl.hpp**.

The API is divided into a number of classes with corresponding OpenCL C types. For example, class **cl::Memory** maps to OpenCL type **cl_mem**. When possible, C++ inheritance provides an extra level of type correctness and abstraction. For example, class **cl::Buffer** derives from base class **cl::Memory** but represents the 1D memory subclass of all possible OpenCL memory objects.

The following sections describe each class in detail. The index section at the end of this document lists each class, constructor, and method defined by the C++ wrapper API.

2. C++ Platform layer

2.1 Platforms

Class **cl::Platform** provides functionality for working with OpenCL platforms.
The following static method lists the available platforms[1]:

static cl_int **cl::Platform::get**(VECTOR_CLASS<Platform> * *platforms*)

platforms is a vector of OpenCL platforms found.

cl::Platform::get returns CL_SUCCESS on success. Otherwise, it returns the following error:

- CL_INVALID_VALUE if *platforms* is NULL.

The method

cl_int **cl::Platform::getInfo**(cl_platform_info *name*,
STRING_CLASS * *param*)

[1] Section 5 describes C++ classes VECTOR_CLASS and STRING_CLASS.

gets specific information about the OpenCL platform. Table 4.1 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_platform_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_platform_info	C++ return Type
CL_PLATFORM_EXTENSIONS	STRING_CLASS
CL_PLATFORM_NAME	STRING_CLASS
CL_PLATFORM_PROFILE	STRING_CLASS
CL_PLATFORM_VENDOR	STRING_CLASS
CL_PLATFORM_VERSION	STRING_CLASS

Table 1: Differences in cl::Platform::getInfo return type vs. *OpenCL Specification* table 4.1

name is an enumeration constant that identifies the platform information being queried. It can be one of the values specified in table 4.1.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 4.1 is returned. If *param* is NULL, it is ignored.

cl::Platform::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_platform_info, *name*>::param_type
cl::Platform::getInfo (void)

gets specific information about the OpenCL platform. Table 4.1 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_platform_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the platform information being queried. It can be one of the values specified in table 4.1.

cl::Platform::getInfo returns the appropriate value for a given *name* as specified in table 4.1.

The method

cl_int **cl::Platform::getDevices**(cl_device_type *type*,
VECTOR_CLASS<Device> * *devices*)

gets the list of devices available on a platform.

type is a bitfield that identifies OpenCL device type. The *type* can be used to query specific OpenCL devices or all available OpenCL devices. Table 4.2 of the *OpenCL Specification* Version 1.2 specifies the valid values for *type*.

devices returns a vector of OpenCL devices found. *devices* must not be NULL.

cl::Platform::getDevices returns CL_SUCCESS if the method is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_DEVICE_TYPE if *type* is not a valid value.
- CL_INVALID_ARG_VALUE if *devices* is NULL.
- CL_DEVICE_NOT_FOUND if no OpenCL devices matching *type* were found.

2.2 Devices

Class **cl::Device** provides functionality for working with OpenCL devices.

The constructor

cl::Device::Device(cl_device_id *device*)

creates an OpenCL device wrapper for a device.

device is an OpenCL device id.

The method

template <typename *T*>
cl_int **cl::Device::getInfo**(cl_device_info *name*,
T * *param*)

gets specific information about the OpenCL device. Table 4.3 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_device_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_device_info	C++ return Type
CL_DEVICE_BUILT_IN_KERNELS	STRING_CLASS
CL_DEVICE_EXTENSIONS	STRING_CLASS
CL_DEVICE_MAX_WORK_ITEM_SIZES	VECTOR_CLASS<::size_t>
CL_DEVICE_NAME	STRING_CLASS
CL_DEVICE_OPENCL_C_VERSION	STRING_CLASS
CL_DEVICE_PARTITION_PROPERTIES	VECTOR_CLASS<cl_device_partition_property>
CL_DEVICE_PARTITION_TYPE	VECTOR_CLASS<cl_device_partition_property>
CL_DEVICE_PROFILE	STRING_CLASS
CL_DEVICE_VENDOR	STRING_CLASS
CL_DEVICE_VERSION	STRING_CLASS
CL_DRIVER_VERSION	STRING_CLASS

Table 2: Differences in cl::Context::getInfo return type vs. *OpenCL Specification* table 4.3

T is a compile time argument that is the return for the specific information being queried. It corresponds to the values in table 4.3.

name is an enumeration constant that identifies the device information being queried. It can be one of the values specified in table 4.3.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 4.3 is returned. If *param* is NULL, it is ignored.
cl::Device::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_device_info, *name*>::param_type
cl::Device::getInfo(void)

gets specific information about the OpenCL device. Table 4.3 specifies the information that can be queried.

name is an enumeration constant that identifies the device information being queried. It can be one of the values specified in table 4.3.

cl::Device::getInfo returns the appropriate value for a given *name* as specified in table 4.3 in conjunction with the table above.

The method

cl_int **cl::Device::createSubDevices**(const cl_device_partition_property **properties*,
VECTOR_CLASS<Device>* *devices*)

creates an array of sub-devices that each reference a non-intersecting set of compute units of an OpenCL device.

properties specifies how to partition the device. Each property should be an enumeration constant as specified in table 4.4 of the *OpenCL Specification* Version 1.2.

devices is a buffer where the OpenCL sub-devices are returned. *devices* must not be null.

cl::Device::createSubDevices returns CL_SUCCESS if the method is executed successfully. Otherwise, it returns an error code as returned by the underlying OpenCL *clCreateSubDevices* call.

2.3 Contexts

cl
Class **cl::Context** provides functionality for working with OpenCL contexts.

The constructor

cl::Context::Context(VECTOR_CLASS<Device>& *devices*,
cl_context_properties * *properties* = NULL,
void (CL_CALLBACK * *pfn_notify*)(
const char * *errorinfo*,
const void * *private_info*,
::size_t *cb*,
void * *user_data*) = NULL,

void * *user_data* = NULL,
cl_int * err = NULL)

creates an OpenCL context.

devices is a pointer to a vector of unique devices returned by **cl::Platform::getDevices**. If more than one device is specified, a selection criteria may be applied to determine if the list of devices specified can be used together to create a context.

properties specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties is described in table 4.5 of the *OpenCL Specification*. *properties* can be NULL, in which case the platform that is selected is implementation-defined.

pfn_notify is a callback function registered by the application. This callback function is used by the OpenCL implementation to report information on errors that occur in this context. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. The parameters to the callback function are:

- *errinfo* is a pointer to an error string.
- *private_info* and *cb* represent a pointer to binary data returned by the OpenCL implementation that can be used to log additional information helpful in debugging the error.
- *user_data* is a pointer to user supplied data.

If *pfn_notify* is NULL, no callback function is registered.

user_data is passed as the *user_data* argument when *pfn_notify* is called. *user_data* can be NULL.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Context::Context returns a valid object of type **cl::Context** and sets *err* to CL_SUCCESS if it creates the context successfully. Otherwise, it returns one of the following error values in *err*:

- CL_INVALID_PROPERTY if context property name in properties is not a supported property name, if the value specified for a supported property name is not valid, or if the same property name is specified more than once.
- CL_INVALID_VALUE if *devices* is of length zero.
- CL_INVALID_VALUE if *pfn_notify* is NULL but *user_data* is not NULL.
- CL_INVALID_DEVICE if *devices* contains an invalid device.
- CL_DEVICE_NOT_AVAILABLE if a device in *devices* is currently not available even though the device was returned by **cl::Platform::getDevices**.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The constructor

cl::Context::Context(cl_device_type *type*,
cl_context_properties * *properties* = NULL,
void (CL_CALLBACK * *pfn_notify*)(
const char * *errorinfo*,
const void * *private_info*,

::size_t *cb*,
void * *user_data*) = NULL,
void * *user_data* = NULL,
cl_int * *err* = NULL)

creates an OpenCL context from a device type that identifies the specific devices to use. The constructor attempts to use the first platform that has a device of the specified type.

type is a bit-field that identifies the type of device, as described in table 4.2 in section 4.2 of the *OpenCL Specification* Version 1.2.

properties specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties is described in table 4.4. properties can be NULL, in which case the platform that is selected is implementation-defined.

pfn_notify is a callback function registered by the application. This callback function is used by the OpenCL implementation to report information on errors that occur in this context. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. The parameters to the callback function are:

- *errinfo* is a pointer to an error string.
- *private_info* and *cb* represent a pointer to binary data that is returned by the OpenCL implementation that can be used to log additional information helpful in debugging the error.
- *user_data* is a pointer to user supplied data.

If *pfn_notify* is NULL, no callback function is registered.

user_data is passed as the *user_data* argument when *pfn_notify* is called. *user_data* can be NULL.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Context::Context returns a valid object of type **cl::Context** and sets *err* to CL_SUCCESS if it creates the context successfully. Otherwise, it returns one of the following error values in *err*:

- CL_INVALID_PROPERTY if context property name in properties is not a supported property name, if the value specified for a supported property name is not valid, or if the same property name is specified more than once.
- CL_INVALID_VALUE if *pfn_notify* is NULL but user_data is not NULL.
- CL_INVALID_DEVICE_TYPE if *type* is not a valid value.
- CL_DEVICE_NOT_AVAILABLE if no devices that match *type* and property values specified in properties are currently available.
- CL_DEVICE_NOT_FOUND if no devices that match *type* and property values specified in properties were found.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

template <typename *T*>
cl_int **cl::Context::getInfo**(cl_context_info *name*,

*T * param*)

gets specific information about the OpenCL context. Table 4.6 of the *OpenCl Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_context_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_context_info	**C++ return Type**
CL_CONTEXT_DEVICES	VECTOR_CLASS<cl::Device>
CL_CONTEXT_PROPERTIES	VECTOR_CLASS<cl_context_properties>

Table 3: Differences in cl::Context::getInfo return type vs. *OpenCL Specification* table 4.6

T is a compile time argument that is the return type for the specific information being queried. It corresponds to the values in tables 4.6.

name is an enumeration constant that identifies the context information being queried. It can be one of the values specified in table 4.6.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 4.5 is returned. If *param* is NULL, it is ignored.

cl::Context::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_context_info, *name*>::param_type
cl::Context::getInfo(void)

gets specific information about the OpenCL context. Table 4.6 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_context_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the context information being queried. It can be one of the values specified in table 4.6.

cl::Context::getInfo returns the appropriate value for a given *name* as specified in table 4.6.

The method

cl_int **cl::Context::getSupportedImageFormats**(
cl_mem_flags *flags*,
cl_mem_object_type *image_type*,
VECTOR_CLASS<ImageFormat> * *formats*)

can be used to get the list of image formats supported by an OpenCL implementation for the context, when the following information about an image memory object is specified:

- Context
- Image type - 2D, or 3D image.

- Image object allocation information

flags is a bit-field that specifies allocation and usage information about the image memory object being created, as described in table 5.3 of the *OpenCL Specification* Version 1.2.

image_type describes the image type. It must be either CL_MEM_OBJECT_IMAGE2D or CL_MEM_OBJECT_IMAGE3D.

formats is a pointer to a memory location where the vector of supported image formats is returned. Each entry describes a instance of the class **cl::ImageFormat**, which is a mapping for **cl_image_format** structure supported by the OpenCL implementation. If *formats* is NULL, it is ignored.

cl::Context::getSupportedImageFormats returns CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *flags* or *image_type* are not valid.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

3. C++ Runtime layer

3.1 Memory Objects

Class **cl::Memory** provides a base class for working with OpenCL memory objects. It is used to build buffers and images in the following sections.

The method

```
template <typename T>
cl_int cl::Memory::getInfo(cl_context_info name,
                           T * param)
```

gets specific information about the OpenCL memory object. Table 5.11 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_memory_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_memory_info	C++ return Type
CL_MEM_CONTEXT	cl::Context

Table 4: Differences in cl::Memory::getInfo return type vs. *OpenCL Specification* table 5.11

T is a compile time argument that is the return type for the specific information being queried. It corresponds to the values in tables 5.11.

name is an enumeration constant that identifies the context information being queried. It can be one of the values specified in table 5.11.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.11 is returned. If *param* is NULL, it is ignored.

cl::Memory::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

> template <cl_int *name*> typename
> detail::param_traits<detail::cl_context_info, *name*>::param_type
> **cl::Memory::getInfo**(void)

gets specific information about the OpenCL memory object. Table 5.11 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_memory_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the memory object information being queried. It can be one of the values specified in table 5.11.

cl::Memory::getInfo returns the appropriate value for a given *name* as specified in table 5.11.

The method

> cl_int **cl::Memory::setDestructorCallback**(
> void (CL_CALLBACK * *pfn_notify*)(cl_mem *memobj*,
> void * *user_data*),
> void * *user_data* = NULL)

registers a user callback function that is called when the memory object is deleted and its resources freed. The description of **clSetMemObjectDestructorCallback** in section 5.4 of the *OpenCL Specification* gives a detailed overview.

pfn_notify is a callback function registered by the application. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. The parameters to the callback function are:

- *memobj* is the memory object being deleted.
- *user_data* is a pointer to user supplied data.

user_data is passed as the *user_data* argument when *pfn_notify* is called.

cl::Memory::setDestructorCallback returns CL_SUCCESS if executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *pfn_notify* is NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.
-

3.2 Buffers

Class **cl::Buffer : public Memory** provides functionality for working with OpenCL buffers.

The constructor

cl::Buffer::Buffer(
const Context& *context*,
cl_mem_flags *flags*,
::size_t *size*,
void * *host_ptr* = NULL,
cl_int * *err* = NULL)

creates an OpenCL buffer object.

context is a valid OpenCL context used to create the buffer object.

flags is a bit-field that specifies allocation and usage information, such as the memory arena that used to allocate the buffer object and how it is used. Table 5.3 of the *OpenCL Specification* Version 1.2 describes the valid values for *flags*.

size is the size in bytes of the buffer memory object to be allocated.

host_ptr is a pointer to buffer data that may already be allocated by the application. The size of the buffer that *host_ptr* points to must be at least *size* bytes.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

The constructor

cl::Buffer::Buffer(
cl_mem_flags *flags*,
::size_t *size*,
void * *host_ptr* = NULL,
cl_int * *err* = NULL)

creates an OpenCL buffer object in the default context. The default context is constructed via *clCreateContextFromType* with a type of **CL_DEVICE_TYPE_DEFAULT**.

flags is a bit-field that specifies allocation and usage information, such as the memory arena that used to allocate the buffer object and how it is used. Table 5.3 of the *OpenCL Specification* Version 1.2 describes the valid values for *flags*.

size is the size in bytes of the buffer memory object to be allocated.

host_ptr is a pointer to buffer data that may already be allocated by the application. The size of the buffer that *host_ptr* points to must be at least *size* bytes.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

The constructor

```
template <typename IteratorType>
cl::Buffer::Buffer(
    IteratorType startIterator,
    IteratorType endIterator,
    bool readonly ,
    bool useHostPtr = true,
    cl_int * err = NULL)
```

creates an initialized OpenCL buffer object.

The given IteratorType must be a random access iterator. If *useHostPtr* is true, it must use contiguous storage.

startIterator and *endIterator* of the given IteratorType provide start and end iterators for the data source.

readonly specifies whether the buffer object is readonly (**cl_mem_flags** value **CL_MEM_READONLY** or **CL_MEM_READWRITE**).

useHostPtr specifies whether the buffer uses a host pointer (**cl_mem_flags** value **CL_MEM_USE_HOST_PTR**).

The constructor

```
template <typename IteratorType>
cl::Buffer::Buffer(
    const Context &context,
    IteratorType startIterator,
    IteratorType endIterator,
    bool readonly ,
    bool useHostPtr = true,
    cl_int * err = NULL)
```

creates an initialized OpenCL buffer object.

The given IteratorType must be a random access iterator. If *useHostPtr* is true, it must use contiguous storage.

context gives the Context.

startIterator and *endIterator* of the given IteratorType provide start and end iterators for the data source.

readonly specifies whether the buffer object is readonly (**cl_mem_flags** value **CL_MEM_READONLY** or **CL_MEM_READWRITE**).

useHostPtr specifies whether the buffer uses a host pointer (**cl_mem_flags** value **CL_MEM_USE_HOST_PTR**).

If *useHostPtr* is false, the constructor creates a CommandQueue and enqueus a copy of the host memory to the first device in the context. This constructor waits for the copy to complete and then releases the CommandQueue. Note that this constructor is a blocking call in this case.

cl::Buffer::Buffer creates a valid non-zero buffer object and sets *err* to CL_SUCCESS if it creates the buffer object successfully. Otherwise, it returns one of the following error values returned in *err*:

- CL_INVALID_CONTEXT if context is not a valid context.
- CL_INVALID_VALUE if values specified in flags are not valid.
- CL_INVALID_BUFFER_SIZE if size is 0.
- CL_INVALID_HOST_PTR if *host_ptr* is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in flags or if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in flags.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for buffer object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl::Buffer cl::Buffer::createSubBuffer(cl_mem_flags *flags*,
cl_buffer_create_type *buffer_create_type*,
const void * *buffer_create_info*,
cl_int * *err* = NULL)

creates a new buffer object from an existing buffer object.

flags is a bit-field that specifies allocation and usage information about the image memory object being created, as described in table 5.3 of the *OpenCL Specification* Version 1.2.

buffer_create_type and *buffer_create_info* describe the type of buffer object to be created. The list of supported values for *buffer_create_type* and corresponding descriptor for *buffer_create_info* are described in table 5.4 of the *OpenCL Specification* Version 1.2.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Buffer::createSubBuffer returns CL_SUCCESS in *err* on success. It returns an error code in *err* from the underlying OpenCL *clCreateSubBuffer* call if it fails. Otherwise, it returns one of the following errors in *err*:

- CL_INVALID_VALUE if values specified in *flags* are not valid.
- CL_INVALID_VALUE if value specified in *buffer_create_type* is not valid.
- CL_INVALID_VALUE if value specified in *buffer_create_info* (for a given *buffer_create_type*) is not valid or if *buffer_create_info* is NULL.

3.2.1 **Buffer copy**

The method

```
template <typename IteratorType>
cl_int copy(IteratorType startIterator,
            IteratorType endIterator,
            cl::Buffer &buffer)
```

performs a blocking copy to a buffer. The given IteratorType must be a random access iterator.

startIterator and *endIterator* of the given IteratorType provide start and end iterators for the data source.

buffer gives the destination buffer.

The method

```
template <typename IteratorType>
cl_int copy(const cl::Buffer &buffer,
            IteratorType startIterator,
            IteratorType endIterator)
```

performs a blocking copy from a buffer.

buffer gives the source buffer.

startIterator and *endIterator* of the given IteratorType provide start and end iterators for the data destination.

The method

```
template <typename IteratorType>
cl_int copy(const cl::CommandQueue &queue,
            IteratorType startIterator,
            IteratorType endIterator,
            cl::Buffer &buffer)
```

performs a blocking copy to a buffer. The given IteratorType must be a random access iterator.

queue gives the Command Queue to enqueue the copy to.

startIterator and *endIterator* of the given IteratorType provide start and end iterators for the data source.

buffer gives the destination buffer.

The method

```
template <typename IteratorType>
cl_int copy(const cl::CommandQueue &queue,
            const cl::Buffer &buffer,
            IteratorType startIterator,
            IteratorType endIterator)
```

performs a blocking copy from a buffer.

queue gives the Command Queue to enqueue the copy to.

buffer gives the source buffer.

startIterator and *endIterator* of the given IteratorType provide start and end iterators for the data destination.

3.2.2 **BufferGL objects**

Class **cl::BufferGL : public Buffer** provides functionality for OpenGL buffer interoperability.

The constructor

cl::BufferGL::BufferGL(
const Context& *context*,
cl_mem_flags *flags*,
GLuint bufobj,
cl_int * *err* = NULL)

creates an OpenGL-compatible buffer object.

context is a valid OpenCL context used to create the buffer object.

flags is a bit-field. Table 5.3 of the *OpenCL Specification* Version 1.2 describes the valid values for *flags*.

bufobj is an OpenGL buffer handle.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

The method

cl_int **cl::BufferGL::getObjectInfo**(cl_gl_object_type **type*,
GLuint **gl_object_name*)

gets specific information about an OpenGL buffer object.

type returns the type of the GL buffer object, as defined in *OpenCL Specification* Version 1.1, section 9.8.5. If *type* is NULL, it is ignored.

gl_object_name returns the GL object name used to create the OpenGL renderbuffer object. If *gl_object_name* is NULL, it is ignored.

cl::BufferGL::getObjectInfo returns CL_SUCCESS on success.

3.2.3 **BufferRenderGL objects**

Class **cl::BufferRenderGL : public Buffer** provides functionality for OpenGL renderbuffer interoperability.

The constructor

cl::BufferRenderGL::BufferRenderGL(
const Context& *context*,
cl_mem_flags *flags*,
GLuint bufobj,
cl_int * *err* = NULL)

creates an OpenGL-compatible renderbuffer object.

context is a valid OpenCL context used to create the buffer object.

flags is a bit-field. Table 5.3 describes the valid values for *flags*.

bufobj is an OpenGL renderbuffer handle.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

The method

cl_int **cl::BufferRenderGL::getObjectInfo**(cl_gl_object_type **type*,
GLuint **gl_object_name*)

gets specific information about an OpenGL renderbuffer object.

type returns the type of the GL renderbuffer object, as defined in *OpenCL Specification* Version 1.1, section 9.8.5.

gl_object_name returns the GL object name used to create the OpenGL renderbuffer object. If *gl_object_name* is NULL, it is ignored.

cl::BufferRenderGL::getObjectInfo returns CL_SUCCESS on success.

3.3 Images

Class **cl::Image: public Memory** provides a base class for working with OpenCL image objects. It is used to build 1D, 2D, 3D, and array images in the following sections.

The method

template <typename *T*>
cl_int **cl::Image::getImageInfo**(cl_image_info *name*,
*T * param*)

gets specific information about the OpenCL image object. Table 5.9 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried.

T is a compile time argument that is the return for the specific information being queried. It corresponds to the values in table 5.9.

name is an enumeration constant that identifies the context information being queried. It can be one of the values specified in table 5.9.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.9 is returned. If *param* is NULL, it is ignored.

cl::Memory::getImageInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_image_info, *name*>::param_type
cl::Image::getImageInfo(void)

gets specific information about the OpenCL image object. Table 5.9 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried.

name is an enumeration constant that identifies the memory object information being queried. It can be one of the values specified in table 5.9.

cl::Image::getImageInfo returns the appropriate value for a given *name* as specified in table 5.9.

3.3.1 Image1D objects

Class **cl::Image1D : public Image** provides functionality for working with OpenCL 1D images.

The constructor

cl::Image1D::Image1D(Context& *context*,
cl_mem_flags *flags*,
ImageFormat format,
::size_t *width*,
void * host_ptr = NULL,
cl_int * *err* = NULL)

creates an OpenCL 1D image object.

context is a valid OpenCL context on which the image object is to be created.

flags is a bit-field that specifies allocation and usage information about the image object being created. It is described in table 5.3 of the *OpenCL Specification* Version 1.2.

format is a class that describes format properties of the image to be allocated. **cl::ImageFormat** is a mapping for OpenCL image format descriptor structure **cl_image_format**, defined in Section 5.3.1.1 of the *OpenCL Specification* Version 1.2:

```
typedef struct _cl_image_format {
        cl_channel_order    image_channel_order;
        cl_channel_type     image_channel_data_type;
} cl_image_format;
```

width gives the width of the image in pixels. It must be greater than or equal to 1.

host_ptr is a pointer to the image data that may already be allocated by the application. The size of the buffer that *host_ptr* points to must be >= row_pitch * height. The size of each element in bytes must be a power of 2.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Image1D::Image1D returns a valid non-zero image object and sets *err* to CL_SUCCESS if it creates the image object successfully. Otherwise, it returns one of the following error values in *err*:

- CL_INVALID_CONTEXT if context is not a valid context.
- CL_INVALID_VALUE if values specified in flags are not valid.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in *format* are not valid.
- CL_INVALID_IMAGE_SIZE if *width* is 0 or exceeds the value specified in CL_DEVICE_IMAGE2D_MAX_WIDTH for all devices in context or if values specified by *row_pitch* do not follow rules described in the argument description above.
- CL_INVALID_HOST_PTR if *host_ptr* is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in flags or if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in flags.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if the *image_format* is not supported.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for image object.
- CL_INVALID_OPERATION if there are no devices in context that support images (i.e., CL_DEVICE_IMAGE_SUPPORT specified in table 4.3 is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Class **cl::Image1DArray : public Image** provides functionality for working with arrays of OpenCL 1D images.

The constructor

```
cl::Image1DArray::Image1DArray(Context& context,
                               cl_mem_flags  flags,
```

ImageFormat format,
::size_t *array_size*,
::size_t *width*,
::size_t *row_pitch = 0*,
void * host_ptr = NULL,
cl_int * *err* = NULL)

creates an OpenCL 1D image array object.

context is a valid OpenCL context on which the object is to be created.

flags is a bit-field that specifies allocation and usage information about the image memory object being created, as described in table 5.3.

format is a class that describes format properties of the image to be allocated. **cl::ImageFormat** is a mapping for the OpenCL structure **cl_image_format**. Section 5.3.1.1 gives a detailed description of the image format descriptor.

array_size is the number of images in the image array.

width is the width of each image in pixels. This value must be greater than or equal to 1.

row_pitch is the scan-line pitch in bytes. This must be 0 if *host_ptr* is NULL and can be either 0 or >= *width* * size of element in bytes if *host_ptr* is not NULL. If *host_ptr* is not NULL and *row_pitch* = 0, *row_pitch* is calculated as *width* * size of element in bytes. If *row_pitch* is not 0, it must be a multiple of the image element size in bytes.

host_ptr is a pointer to the image data that may already be allocated by the application. The size of the buffer that *host_ptr* points to must be >= *row_pitch* * *array_size*. The size of each element in bytes must be a power of 2. The image data specified by *host_ptr* is stored as a linear sequence of adjacent scanlines. Each scanline is stored as a linear sequence of image elements.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Image1DArray::Image1DArray returns a valid non-zero object and sets *err* to CL_SUCCESS if it creates the image array object successfully.

Class **cl::Image1DBuffer : public Image** provides functionality for working with OpenCL 1D image buffers.

The constructor

cl::Image1DBuffer::Image1DBuffer(Context& *context*,
cl_mem_flags *flags*,
ImageFormat format,
::size_t *width*,
const Buffer &buffer,
cl_int * *err* = NULL)

creates an OpenCL 1D image buffer object.

context is a valid OpenCL context on which the object is to be created.

flags is a bit-field that specifies allocation and usage information about the image memory object being created, as described in table 5.3. See the list of error codes for restrictions on how flags may be set.

format is a class that describes format properties of the image to be allocated. **cl::ImageFormat** is a mapping for the OpenCL structure **cl_image_format**. Section 5.3.1.1 gives a detailed description of the image format descriptor.

width is the width of the image in pixels. This value must be greater than or equal to 1.

buffer refers to a valid buffer memory object.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Image1DArray::Image1DArray returns a valid non-zero object and sets *err* to CL_SUCCESS if it creates the image array object successfully. Otherwise, it returns one of the following error codes:

- CL_INVALID_CONTEXT if context is not a valid context.
- CL_INVALID_VALUE if values specified in flags are not valid.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in image_format are not valid or if image_format is NULL.
- CL_INVALID_IMAGE_DESCRIPTOR if values specified in image_desc are not valid or if image_desc is NULL.
- CL_INVALID_IMAGE_SIZE if image dimensions specified in image_desc exceed the minimum maximum image dimensions described in the table of allowed values for param_name for clGetDeviceInfo for all devices in context.
- CL_INVALID_HOST_PTR if host_ptr in image_desc is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in flags or if host_ptr is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in flags.
- CL_INVALID_VALUE if a 1D image buffer is being created and the buffer object was created with CL_MEM_WRITE_ONLY and flags specifies CL_MEM_READ_WRITE or CL_MEM_READ_ONLY, or if the buffer object was created with CL_MEM_READ_ONLY and flags specifies CL_MEM_READ_WRITE or CL_MEM_WRITE_ONLY, or if flags specifies CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR or CL_MEM_COPY_HOST_PTR.
- CL_INVALID_VALUE if a 1D image buffer is being created and the buffer object was created with CL_MEM_HOST_WRITE_ONLY and flags specifies CL_MEM_HOST_READ_ONLY, or if the buffer object was created with CL_MEM_HOST_READ_ONLY and flags specifies CL_MEM_HOST_WRITE_ONLY, or if the buffer object was created with CL_MEM_HOST_NO_ACCESS and flags specifies CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_WRITE_ONLY.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if the image_format is not supported.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for image object.
- CL_INVALID_OPERATION if there are no devices in context that support images (i.e. CL_DEVICE_IMAGE_SUPPORT (specified in the table of OpenCL Device Queries for clGetDeviceInfo) is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.

- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

3.3.2 Image2D objects

Class **cl::Image2D : public Image** provides functionality for working with OpenCL 2D images.

The constructor

cl::Image2D::Image2D(Context& *context*,
cl_mem_flags *flags*,
ImageFormat format,
::size_t *width*,
::size_t *height*,
::size_t *row_pitch = 0*,
void * host_ptr = NULL,
cl_int * *err* = NULL)

creates an OpenCL 2D image object.

context is a valid OpenCL context on which the image object is to be created.

flags is a bit-field that specifies allocation and usage information about the image memory object being created, as described in table 5.3 of the *OpenCL Specification* Version 1.2.

format is a class that describes format properties of the image to be allocated. **cl::ImageFormat** is a mapping for the OpenCL structure **cl_image_format**. Section 5.3.1.1 gives for a detailed description of the image format descriptor.

width, and *height* are the width and height of the image in pixels. Each must be greater than or equal to 1.

row_pitch is the scan-line pitch in bytes. This must be 0 if *host_ptr* is NULL and can be either 0 or >= *width* * size of element in bytes if *host_ptr* is not NULL. If *host_ptr* is not NULL and *row_pitch* = 0, *row_pitch* is calculated as *width* * size of element in bytes. If *row_pitch* is not 0, it must be a multiple of the image element size in bytes.

host_ptr is a pointer to the image data that may already be allocated by the application. The size of the buffer that *host_ptr* points to must be >= *row_pitch* * height. The size of each element in bytes must be a power of 2. The image data specified by *host_ptr* is stored as a linear sequence of adjacent scanlines. Each scanline is stored as a linear sequence of image elements.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Image2D::Image2D returns a valid non-zero image object and sets *err* to CL_SUCCESS if it creates the image object successfully. Otherwise, it returns one of the following error values in *err*:

- CL_INVALID_CONTEXT if context is not a valid context.
- CL_INVALID_VALUE if values specified in flags are not valid.

- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in *format* are not valid.
- CL_INVALID_IMAGE_SIZE if *width* or *height* are 0 or if they exceed values specified in CL_DEVICE_IMAGE2D_MAX_WIDTH or CL_DEVICE_IMAGE2D_MAX_HEIGHT respectively for all devices in context or if values specified by *row_pitch* do not follow rules described in the argument description above.
- CL_INVALID_HOST_PTR if *host_ptr* is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in flags or if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in flags.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if the *image_format* is not supported.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for image object.
- CL_INVALID_OPERATION if there are no devices in context that support images (i.e., CL_DEVICE_IMAGE_SUPPORT specified in table 4.3 is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Class **cl::Image2DArray : public Image** provides functionality for working with arrays of OpenCL 2D images.

The constructor

cl::Image2DArray::Image2DArray(Context& *context*,
cl_mem_flags *flags*,
ImageFormat format,
::size_t *array_size*,
::size_t *width*,
::size_t *height*,
::size_t *row_pitch = 0*,
::size_t *slice_pitch = 0*,
void * host_ptr = NULL,
cl_int * *err* = NULL)

creates an OpenCL 2D image array object.

context is a valid OpenCL context on which the object is to be created.

flags is a bit-field that specifies allocation and usage information about the image memory object being created, as described in table 5.3.

format is a class that describes format properties of the image to be allocated. **cl::ImageFormat** is a mapping for the OpenCL structure **cl_image_format**. Section 5.3.1.1 gives a detailed description of the image format descriptor.

array_size is the number of images in the image array.

width and *height* are the width and height of each image in pixels. These must be values greater than or equal to 1.

row_pitch is the scan-line pitch in bytes. This must be 0 if *host_ptr* is NULL and can be either 0 or >= *width* * size of element in bytes if *host_ptr* is not NULL. If *host_ptr* is not NULL and *row_pitch* = 0, *row_pitch* is calculated as *width* * size of element in bytes. If *row_pitch* is not 0, it must be a multiple of the image element size in bytes.

slice_pitch gives the size in bytes of each image in the image array. It must be 0 if *host_ptr* is NULL. If *host_ptr* is not NULL, *slice_pitch* can be either 0 or equal to *row_pitch*.

host_ptr is a pointer to the image data that may already be allocated by the application. The size of the buffer that *host_ptr* points to must be >= *slice_pitch* * *array_size*. The size of each element in bytes must be a power of 2. The image data specified by *host_ptr* is stored as a linear sequence of adjacent scanlines. Each scanline is stored as a linear sequence of image elements.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Image2DArray::Image2DArray returns a valid non-zero object and sets *err* to CL_SUCCESS if it creates the image array object successfully.

3.3.3 Image3D objects

Class **cl::Image3D : public Image** provides functionality for working with OpenCL 3D images.

The constructor

cl::Image3D::Image3D(const Context& *context*,
cl_mem_flags *flags*,
ImageFormat *format*,
::size_t *width*,
::size_t *height*,
::size_t *depth*,
::size_t *row_pitch = 0*,
::size_t *slice_pitch* = 0,
void * *host_ptr* = NULL,
cl_int * *err* = NULL)

creates an OpenCL 3D image object.

context is a valid OpenCL context on which the image object is to be created.

flags is a bit-field that specifies allocation and usage information about the image memory object being created, as described in table 5.3.

format is a class that describes format properties of the image to be allocated. **cl::ImageFormat** is a mapping for the OpenCL structure **cl_image_format**. Section 5.3.1.1 gives a detailed description of the image format descriptor.

width, and *height* are the width and height of the image in pixels. Each must be greater than or equal to 1.

depth is the depth of the image in pixels. This must be a value > 1.

row_pitch is the scan-line pitch in bytes. This must be 0 if *host_ptr* is NULL and can be either 0 or >= *width* * size of element in bytes if *host_ptr* is not NULL. If *host_ptr* is not NULL and *row_pitch* = 0, *row_pitch* is calculated as *width* * size of element in bytes. If *row_pitch* is not 0, it must be a multiple of the image element size in bytes.

slice_pitch is the size in bytes of each 2D slice in the 3D image. This must be 0 if *host_ptr* is NULL and can be either 0 or >= *row_pitch* * *height* if *host_ptr* is not NULL. If *host_ptr* is not NULL and *slice_pitch* = 0, *slice_pitch* is calculated as *row_pitch* * *height*. If *slice_pitch* is not 0, it must be a multiple of the *row_pitch*.

host_ptr is a pointer to the image data that may already be allocated by the application. The size of the buffer that *host_ptr* points to must be >= *slice_pitch* * *depth*. The size of each element in bytes must be a power of 2. The image data specified by *host_ptr* is stored as a linear sequence of adjacent 2D slices. Each 2D slice is a linear sequence of adjacent scanlines. Each scanline is a linear sequence of image elements.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Image3D::Image3D returns a valid non-zero image object and sets *err* to CL_SUCCESS if it creates the image object successfully. Otherwise, it returns one of the following error values in *err*:

- CL_INVALID_CONTEXT if context is not a valid context.
- CL_INVALID_VALUE if values specified in flags are not valid.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in *format* are not valid.
- CL_INVALID_IMAGE_SIZE if *width*, *height* are 0 or if *depth* <= 1 or if they exceed values specified in CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_MAX_HEIGHT or CL_DEVICE_IMAGE3D_MAX_DEPTH respectively for all devices in context or if values specified by *row_pitch* and *slice_pitch* do not follow rules described in the argument description above.
- CL_INVALID_HOST_PTR if *host_ptr* is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in flags or if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in flags. CL_IMAGE_FORMAT_NOT_SUPPORTED if the *image_format* is not supported.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for image object.
- CL_INVALID_OPERATION if there are no devices in context that support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in table 4.3 is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

3.3.4 ImageGL objects

Class **cl::ImageGL : public Image** provides functionality for OpenGL image interoperability. This class abstracts all OpenGL image objects (texture buffer objects, 1D texture objects, 1D texture array objects, 2D texture objects, 2D texture array objects, and 3D texture objects).

The constructor

cl::ImageGL::ImageGL(Context& *context*,
cl_mem_flags *flags*,
GLenum *target*,
GLint *miplevel*,
GLuint *texobj*,
cl_int * *err* = NULL)

creates an OpenGL image object.

context is a valid OpenCL context on which the image object is to be created.

flags is a bit-field that specifies allocation and usage information about the image object being created and is described in table 5.3 of the *OpenCL Specification* Version 1.2.

target is an OpenGL texture type, as defined in section 9.8.3 of the *OpenCL Specification* Version 1.1.

miplevel is the mipmap level to be used.

texobj is the name of an OpenGL texture object.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::ImageGL::ImageGL returns a valid non-zero image object and sets *err* to CL_SUCCESS if it creates the image object successfully.

3.4 Samplers

Class **cl::Sampler** provides functionality for working with OpenCL samplers.

The constructor

cl::Sampler::Sampler(const Context& *context*,
cl_bool *normalized_coords*,
cl_addressing_mode *addressing_mode*,
cl_filter_mode *filter_mode*,
cl_int * *err* = NULL)

creates an OpenCL sampler object. Refer to section 6.11.13.1 for a detailed description of how samplers work.

context must be a valid OpenCL context.

normalized_coords determines if the image coordinates specified are normalized (if *normalized_coords* is CL_TRUE) or not (if *normalized_coords* is CL_FALSE).

addressing_mode specifies how out-of-range image coordinates are handled when reading from an image. It can be CL_ADDRESS_MIRRORED_REPEAT, CL_ADDRESS_REPEAT, CL_ADDRESS_CLAMP_TO_EDGE, CL_ADDRESS_CLAMP, or CL_ADDRESS_NONE.

filtering_mode specifies the type of filter that must be applied when reading an image. It can be CL_FILTER_NEAREST, or CL_FILTER_LINEAR.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Sampler::Sampler constructs a valid non-zero sampler object and sets *err* to CL_SUCCESS if it creates the sampler object successfully. Otherwise, it returns one of the following error values in *err*:

- CL_INVALID_CONTEXT if context is not a valid context.
- CL_INVALID_VALUE if an argument value for *addressing_mode*, *filter_mode*, or *normalized_coords* is not valid.
- CL_INVALID_OPERATION if images are not supported by any device associated with *context* (i.e., CL_DEVICE_IMAGE_SUPPORT specified in table 4.3 is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

template <typename *T*>
cl_int **cl::Sampler::getInfo**(cl_sampler_info name,
*T * param*)

gets specific information about the OpenCL Sampler. Table 5.12 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_sampler_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_sampler_info	**C++ return Type**
CL_SAMPLER_CONTEXT	cl::Context

Table 5: Differences in cl::Sampler::getInfo return type vs. *OpenCL Specification* table 5.12

T is a compile time argument that is the return for the specific information being queried. It corresponds to the values in tables 5.12.

name is an enumeration constant that identifies the sampler information being queried. It can be one of the values specified in table 5.12.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.12 is returned.

If *param* is NULL, it is ignored.

cl::Sampler::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_sampler_info, *name*>::param_type
cl::Sampler::getInfo(void)

gets specific information about the OpenCL sampler. Table 5.12 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_sampler_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the sampler information being queried. It can be one of the values specified in table 5.12.

cl::Sampler::getInfo returns the appropriate value for a given *name* as specified in table 5.12.

3.5 Programs

Class **cl::Program** provides functionality for working with OpenCL programs.

Class **cl::Program** provides two public typedefs for working with source files and binaries, respectively

typedef VECTOR_CLASS<std::pair<const void*, ::size_t> > **Binaries**

and

typedef VECTOR_CLASS<std::pair<const char*, ::size_t> > **Sources**

The constructor

cl::Program::Program(const STRING_CLASS& *source*,
bool *build* = false,
cl_int * *err* = NULL)

creates an OpenCL program object.

source is the program source code.

bool is a flag that indicates whether to build the program.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

The constructor

cl::Program::Program(const Context& *context*,
const STRING_CLASS& *source*,
bool *build*,
cl_int * *err* = NULL)

creates an OpenCL program object for an OpenCL context.

context must be a valid OpenCL context.

source is the program source code.

bool is a flag that indicates whether to build the program.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

The constructor

cl::Program::Program(const Context& *context*,
const Sources& sources,
cl_int * *err* = NULL)

creates an OpenCL program object for a context and loads the source code specified by the text strings in each element of the vector *sources* into the program object.

context must be a valid OpenCL context.

sources is a vector of source/size tuples that make up the source code.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Program::Program returns a valid program object and sets *err* to CL_SUCCESS if it creates the program object successfully. Otherwise, it returns one of the following error values returned in *err*:

- CL_INVALID_CONTEXT if context is not a valid context.
- CL_INVALID_VALUE if *sources* contains zero entries or if any entry in *sources* contains a tuple with NULL. CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The constructor

cl::Program::Program(const Context& *context*,
const VECTOR_CLASS<Device>& *devices*,
const Binaries& *binaries*,
VECTOR_CLASS<cl_int> * *binaryStatus* = NULL,
cl_int * *err* = NULL)

creates an OpenCL program object for a context and loads the binary bits specified by the binary in each element of the vector *binaries* into the program object.

context must be a valid OpenCL context.

devices is a list of devices. *devices* must be of non-zero length and each device specified by *devices* must be associated with *context*. The *binaries* are loaded for devices specified in this list. The devices associated with the program object will be the given *devices*.

binaries is a vector of program binaries to be loaded for devices specified by *devices*. For each *devices*[i], the program binary for that device is *binaries*[i].

binary_status returns whether the program binary for each device specified in *devices* was loaded successfully. If *binary_status* is NULL, it is ignored. If non-NULL, *binary_status* will be resized to match the length of *devices*. *binary_status*[i] returns CL_SUCCESS if the binary for *devices*[i] is successfully loaded or CL_INVALID_BINARY if program *binaries*[i] is NULL.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::Program::Program returns a valid program object and sets *err* to CL_SUCCESS if it creates the program object successfully. Otherwise, it returns one of the following error values returned in *err*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if *devices* is of length zero or if the length of *binaries* is not equal to the length of *devices* or if any entry in *binaries* is not valid..
- CL_INVALID_DEVICE if OpenCL devices listed in *devices* are not in the list of devices associated with context.
- CL_INVALID_BINARY if an invalid program binary was encountered for any device. *binaryStatus* returns specific status for each device.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The constructor

cl::Program::Program(const Context& *context*,
const VECTOR_CLASS<Device>& *devices*,
STRING_CLASS& *kernelNames*,
cl_int * *err* = NULL)

creates an OpenCL program object with a set of built-in kernel names for a context.

context must be a valid OpenCL context.

devices is a vector list of devices that are in *context*. *devices* must be of non-zero length and each device specified by *devices* must be associated with *context*. The binaries are loaded for devices specified in this list. The devices associated with the program object will be the given *devices.*

kernelNames is a semi-colon separated list of built-in kernel names.

cl::Program::Program returns a valid program object and sets *err* to CL_SUCCESS if it creates the program object successfully.

The method

cl_int **cl::Program::build**(const VECTOR_CLASS<Device>& devices,

const char * *options* = NULL,
(CL_CALLBACK * pfn_notify)
(cl_program,
void * *user_data*) = NULL,
void * *data* = NULL)

builds (compiles and links) a program executable from the program source or binary for all the devices or specific devices in the OpenCL context associated with program.

devices is a list of devices associated with program. If *devices* is of length zero, the program executable is built for all devices associated with program for which a source or binary has been loaded. If *devices* is of non-zero length, the program executable is built for devices specified in the list for which a source or binary has been loaded.

options is a pointer to a string that describes the build options to be used for building the program executable. Section 5.6.4 of the *OpenCl Specification* Version 1.2 describes the supported options.

pfn_notify is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which is called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not NULL, **cl::Program::build** does not need to wait for the build to complete and can return immediately. If *pfn_notify* is NULL, **cl::Program::build** does not return until the build has completed. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe.

data is passed as an argument when *pfn_notify* is called. *data* can be NULL.

cl::Program::build returns CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *pfn_notify* is NULL but *data* is not NULL.
- CL_INVALID_DEVICE if OpenCL devices listed in *devices* are not in the list of devices associated with program
- CL_INVALID_BINARY if program is created with the **cl::Program::Program** taking a list of binaries and devices listed in *devices* do not have a valid program binary loaded.
- CL_INVALID_BUILD_OPTIONS if the build options specified by options are invalid.
- CL_INVALID_OPERATION if the build of a program executable for any of the devices listed in *devices* by a previous call to **cl::Program::build** for program has not completed.
- CL_COMPILER_NOT_AVAILABLE if program is created from source and a compiler is not available, i.e., CL_DEVICE_COMPILER_AVAILABLE specified in table 4.3 is set to CL_FALSE.
- CL_BUILD_PROGRAM_FAILURE if there is a failure to build the program executable. This error is returned if **cl::Program::build** does not return until the build has completed.
- CL_INVALID_OPERATION if there are kernel objects attached to program.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::Program::build(**const char * *options* = NULL,
(CL_CALLBACK * pfn_notify)

(cl_program,
void * *user_data*) = NULL,
void * *data* = NULL)

builds (compiles and links) a program executable from the program source or binary in the OpenCL context associated with program.

options is a pointer to a string that describes the build options to be used for building the program executable. The list of supported options is described in section 5.6.4 of the *OpenCL Specification* Version 1.2.

pfn_notify is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which is called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not NULL, **cl::Program::build** does not need to wait for the build to complete and can return immediately. If *pfn_notify* is NULL, **cl::Program::build** does not return until the build has completed. This callback function may be called asynchronously by the OpenCL implementation. The application must ensure that the callback function is thread-safe.

data is passed as an argument when *pfn_notify* is called. *data* can be NULL.

cl::Program::build returns CL_SUCCESS on success.

The method

cl_int **cl::Program::compile(**const char * *options* = NULL,
(CL_CALLBACK * pfn_notify)
(cl_program,
void * *user_data*) = NULL,
void * *data* = NULL)

compiles (but does not link) a program executable from the program source or binary in the OpenCL context associated with program.

options is a pointer to a string that describes the compilation options to be used for building the program executable. Section 5.6.4 of the *OpenCL Specification*Version 1.2 describes the list of supported options.

pfn_notify is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which is called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not NULL, **cl::Program::compile** does not need to wait for the build to complete and can return immediately. If *pfn_notify* is NULL, **cl::Program::compile** does not return until the build has completed. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe.

data is passed as an argument when *pfn_notify* is called. *data* can be NULL.

cl::Program::compile returns CL_SUCCESS on success.

The function

cl::Program **linkProgram(**Program *input1,*
Program *input2,*

const char * *options* = NULL,
(CL_CALLBACK * pfn_notify)
(cl_program,
void * *user_data*) = NULL,
void * *data* = NULL,
cl_int **err* = NULL)

links programs *input1* and *input2*.

input1 and *input2* are program objects.

options is a pointer to a string that describes the compilation options to be used for linking the program executable. The list of supported options is described in section 5.6.4 of the *OpenCL Specification 1.2*.

pfn_notify is a function pointer to a notification routine. The notification routine is a callback function registered by the application which is called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not NULL, **cl::Program::linkProgram** does not need to wait for the build to complete and can return immediately. If *pfn_notify* is NULL, **cl::Program::linkProgram** does not return until the build has completed. This callback function may be called asynchronously by the OpenCL implementation. The application must ensure that the callback function is thread-safe.

data is passed as an argument when *pfn_notify* is called. *data* can be NULL.

linkProgram returns CL_SUCCESS if executed successfully.

The function

cl::Program **linkProgram(**VECTOR_CLASS<Program> *InputPrograms*,
const char * *options* = NULL,
(CL_CALLBACK * pfn_notify)
(cl_program,
void * *user_data*) = NULL,
void * *data* = NULL,
cl_int **err* = NULL)

links a set of programs.

InputPrograms is a set of program objects.

options is a pointer to a string that describes the compilation options to be used for linking the program executable. The list of supported options is described in section 5.6.4 of the *OpenCL Specification* Version 1.2.

pfn_notify is a function pointer to a notification routine. The notification routine is a callback function registered by the application which is called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not NULL, **cl::Program::linkProgram** does not need to wait for the build to complete and returns immediately. If *pfn_notify* is NULL, **cl::Program::linkProgram** does not return until the build has completed. This callback function may be called asynchronously by the OpenCL implementation. The application must ensure that the callback function is thread-safe.

data is passed as an argument when *pfn_notify* is called. *data* can be NULL.

linkProgram returns CL_SUCCESS if executed successfully.

The method

template <typename *T*>
cl_int **cl::Program::getInfo**(cl_program_info *name*,
T * *param*)

gets specific information about the OpenCL Program. Table 5.13 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_program_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_program_info	**C++ return Type**
CL_PROGRAM_BINARIES	VECTOR_CLASS<char *>
CL_PROGRAM_BINARY_SIZES	VECTOR_CLASS<::size_t>
CL_PROGRAM_CONTEXT	cl::Context
CL_PROGRAM_DEVICES	VECTOR_CLASS<cl::Device>
CL_PROGRAM_KERNEL_NAMES	STRING_CLASS
CL_PROGRAM_SOURCE	STRING_CLASS

Table 6: Differences in cl::Program::getInfo return type vs. *OpenCL Specification* table 5.13

T is a compile time argument that is the return for the specific information being queried. It corresponds to the values in table 5.13.

name is an enumeration constant that identifies the program information being queried. It can be one of the values specified in table 5.13.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.13 is returned. If *param* is NULL, it is ignored.

cl::Program::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_program_info, *name*>::param_type
cl::Program::getInfo(void)

gets specific information about the OpenCL program. Table 5.13 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_program_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the program information being queried. It can be one of the values specified in table 5.13.

cl::Program::getInfo returns the appropriate value for a given *name* as specified in table 5.13.

The method

template <typename *T*>
cl_int **cl::Program::getBuildInfo**(cl_program_build_info *name*,
T * *param*)

returns build information for each device in the program object. Table 5.14 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_program_build_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_program_build_info	**C++ return Type**
CL_PROGRAM_BUILD_LOG	STRING_CLASS
CL_PROGRAM_BUILD_OPTIONS	STRING_CLASS

Table 7: Differences in cl::Program::getBuildInfo return type vs. *OpenCL Specification* table 5.14

T is a compile time argument that is the return type for the specific information being queried. It corresponds to the values in tables 5.14.

name is an enumeration constant that identifies the program build information being queried. It can be one of the values specified in table 5.14.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.14 is returned. If *param* is NULL, it is ignored.

cl::Program::getBuildInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int name> typename
detail::param_traits<detail::cl_program_info, *name*>::param_type
cl::Program::getBuildInfo(void)

returns build information for each device in the program object. Table 5.14 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_program_build_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the program information being queried. It can be one of the values specified in table 5.14.

cl::Program::getBuildInfo returns the appropriate value for a given *name* as specified in table 5.14.

The method

cl_int **cl::Program::createKernels**(const VECTOR_CLASS<Kernel> * *kernels*)

creates kernel objects (i.e., objects of type **cl::Kernel**, see section 3.6 below) for all kernels in the program.

kernels is a pointer to a vector where the kernel objects for *kernels* in the program are returned.

c::Program::createKernels returns CL_SUCCESS if the kernel objects were successfully allocated. Otherwise, it returns one of the following errors:

- CL_INVALID_PROGRAM_EXECUTABLE if there is no successfully built executable for any device in program.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

3.6 Kernels

Class **cl::Kernel** provides functionality for working with OpenCL kernels.

The constructor

cl::Kernel::Kernel(const Program& *program*,
const char * *name*,
cl_int * *err* = NULL)

creates a kernel object.

program is a program object with a successfully built executable.

name is a function name in the program declared with the **__kernel** qualifier.

err returns an appropriate error code. If *err* is NULL, no error code is returned.
cl::Kernel::Kernel returns a valid kernel object and sets *err* to CL_SUCCESS if it creates the kernel object successfully. Otherwise, it returns one of the following error values returned in *err*:

- CL_INVALID_PROGRAM if program is not a valid program object.
- CL_INVALID_PROGRAM_EXECUTABLE if there is no successfully built executable for program.
- CL_INVALID_KERNEL_NAME if *name* is not found in program.
- CL_INVALID_KERNEL_DEFINITION if the function definition for __kernel function given by *name* such as the number of arguments, the argument types are not the same for all devices for which the program executable has been built.
- CL_INVALID_VALUE if *name* is NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

template <typename *T*>
cl_int **cl::Kernel::getInfo**(cl_kernel_info *name*,
*T * param*)

gets specific information about the OpenCL kernel. Table 5.15 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_kernel_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_kernel_info	**C++ return Type**
CL_KERNEL_ATTRIBUTES	STRING_CLASS
CL_KERNEL_CONTEXT	cl::Context
CL_KERNEL_FUNCTION_NAME	STRING_CLASS
CL_KERNEL_PROGRAM	cl::Program

Table 8: Differences in cl::Kernel::getInfo return type vs. *OpenCL Specification* table 5.15

T is a compile time argument that is the return type for the specific information being queried. It corresponds to the values in table 5.15.

name is an enumeration constant that identifies the kernel information being queried. It can be one of the values specified in table 5.15.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.15 is returned. If *param* is NULL, it is ignored.

cl::Kernel::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int name> typename
detail::param_traits<detail::cl_kernel_info, *name*>::param_type
cl::Kernel::getInfo(void)

gets specific information about the OpenCL kernel. Table 5.15 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_kernel_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the kernel information being queried. It can be one of the values specified in table 5.15.

cl::Kernel::getInfo returns the appropriate value for a given *name* as specified in table 5.15.

The method

template <typename *T*>
cl_int **cl::Kernel::getArgInfo**(cl_uint *argIndex*,
cl_kernel_arg_info *name*,
T * param)

gets specific information about an OpenCL kernel argument. Table 5.17 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_kernel_arg_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_kernel_arg_info	C++ return Type
CL_KERNEL_ARG_NAME	STRING_CLASS
CL_KERNEL_ARG_TYPE_NAME	STRING_CLASS

Table 9: Differences in cl::Kernel::getArgInfo return type vs. *OpenCL Specification* table 5.17

T is a compile time argument that is the return type for the specific information being queried. It corresponds to the values in table 5.15.

argIndex is the index of the kernel argument being queried.

name is an enumeration constant that identifies the kernel argument information being queried. It can be one of the values specified in table 5.17.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.17 is returned. If *param* is NULL, it is ignored.

cl::Kernel::getArgInfo returns CL_SUCCESS on success.

The method

template <cl_int name> typename
detail::param_traits<detail::cl_kernel_arg_info, *name*>::param_type
cl::Kernel::getArgInfo(xl_uint *argIndex*, cl_int* *err*)

gets specific information about an OpenCL kernel argument. Table 5.15 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_kernel_arg_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the kernel information being queried. It can be one of the values specified in table 5.17.

argIndex is the index of the kernel argument being queried.

cl::Kernel::getArgInfo returns the appropriate value for a given *name* as specified in table 5.17.

The method

template <typename *T*>
cl_int **cl::Kernel::getWorkGroupInfo**(cl_kernel_work_group_info *name*,
*T * param*)

gets specific information about the OpenCL kernel object that may be specifc to a device. Table 5.16 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_kernel_work_group_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_kernel_work_group_info	C++ return Type

CL_KERNEL_COMPILE_WORK_GROUP_SIZE	cl::size_t<3>[2]

Table 10: Differences in cl::Kernel::getWorkGroupInfo return type vs. *OpenCL Specification* table 5.16

T is a compile time argument that is the return type for the specific information being queried. It corresponds to the values in tables 5.16.

name is an enumeration constant that identifies the kernel information being queried. It can be one of the values specified in table 5.16.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.16 is returned. If *param* is NULL, it is ignored.

cl::Kernel::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_kernel_work_group_info, *name*>::param_type
cl::Kernel::getWorkGroupInfo(void)

gets specific information about the OpenCL kernel object that may be specific to a device. Table 5.16 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_kernel_work_group_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the kernel information being queried. It can be one of the values specified in table 5.16.

cl::Kernel::getWorkGroupInfo returns the appropriate value for a given *name* as specified in table 5.16.

The method

template <typename *T*>
cl_int **cl::Kernel::setArg**(cl_uint *index*,
T value)

sets the argument value for a specific argument of a kernel.

T is a compile time argument that determines the type of a kernel argument being set. It can be one of the following:

- A **cl::Memory** object. e.g. a **cl::Buffer** or **cl::Image3D** would be possible values.
- A **cl::Sampler** object.
- A value of type **cl::LocalSpaceArg**[3], which corresponds to an argument of **__local** in the kernel object.

[2] cl::size_t<3> is a internal type that can be treated as a 3D array whose components correspond to x,y,z values of the work-group size.

- A constant value that is passed by value to the kernel.

index is the argument index. Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to n - 1, where n is the total number of arguments declared by a kernel.

value is the data that should be used aas rthe argument value for argument specified by *index*.

cl::Kernel::setArg returns CL_SUCCESS if the function was executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_ARG_INDEX if *index* is not a valid argument index.
- CL_INVALID_MEM_OBJECT for an argument declared to be a memory object when the specified *value* is not a valid memory object.
- CL_INVALID_SAMPLER for an argument declared to be of type *cl::Sampler* when the specified *value* is not a valid sampler object.

3.6.1 Kernel functors

Kernel functors provide additional functionality to simplify kernel invocations. The template

template <typename *T0*, typename *T1* = detail::NullType, typename *T2* = detail::NullType,
typename *T3* = detail::NullType, typename *T4* = detail::NullType, typename *T5* = detail::NullType,
typename *T6* = detail::NullType, typename *T7* = detail::NullType, typename *T8* = detail::NullType,
typename *T9* = detail::NullType, typename *T10* = detail::NullType, typename *T11* = detail::NullType,
typename *T12* = detail::NullType, typename *T13* = detail::NullType, typename *T14* = detail::NullType,
typename *T15* = detail::NullType, typename *T16* = detail::NullType, typename *T17* = detail::NullType,
typename *T18* = detail::NullType, typename *T19* = detail::NullType, typename *T20* = detail::NullType,
typename *T21* = detail::NullType, typename *T22* = detail::NullType, typename *T23* = detail::NullType,
typename *T24* = detail::NullType, typename *T25* = detail::NullType, typename *T26* = detail::NullType,
typename *T27* = detail::NullType, typename *T28* = detail::NullType, typename *T29* = detail::NullType,
typename *T30* = detail::NullType, typename *T31* = detail::NullType>
struct make_kernel :: detail::functionImplementation<*T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T22, T23, T24, T25, T26, T27, T28, T29, T30, T31*>
cl::make_kernel::make_kernel(
const Program &*program*,
const STRING_CLASS *name*,
cl_int **err* = NULL)

makes a kernel functor for a kernel with 1 to 32 arguments.

T0 to *T31* are the kernel argument types.

program is the OpenCL program that defines the kernel.

name is the name of the kernel functor.

---

[3] The function cl::LocalSpaceArg cl::**Local**(::size_t) can be used to construct arguments specifying the size of a **Local** kernel argument. For example, cl::**Local**(100) would allocate sizeof(cl_char) * 100 of local memory.

err returns an appropriate error code. If *err* is NULL, no error is returned.

The template

template <typename *T0*, typename *T1* = detail::NullType, typename *T2* = detail::NullType,
typename *T3* = detail::NullType, typename *T4* = detail::NullType, typename *T5* = detail::NullType,
typename *T6* = detail::NullType, typename *T7* = detail::NullType, typename *T8* = detail::NullType,
typename *T9* = detail::NullType, typename *T10* = detail::NullType, typename *T11* = detail::NullType,
typename *T12* = detail::NullType, typename *T13* = detail::NullType, typename *T14* = detail::NullType,
typename *T15* = detail::NullType, typename *T16* = detail::NullType, typename *T17* = detail::NullType,
typename *T18* = detail::NullType, typename *T19* = detail::NullType, typename *T20* = detail::NullType,
typename *T21* = detail::NullType, typename *T22* = detail::NullType, typename *T23* = detail::NullType,
typename *T24* = detail::NullType, typename *T25* = detail::NullType, typename *T26* = detail::NullType,
typename *T27* = detail::NullType, typename *T28* = detail::NullType, typename *T29* = detail::NullType,
typename *T30* = detail::NullType, typename *T31* = detail::NullType>
struct make_kernel :: detail::functionImplementation<*T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10,*
T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T22, T23, T24, T25, T26, T27, T28, T29, T30, T31>
cl::make_kernel::make_kernel(
const Kernel *kernel*,
cl_int **err* = NULL)

makes a kernel functor for the given *kernel* with 1 to 32 arguments.

T0 to *T31* are the kernel argument types.

err returns an appropriate error code. If *err* is NULL, no error is returned.
The overloaded operator () for a kernel functor invocation takes one of the following forms:

Event operator() (EnqueueArgs& *args*,
T0 t0, *T1 t1* = NullType, …, *T31 t31* = NullType)
Event operator() (EnqueueArgs& *args*,
const Event& *waitEvent*,
T0 t0, *T1 t1* = NullType, …, *T31 t31* = NullType)
Event operator() (EnqueueArgs &*args*,
const VECTOR_CLASS<Event>& *waitEvents*,
T0 t0, *T1 t1* = NullType, …, *T31 t31* = NullType)

The first enqueues the kernel's arguments, invokes the kernel, and returns an event object representing the kernel execution. The second and third forms are similar, but the second waits for the completion of the given *waitEvent* before invoking the kernel, while the third waits for completion of all *waitEvents* before invoking the kernel.

For example, suppose myKernelProgram is a Program with a string defining the source for kernel myKernel, and suppose myKernel takes three Buffer arguments (two input buffers and an output buffer). Then:

typedef cl::make_kernel <cl::Buffer&, cl::Buffer&, cl::Buffer&> KernelType;
std::function<KernelType::type_> myKernel = KernelType(myKernelProgram, "myKernel");

constructs kernel functor myKernel. Similarly, using auto (for C++11 users):

```
auto myKernel = cl::make_kernel<cl::Buffer&, cl::Buffer&, cl::Buffer&> (myKernelProgram,
                                                                        "myKernel");
```

To invoke the kernel with enqueued input buffers inBuf1 and inBuf2 of size n and output buffer outBuf:

```
myKernel(cl::EnqueueArgs(cl::NDRange(n), cl::NDRange(n)), inBuf1, inBuf2, outBuf);
```

Class **EnqueueArgs** is described in the next subsection below.

Due to the way the parameter state of the underlying kernel objects is defined, functors that reference the same kernel object must not be called concurrently.

3.6.2 EnqueueArgs

Class **cl::EnqueueArgs** parameterizes argument dispatch. Its constructors, listed below, allow for orthogonal overloading of dispatch parameters and parameter count for functors. If a single event is passed, **EnqueueArgs** constructs a list of one event for the enquque. If a vector of events is passed, it constructs a list of input event dependencies. Argument dispatch occurs through the default queue or through a specified queue.

The constructors for **EnqueueArgs** are:

cl::EnqueueArgs::EnqueueArgs(NDRange *global***)**
cl::EnqueueArgs::EnqueueArgs(NDRange *global***, NDRange** *local***)**
cl::EnqueueArgs::EnqueueArgs(NDRange *offset***, NDRange** *global***, NDRange** *local***)**

cl::EnqueueArgs::EnqueueArgs(Event *e***, NDRange** *global***)**
cl::EnqueueArgs::EnqueueArgs(Event *e***, NDRange** *global***, NDRange** *local***)**
cl::EnqueueArgs::EnqueueArgs(Event *e***, NDRange** *offset***, NDRange** *global***, NDRange** *local***)**

cl::EnqueueArgs::EnqueueArgs(const VECTOR_CLASS<Event> &*events***, NDRange** *global***)**
cl::EnqueueArgs::EnqueueArgs(const VECTOR_CLASS<Event> &*events***, NDRange** *global***,**
NDRange *local***)**
cl::EnqueueArgs::EnqueueArgs(const VECTOR_CLASS<Event> &*events***, NDRange** *offset***,**
NDRange *global***, NDRange** *local***)**

cl::EnqueueArgs::EnqueueArgs(CommandQueue &*queue***, NDRange** *global***)**
cl::EnqueueArgs::EnqueueArgs(CommandQueue &*queue***, NDRange** *global***, NDRange** *local***)**
cl::EnqueueArgs::EnqueueArgs(CommandQueue &*queue***, NDRange** *offset***, NDRange** *global***,**
NDRange *local***)**

cl::EnqueueArgs::EnqueueArgs(CommandQueue &*queue***, Event** *e***, NDRange** *global***)**
cl::EnqueueArgs::EnqueueArgs(CommandQueue &*queue***, Event** *e***, NDRange** *global***,**
NDRange *local***)**
cl::EnqueueArgs::EnqueueArgs(CommandQueue &*queue***, Event** *e***, NDRange** *offset***,**
NDRange *global***, NDRange** *local***)**

cl::EnqueueArgs::EnqueueArgs(CommandQueue &*queue***,**

const VECTOR_CLASS<Event> &*events***, NDRange** *global***)**
cl::EnqueueArgs::EnqueueArgs(CommandQueue &*queue***,**
const VECTOR_CLASS<Event> &*events***, NDRange** *global***, NDRange** *local***)**
cl::EnqueueArgs::EnqueueArgs(CommandQueue &*queue***,**
const VECTOR_CLASS<Event> &*events***, NDRange** *offset***, NDRange** *global***,**
NDRange *local***)**

global is a global work size corresponding to the *global_work_size* argument of the underlying OpenCL **EnqueueNDRangeKernel** call.

local is a local work size corresponding to the *local_work_size* argument of the underlying OpenCL **EnqueueNDRangeKernel** call. If *local* is not specified, a NULL *local_work_size* is used.

offset is an offset corresponding to the *global_work_offset* argument of the underlying OpenCL **EnqueueNDRangeKernel** call. If *offset* is not specified, a NULL *global_work_offset* is used.

e is an **Event** that must be completed before the **EnqueueArgs** may be executed, and similarly *events* is a list of events that must be completed before the **EnqueueArgs** may be executed. If neither *e* nor *events* is specified, the **EnqueueArgs** is executed without waiting on any events.

queue is a **CommandQueue** to which the **EnqueueArgs** is submitted. If *queue* is not specified, **EnqueueArgs** is submitted to the default queue.

3.7 Events

Class **cl::Event** provides functionality for working with OpenCL events.

The method

template <typename *T*>
cl_int **cl::Event::getInfo**(cl_event_info *name*,
T * param)

gets specific information about the OpenCL event. Table 5.18 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_event_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_event_info	**C++ return Type**
CL_EVENT_CONTEXT	cl::Context
CL_EVENT_COMMAND_QUEUE	cl::CommandQueue

Table 11: Differences in cl::Event::getInfo return type vs. *OpenCL Specification* table 5.18

T is a compile time argument that is the return type for the specific information being queried. It corresponds to the values in tables 5.18.

name is an enumeration constant that identifies the event information being queried. It can be one of the values specified in table 5.18.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.18 is returned. If *param* is NULL, it is ignored.

cl::Event::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_event_info, *name*>::param_type
cl::Event::getInfo(void)

gets specific information about an OpenCL event. Table 5.18 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_event_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the event information being queried. It can be one of the values specified in table 5.18.

cl::Event::getInfo returns the appropriate value for a given *name* as specified in table 5.18.

The method

template <typename *T*>
cl_int **cl::Event::getProfilingInfo**(cl_profiling_info *name*,
T * *param*)

returns profiling information for the command associated with event. Table 5.19 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried.

T is a compile time argument that is the return type for the specific information being queried. It corresponds to the values in tables 5.19.

name is an enumeration constant that identifies the profiling information being queried. It can be one of the values specified in table 5.19.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.19 is returned. If *param* is NULL, it is ignored.

cl::Event::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_profiling_info, *name*>::param_type
cl::Event::getProfilingInfo(void)

returns profiling information for the command associated with event. Table 5.19 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried.

name is an enumeration constant that identifies the profiling information being queried. It can be one of the values specified in table 5.19.

cl::Event::getProfilingInfo returns the appropriate value for a given *name* as specified in table 5.19.

The method

cl_int **cl::Event::wait**(void)

waits on the host thread for the command associated with the particular event to complete.

cl::Event::wait returns CL_SUCCESS if the function was executed successfully. Otherwise, it returns one of the following errors:

- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::Event::setCallback**(cl_int type,
void (CL_CALLBACK * *pfn_notify*)
(cl_event *event*,
cl_int *command_exec_status*,
void * user_data),
void * *user_data* = NULL)

registers a user callback function for a specific command execution status. The registered callback function is called when the execution status of command associated with event changes to the execution status specified by *command_exec_status*.

type specifies the command execution status for which the callback is registered. The command execution callback mask values for which a callback can be registered are: CL_COMPLETE. There is no guarantee that the callback functions registered for various execution status values for an event is called in the exact order that the execution status of a command changes.

pfn_notify is the event callback function registered by the application. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. The parameters to the callback function are:

- *event* is the event object for which the callback function is invoked.
- *command_exec_status* represents the execution status of command for which this callback function is invoked. Refer to table 5.15 for the command execution status values. If the callback is called as the result of the command associated with event being abnormally terminated, an appropriate error code for the error that caused the termination is passed to *command_exec_status* instead.
- *user_data* is a pointer to user supplied data.

user_data is passed as the *user_data* argument when *pfn_notify* is called. *user_data* can be NULL.

cl:Event::setCallback returns CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *pfn_notify* is NULL or if *command_exec_callback_type* is not a valid command execution status.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The static method

static cl_int **cl::Event::waitForEvents**(const VECTOR_CLASS<Event>& *events*)

waits on the host thread for commands identified by event objects in *events* to complete. A command is considered complete if its execution status is CL_COMPLETE or a negative value. The events specified in *events* act as synchronization points.

cl::Event::waitForEvents returns CL_SUCCESS if the function was executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *events* is of length zero.
- CL_INVALID_CONTEXT if events specified in *events* do not belong to the same context.
- CL_INVALID_EVENT if event objects specified in *events* are not valid event objects.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

3.8 User Events

Class **cl::UserEvent : public Event** provides functionality for working with OpenCL user events.

The constructor

cl::UserEvent::UserEvent(Context& *context*,
cl_int * *err* = NULL)

creates a user event object. User events allow applications to enqueue commands that wait on a user event to finish before the command is executed by the device.

context must be a valid OpenCL context.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::UserEvent::UserEvent returns a valid object and sets *err* to CL_SUCCESS if it creates the user event object successfully. Otherwise, it returns one of the following error values returned in *err*:

- CL_INVALID_CONTEXT if context is not a valid context.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::UserEvent::setStatus**(cl_int *status*)

sets the execution status of a user event object.

status specifies the new execution status to be set and can be CL_COMPLETE or a negative integer value to indicate an error.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::UserEvent::setStatus returns CL_SUCCESS if the function was executed successfully.
Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if the *status* is not CL_COMPLETE or a negative integer value.
- CL_INVALID_OPERATION if the *status* for event has already been changed by a previous call to **cl::UserEvent::setStatus**.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

3.9 Command Queues

Class **cl::NDRange** provides functionality for working with global and local NDRanges, as described in section 5.1 of the *OpenCL Specification*. This is a necessary type for certain enqueue commands.

The constructor

cl::NDRange::NDRange(::size_t *size0*)

returns a 1D range.

size0 describes the number of global or local work-items in dimension 0.

The constructor

cl::NDRange::NDRange (::size_t *size0,*
::size_t *size1*)

returns a 2D range.

size0 describes the number of global or local work-items in dimension 0.
size1 describes the number of global or local work-items in dimension 1.

The constructor

cl::NDRange::NDRange (::size_t *size0*,
::size_t size1,
::size_t *size2*)

returns a 3D range.

size0 describes the number of global or local work-items in dimension 0.
size1 describes the number of global or local work-items in dimension 1.
size2 describes the number of global or local work-items in dimension 2.

The operator

cl::NDRange::operator const ::size_t *** ()** const

returns a pointer to an array of, 1, 2, or 3 elements of the range.

The method

::size_t **cl::NDRange::dimensions**(void)

returns the number of dimensions defined in the range.

Class **cl::CommandQueue** provides functionality for working with OpenCL command-queues.

The constructor

cl::CommandQueue::CommandQueue(
const Context& *context*,
const Device& *device*,
cl_command_queue_properties *properties* = 0,
cl_int * *err* = NULL)

creates a command-queue on a specific device.

device must be a device associated with context. It can either be in the list of devices specified when context is created using **cl::Context::Context**.

properties specifies a list of properties for the command-queue. This is a bit-field and is described in table 5.1 of the *OpenCL Specification* Version 1.2. Only command-queue properties specified in table 5.1 can be set in properties; otherwise the value specified in properties is considered to be not valid.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

The constructor

explicit cl::CommandQueue::CommandQueue(
const Context& *context*,
cl_command_queue_properties *properties* = 0,
cl_int * *err* = NULL)

creates a command-queue on the first device in the context.

properties specifies a list of properties for the command-queue. This is a bit-field and is described in table 5.1 of the *OpenCL Specification* Version 1.2. Only command-queue properties specified in table 5.1 can be set in properties; otherwise the value specified in properties is considered to be not valid.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::CommandQueue::CommandQueue returns a valid command-queue and sets *err* to CL_SUCCESS if it creates the command-queue successfully. Otherwise, it returns one of the following error values returned in *err*:

- CL_INVALID_CONTEXT if context is not a valid context.
- CL_INVALID_DEVICE if device is not a valid device or is not associated with context.
- CL_INVALID_VALUE if values specified in properties are not valid.
- CL_INVALID_QUEUE_PROPERTIES if values specified in properties are valid but are not supported by the device.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

template <typename *T*>
cl_int **cl::CommandQueue::getInfo**(cl_command_queue_info *name*,
*T * param*)

gets specific information about an OpenCL command queue. Table 5.2 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table below lists **cl_command_queue_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

cl_command_queue_info	**C++ return Type**
CL_QUEUE_CONTEXT	cl::Context
CL_QUEUE_DEVICE	cl::Device

Table 12: Differences in cl::CommandQueue::getInfo return type vs. *OpenCL Specification* table 5.2

T is a compile time argument that is the return type for the specific information being queried. It corresponds to the values in table 5.2.

name is an enumeration constant that identifies the command-queue information being queried. It can be one of the values specified in table 5.2.

param is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.2 is returned. If *param* is NULL, it is ignored.

cl::CommandQueue::getInfo returns CL_SUCCESS on success. Otherwise, it returns:

- CL_INVALID_VALUE if *name* is not one of the supported values.

The method

template <cl_int *name*> typename
detail::param_traits<detail::cl_command_queue_info, *name*>::param_type
cl::CommandQueue::getInfo(void)

gets specific information about the OpenCL command-queue. Table 5.2 of the *OpenCL Specification* Version 1.2 specifies the information that can be queried. The table above lists **cl_command_queue_info** values that differ in return type between the OpenCL C API and the OpenCL C++ API.

name is an enumeration constant that identifies the command-queue information being queried. It can be one of the values specified in table 5.2.

cl::CommandQueue::getInfo returns the appropriate value for a given *name* as specified in table 5.2.

The methods

cl_int **cl::CommandQueue::enqueueReadBuffer**(
const Buffer& *buffer*,
cl_bool *blocking_read*,
::size_t *offset*,
::size_t *size*,
const void * *ptr*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

and

cl_int **cl::CommandQueue::enqueueWriteBuffer**(
const Buffer& *buffer*,
cl_bool *blocking_write*,
::size_t *offset*,
::size_t *size*,
const void * *ptr*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueue a command to read from a buffer object to host memory or to write to a buffer object from host memory.

buffer refers to a valid buffer object.

blocking_read and *blocking_write* indicate if the read and write operations are blocking or nonblocking.

If *blocking_read* is CL_TRUE (i.e., the read command is blocking), **cl::CommandQueue::enqueueReadBuffer** does not return until the buffer data has been read and copied into memory pointed to by *ptr*.

If *blocking_read* is CL_FALSE (i.e., the read command is non-blocking), **cl::CommandQueue::enqueueReadBuffer** queues a non-blocking read command and returns. The contents of the buffer to which *ptr* points cannot be used until the read command has completed. The event argument returns an event object which can be used to query the execution status of the read command. When the read command has completed, the contents of the buffer that ptr points to can be used by the application.

If *blocking_write* is CL_TRUE, the OpenCL implementation copies the data referred to by *ptr* and enqueues the write operation in the command-queue. The memory pointed to by *ptr* can be reused by the application after the **cl::CommandQueue::enqueueWriteBuffer** call returns.

If *blocking_write* is CL_FALSE, the OpenCL implementation will use *ptr* to perform a nonblocking write. As the write is non-blocking the implementation can return immediately. The memory pointed to by *ptr* cannot be reused by the application after the call returns. The event argument returns an event object which can be used to query the execution status of the write command. When the write command has completed, the memory pointed to by *ptr* can then be reused by the application.

offset is the offset in bytes in the buffer object to read from or write to.

size is the size in bytes of data being read or written.

ptr is the pointer to buffer in host memory where data is to be read into or to be written from.

events is the list of events that need to complete before this particular command can be executed. If *events* is NULL or of zero length, then this particular command does not wait on any event to complete. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular read / write command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueReadBuffer and **cl::CommandQueue::enqueueWriteBuffer** return CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if the context associated with command-queue and buffer are not the same or if the context associated with command-queue and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if buffer is not a valid buffer object.
- CL_INVALID_VALUE if the region being read or written specified by (offset, size) is out of bounds or if *ptr* is a NULL value.
- CL_INVALID_EVENT_WAIT_LIST if event objects in *events* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if buffer is a sub-buffer object and offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with queue.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with buffer.

- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The methods

cl_int **cl::CommandQueue::enqueueReadBufferRect**(
const Buffer& *buffer*,
cl_bool *blocking_read*,
const size_t<3> *buffer_offset*,
const size_t<3> *host_offset*,
const size_t<3> *region*,
::size_t buffer_row_pitch,
::size_t buffer_slice_pitch,
::size_t host_row_pitch,
::size_t host_slice_pitch,
void * *ptr*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * event = NULL)

and

cl_int **cl::CommandQueue::enqueueWriteBufferRect**(
const Buffer& *buffer*,
cl_bool *blocking_write*,
const size_t<3> & *buffer_offset*,
const size_t<3> & *host_offset*,
const size_t<3>& *region*,
::size_t *buffer_row_pitch*,
::size_t *buffer_slice_pitch*,
::size_t *host_row_pitch*,
::size_t *host_slice_pitch*,
void * *ptr*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueue command to read a 2D or 3D rectangular region from a buffer object to host memory or write a 2D or 3D rectangular region of a buffer object from host memory.

buffer refers to a valid buffer object.

blocking_read and *blocking_write* indicate if the read and write operations are blocking or nonblocking.

If *blocking_read* is CL_TRUE (i.e., the read command is blocking),
cl::ComamndQueue::enqueueReadBufferRect does not return until the buffer data has been read and copied into memory pointed to by *ptr*.

If *blocking_read* is CL_FALSE (i.e., the read command is non-blocking),
cl::ComamndQueue::enqueueReadBufferRect queues a non-blocking read command and returns. The contents of the buffer that *ptr* points to cannot be used until the read command has completed. The event

argument returns an event object which can be used to query the execution status of the read command. When the read command has completed, the contents of the buffer that *ptr* points to can be used by the application.

If *blocking_write* is CL_TRUE, the OpenCL implementation copies the data referred to by *ptr* and enqueues the write operation in the command-queue. The memory pointed to by *ptr* can be reused by the application after the **cl::ComamndQueue::enqueueWriteBufferRect** call returns.

If *blocking_write* is CL_FALSE, the OpenCL implementation will use ptr to perform a nonblocking write. As the write is non-blocking the implementation can return immediately. The memory pointed to by *ptr* cannot be reused by the application after the call returns. The event argument returns an event object which can be used to query the execution status of the write command. When the write command has completed, the memory pointed to by *ptr* can then be reused by the application.

buffer_origin defines the (x, y, z) offset in the memory region associated with buffer. For a 2D rectangle region, the z value given by buffer_origin[2] should be 0. The offset in bytes is computed as *buffer_origin*[2] * *buffer_slice_pitch* + *buffer_origin*[1] * *buffer_row_pitch* + *buffer_origin*[0].

host_origin defines the (x, y, z) offset in the memory region pointed to by ptr. For a 2D rectangle region, the z value given by *host_origin*[2] should be 0. The offset in bytes is computed as *host_origin*[2] * *host_slice_pitch* + *host_origin*[1] * *host_row_pitch* + *host_origin*[0].

region defines the (width, height, depth) in bytes of the 2D or 3D rectangle being read or written. For a 2D rectangle copy, the depth value given by *region*[2] should be 1.

buffer_row_pitch is the length of each row in bytes to be used for the memory region associated with *buffer*. If *buffer_row_pitch* is 0, *buffer_row_pitch* is computed as *region*[0].

buffer_slice_pitch is the length of each 2D slice in bytes to be used for the memory region associated with buffer. If *buffer_slice_pitch* is 0, *buffer_slice_pitch* is computed as *region*[1] * *buffer_row_pitch*.

host_row_pitch is the length of each row in bytes to be used for the memory region pointed to by *ptr*. If *host_row_pitch* is 0, *host_row_pitch* is computed as *region*[0].

host_slice_pitch is the length of each 2D slice in bytes to be used for the memory region pointed to by *ptr*. If *host_slice_pitch* is 0, *host_slice_pitch* is computed as region[1] * *host_row_pitch*.

ptr is the pointer to buffer in host memory where data is to be read into or to be written from.

events specifies the events that need to complete before this particular command can be executed. If *events* is NULL or of zero length, then this particular command does not wait on any event to complete. If *events* is not NULL and non-zero length, the list of events must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular read / write command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueReadBufferRect and **cl::CommandQueue::enqueueWriteBufferRect** return CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if the context associated with command-queue and *buffer* are not the same or if the context associated with command-queue and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if buffer is not a valid buffer object.
- CL_INVALID_VALUE if the region being read or written specified by (*buffer_offset*, *region*) is out of bounds.
- CL_INVALID_VALUE if *ptr* is a NULL value.
- CL_INVALID_EVENT_WAIT_LIST if event objects in *events* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if buffer is a sub-buffer object and offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with queue.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with buffer.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::enqueueCopyBuffer**(
const Buffer & *src*,
const Buffer & dst,
::size_t *src_offset*,
::size_t *dst_offset*,
::size_t *size*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to copy a buffer object identified by *src* to another buffer object identified by *dst*. The OpenCL context associated with command-queue, *src* and *dst* must be the same.

src refers to the offset where to begin copying data from *src*.

dst refers to the offset where to begin copying data into *dst*.

size refers to the size in bytes to copy.

events specifies events that need to complete before this particular command can be executed. If *events* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *events* must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete. **cl::CommandQueue::enqueueBarrier** can be used instead.

cl::ComamndQueue::enqueueCopyBuffer returns CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if the context associated with the command-queue, *src* and *dst* are not the same or if the context associated with the command-queue and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if *src* and *dst* are not valid buffer objects.
- CL_INVALID_VALUE if *src*, *dst*, *size*, *src + size* or *dst + size* require accessing elements outside the *src* and *dst* buffer objects respectively.
- CL_INVALID_EVENT_WAIT_LIST if event objects in *events* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *src* is a sub-buffer object and offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with queue.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *dst* is a sub-buffer object and offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with queue.
- CL_MEM_COPY_OVERLAP if *src* and *dst* are the same buffer object and the source and destination regions overlap.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src* or *dst*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::enqueueCopyBufferRect**(
const Buffer& *src_buffer*,
const Buffer& *dst_buffer*,
const size_t<3>& *src_origin*,
const size_t<3>& *dst_origin*,
const size_t<3>& *region*,
::size_t src_row_pitch,
::size_t src_slice_pitch,
::size_t *dst_row_pitch*,
::size_t *dst_slice_pitch*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to copy a 2D or 3D rectangular region from the buffer object identified by *src* to a 2D or 3D region in the buffer object identified by *dst*. The OpenCL context associated with the command-queue, *src* and *dst* must be the same.

src_origin defines the (x, y, z) offset in the memory region associated with *src_buffer*. For a 2D rectangle region, the z value given by *src_origin*[2] should be 0. The offset in bytes is computed as *src_origin*[2] * *src_slice_pitch* + *src_origin*[1] * *src_row_pitch* + *src_origin*[0].

dst_origin defines the (x, y, z) offset in the memory region associated with *dst_buffer*. For a 2D rectangle region, the z value given by *dst_origin*[2] should be 0. The offset in bytes is computed as *dst_origin*[2] * dst_slice_pitch + *dst_origin*[1] * *dst_row_pitch* + *dst_origin*[0].

region defines the (width, height, depth) in bytes of the 2D or 3D rectangle being copied. For a 2D rectangle, the depth value given by *region*[2] should be 1.

src_row_pitch is the length of each row in bytes to be used for the memory region associated with *src_buffer*. If *src_row_pitch* is 0, *src_row_pitch* is computed as *region*[0].

src_slice_pitch is the length of each 2D slice in bytes to be used for the memory region associated with *src_buffer*. If *src_slice_pitch* is 0, *src_slice_pitch* is computed as *region*[1] * *src_row_pitch*.

dst_row_pitch is the length of each row in bytes to be used for the memory region associated with *dst_buffer*. If *dst_row_pitch* is 0, *dst_row_pitch* is computed as *region*[0].

dst_slice_pitch is the length of each 2D slice in bytes to be used for the memory region associated with *dst_buffer*. If *dst_slice_pitch* is 0, *dst_slice_pitch* is computed as *region*[1] * *dst_row_pitch*.

events specifies events that need to complete before this particular command can be executed. If e*vents* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *events* must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueCopyBufferRect returns CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if the context associated with the command-queue, *src_buffer* and *dst_buffer* are not the same or if the context associated with the command-queue and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if *src_buffer* and *dst_buffer* are not valid buffer objects.
- CL_INVALID_VALUE if (*src_offset*, region) or (*dst_offset*, region) require accessing elements outside the *src_buffer* and *dst_buffer* buffer objects respectively.
- CL_INVALID_EVENT_WAIT_LIST if event objects in *events* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *src_buffer* is a sub-buffer object and offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with queue.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *dst_buffer* is a sub-buffer object and offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with queue.
- CL_MEM_COPY_OVERLAP if *src_buffer* and *dst_buffer* are the same buffer object and the source and destination regions overlap.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_buffer* or *dst_buffer*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.

- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

template<typename *PatternType*>
cl_int **cl::CommandQueue::enqueueFillBuffer**(const Buffer& *buffer*,
PatternType pattern,
::size_t *offset*,
::size_t *size*,
const VECTOR_CLASS<Event>* *events* = NULL,
Event* *event* = NULL) const

enqueues a command to fill a buffer object with a pattern of a given size.

PatternType must be an OpenCL data type.

buffer specifies the buffer to be filled.

pattern specifies the pattern.

offset specifies the offset into the buffer where filling begins.

size specifies the size of the pattern.

events specifies events that need to complete before this particular command can be executed. If e*vents* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *events* must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueFillBuffer returns CL_SUCCESS on success.

The methods

cl_int **cl::CommandQueue::enqueueReadImage**(
const Image& *image*,
cl_bool *blocking_read*,
const size_t<3>& *origin*,
const size_t<3>& *region*,
::size_t *row_pitch*,
::size_t *slice_pitch*,
void * *ptr*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

and

cl_int **cl::CommandQueue::enqueueWriteImage**(
const Image& *image*,
cl_bool *blocking_write*,
const size_t<3>& *origin*,
const size_t<3>& *region*,
::size_t *row_pitch*,
::size_t *slice_pitch*,
const void * *ptr*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues commands to read from a 2D or 3D image object to host memory or write to a 2D or 3D image object from host memory.

image refers to a valid 2D or 3D image object.

blocking_read and *blocking_write* indicate if the read and write operations are blocking or nonblocking.

If *blocking_read* is CL_TRUE (i.e., the read command is blocking), **cl::CommandQueue::enqueueReadImage** does not return until the buffer data has been read and copied into memory pointed to by *ptr*.

If *blocking_read* is CL_FALSE (i.e., the read command is non-blocking), **cl::CommandQueue::enqueueReadImage** queues a non-blocking read command and returns. The contents of the buffer to which *ptr* points cannot be used until the read command has completed. The event argument returns an event object that can be used to query the execution status of the read command. When the read command has completed, the contents of the buffer that *ptr* points to can be used by the application.

If *blocking_write* is CL_TRUE, the OpenCL implementation copies the data referred to by *ptr* and enqueues the write command in the command-queue. The memory pointed to by *ptr* can be reused by the application after the **cl::CommandQueue::enqueueWriteImage** call returns.

If *blocking_write* is CL_FALSE, the OpenCL implementation will use *ptr* to perform a nonblocking write. As the write is non-blocking the implementation can return immediately. The memory pointed to by *ptr* cannot be reused by the application after the call returns. The event argument returns an event object which can be used to query the execution status of the write command. When the write command has completed, the memory pointed to by *ptr* can then be reused by the application.

origin defines the (x, y, z) offset in pixels in the image from where to read or write. If image is a 2D image object, the z value given by *origin*[2] must be 0.

region defines the (width, height, depth) in pixels of the 2D or 3D rectangle being read or written. If image is a 2D image object, the depth value given by *region*[2] must be 1.

row_pitch in **cl::CommandQueue::enqueueReadImage** and *input_row_pitch* in **cl::CommandQueue::enqueueWriteImage** is the length of each row in bytes. This value must be greater than or equal to the element size in bytes * width. If *row_pitch* (or *input_row_pitch*) is set to 0, the appropriate row pitch is calculated based on the size of each element in bytes multiplied by width.

slice_pitch in **cl::CommandQueue::enqueueReadImage** and *input_slice_pitch* in **cl::CommandQueue::enqueueWriteImage** is the size in bytes of the 2D slice of the 3D region of a 3D image being read or written respectively. This must be 0 if image is a 2D image. This value must be greater than or equal to *row_pitch* * height. If *slice_pitch* (or *input_slice_pitch*) is set to 0, the appropriate slice pitch is calculated based on the *row_pitch* * height.

ptr is the pointer to a buffer in host memory where image data is to be read from or to be written to.

events specifies events that need to complete before this particular command can be executed. If *events* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and of nonzero length, the list of events pointed to by *events* must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueReadImage and **cl::CommandQueue::enqueueWriteImage** return CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if the context associated with the command-queue and image are not the same or if the context associated with the command-queue and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if image is not a valid image object.
- CL_INVALID_VALUE if the region being read or written specified by *origin* and *region* is out of bounds or if *ptr* is a NULL value.
- CL_INVALID_VALUE if image is a 2D image object and *origin*[2] is not equal to 0 or *region*[2] is not equal to 1 or *slice_pitch* is not equal to 0.
- CL_INVALID_EVENT_WAIT_LIST if event objects in *events* are not valid events.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for image are not supported by device associated with queue.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with image.
- CL_INVALID_OPERATION if the device associated with the command-queue does not support images (i.e., CL_DEVICE_IMAGE_SUPPORT specified in table 4.3 is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::enqueueCopyImage**(
const Image& *src_image*,
const Image& *dst_image*,
const size_t <3>& *src_origin*,
const size_t <3>& *dst_origin*,
const size_t<3>& region,

const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to copy image objects.

src_image and *dst_image* can be 2D or 3D image objects allowing us to perform the following actions:

- Copy a 2D image object to a 2D image object.
- Copy a 2D image object to a 2D slice of a 3D image object.
- Copy a 2D slice of a 3D image object to a 2D image object.
- Copy a 3D image object to a 3D image object.

The OpenCL context associated with command-queue, *src_image* and *dst_image* must be the same.

src_origin defines the starting (x, y, z) location in pixels in *src_image* from where to start the data copy. If *src_image* is a 2D image object, the z value given by *src_origin*[2] must be 0.

dst_origin defines the starting (x, y, z) location in pixels in *dst_image* from where to start the data copy. If *dst_image* is a 2D image object, the z value given by *dst_origin*[2] must be 0
region defines the (width, height, depth) in pixels of the 2D or 3D rectangle to copy. If *src_image* or *dst_image* is a 2D image object, the depth value given by *region*[2] must be 1.

events specifies events that need to complete before this particular command can be executed. If *events* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *events* must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a
wait for this command to complete.

It is currently a requirement that the *src_image* and *dst_image* image memory objects for **cl::CommandQueue::enqueueCopyImage** must have the exact same image format (i.e., the **cl_image_format** descriptor specified when *src_image* and *dst_image* are created must match).

cl::CommandQueue::enqueueCopyImage returns CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if the context associated with the command-queue, *src_image* and *dst_image* are not the same or if the context associated with the command-queue and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if *src_image* and *dst_image* are not valid image objects.
- CL_IMAGE_FORMAT_MISMATCH if *src_image* and *dst_image* do not use the same image format.
- CL_INVALID_VALUE if the 2D or 3D rectangular region specified by *src_origin* and *src_origin* + *region* refers to a region outside *src_image*, or if the 2D or 3D rectangular region specified by *dst_origin* and *dst_origin* + region refers to a region outside *dst_image*.
- CL_INVALID_VALUE if *src_image* is a 2D image object and *src_origin*[2] is not equal to 0 or *region*[2] is not equal to 1.

- CL_INVALID_VALUE if *dst_image* is a 2D image object and *dst_origin*[2] is not equal to 0 or *region*[2] is not equal to 1.
- CL_INVALID_EVENT_WAIT_LIST if event objects in *events* are not valid events.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *src_image* are not supported by device associated with queue.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *dst_image* are not supported by device associated with queue.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_image* or *dst_image*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::enqueueFillImage**(const Image& *image*,
cl_float4 *fillColor*,
const size_t<3>& *origin*,
const size_t<3>& *region*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to fill an image object with a given fill color.

image is a valid image object.

fill_color is the four component RGBA fill color.

origin defines the (x, y, z) offset in pixels in the image to fill. If *image* is a 2D image object, the z value given by *origin*[2] must be 0.

region defines the (width, height, depth) in pixels of the 2D or 3D rectangle to fill. If *image* is a 2D image object, the depth value given by *region*[2] must be 1.

events specifies events that need to complete before this particular command can be executed. If *events* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *events* must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a
wait for this command to complete.

cl::CommandQueue::enqueueFillImage returns CL_SUCCESS on success.

The method

cl_int **cl::CommandQueue::enqueueCopyImageToBuffer**(
const Image& *src_image*,
const Buffer& *dst_buffer*,
const size_t <3>& *src_origin*,
const size_t<3>& *region*,
const ::size_t *dst_offset*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to copy an image object to a buffer object. The OpenCL context associated with the command-queue, *src_image* and *dst_buffer* must be the same.

src_image is a valid image object.

dst_buffer is a valid buffer object.

src_origin defines the (x, y, z) offset in pixels in the image from where to copy. If *src_image* is a 2D image object, the z value given by *src_origin*[2] must be 0.

region defines the (width, height, depth) in pixels of the 2D or 3D rectangle to copy. If *src_image* is a 2D image object, the depth value given by *region*[2] must be 1.

dst_offset refers to the offset where to begin copying data into *dst_buffer*. The size in bytes of the region to be copied referred to as *dst_cb* is computed as width * height * depth * bytes/image element if *src_image* is a 3D image object and is computed as width * height * bytes/image element if *src_image* is a 2D image object.

events specifies events that need to complete before this particular command can be executed. If *events* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *event*s must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueCopyImageToBuffer returns CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if the context associated with the command-queue, *src_image* and *dst_buffer* are not the same or if the context associated with the command-queue and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if *src_image* is not a valid image object or *dst_buffer* is not a valid buffer object.
- CL_INVALID_VALUE if the 2D or 3D rectangular region specified by *src_origin* and *src_origin* + *region* refers to a region outside *src_image*, or if the region specified by *dst_offset* and *dst_offset* + *dst_cb* to a region outside *dst_buffer*.
- CL_INVALID_VALUE if *src_image* is a 2D image object and *src_origin*[2] is not equal to 0 or *region*[2] is not equal to 1.

- CL_INVALID_EVENT_WAIT_LIST if event objects in *events* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *dst_buffer* is a sub-buffer object and offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with queue. CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *src_image* are not supported by device associated with queue.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_image* or *dst_buffer*.
- CL_INVALID_OPERATION if the device associated with the command-queue does not support images (i.e., CL_DEVICE_IMAGE_SUPPORT specified in table 4.3 is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::enqueueCopyBufferToImage**(
const Buffer& *src_buffer*,
const Image& *dst_image*,
const ::size_t *src_offset*,
const size_t <3>& *dst_origin*,
const size_t<3>& *region*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to copy a buffer object to an image object. The OpenCL context associated with the command-queue, *src_buffer* and *dst_image* must be the same.

src_buffer is a valid buffer object.

dst_image is a valid image object.
src_offset refers to the offset where to begin copying data from *src_buffer*.

dst_origin refers to the (x, y, z) offset in pixels where to begin copying data to *dst_image*. If *dst_image* is a 2D image object, the z value given by *dst_origin*[2] must be 0.

region defines the (width, height, depth) in pixels of the 2D or 3D rectangle to copy. If *dst_image* is a 2D image object, the depth value given by *region*[2] must be 1. The size in bytes of the region to be copied from *src_buffer* referred to as *src_cb* is computed as width * height * depth * bytes/image element if *dst_image* is a 3D image object and is computed as width * height * bytes/image element if *dst_image* is a 2D image object.

events specifies events that need to complete before this particular command can be executed. If e*vents* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *event*s must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to

query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueCopyBufferToImage returns CL_SUCCESS on success. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if the context associated with the command-queue, *src_buffer* and *dst_image* are not the same or if the context associated with the command-queue and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if *src_buffer* is not a valid buffer object or *dst_image* is not a valid image object.
- CL_INVALID_VALUE if the 2D or 3D rectangular region specified by *dst_origin* and *dst_origin* + region refer to a region outside *dst_image*, or if the region specified by *src_offset* and *src_offset* + *src_cb* refer to a region outside *src_buffer*.
- CL_INVALID_EVENT_WAIT_LIST if event objects in *events* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *src_buffer* is a sub-buffer object and offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with queue. CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *dst_image* are not supported by device associated with queue.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_buffer* or *dst_image*.
- CL_INVALID_OPERATION if the device associated with the command-queue does not support images (i.e., CL_DEVICE_IMAGE_SUPPORT specified in table 4.3 is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

```
void * cl::CommandQueue::enqueueMapBuffer(
        const Buffer& buffer,
        cl_bool blocking_map,
        cl_map_flags flags,
        ::size_t offset,
        ::size_t size,
        const VECTOR_CLASS<Event> * events = NULL,
        Event * event = NULL,
        cl_int * err = NULL)
```

enqueues a command to map a region of the buffer object given by *buffer* into the host address space and returns a pointer to this mapped region.

blocking_map indicates if the map operation is *blocking* or *non-blocking*.

If *blocking_map* is CL_TRUE, **cl::CommandQueue::enqueueMapBuffer** does not return until the specified region in *buffer* can be mapped.

If *blocking_map* is CL_FALSE (i.e., the map operation is non-blocking), the pointer to the mapped region returned by **cl::CommandQueue::enqueueMapBuffer** cannot be used until the map command has completed. The *event* argument returns an event object that can be used to query the execution status of the map command. When the map command is completed, the application can access the contents of the mapped region using the pointer returned by **cl::CommandQueue::enqueueMapBuffer**.

map_flags is a bit-field and can be set to CL_MAP_READ to indicate that the region specified by (*offset*, *size*) in the buffer object is being mapped for reading, and/or CL_MAP_WRITE to indicate that the region specified by (*offset*, *size*) in the buffer object is being mapped for writing.

buffer is a valid buffer object. The OpenCL context associated with *command_queue* and *buffer* must be the same.

offset and *size* are the offset in bytes and the size of the region in the buffer object that is being mapped.

events specifies events that need to complete before this particular command can be executed. If e*vents* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *event*s must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::CommandQueue::enqueueMapBuffer returns a pointer to the mapped region and sets *err* to CL_SUCCESS on success.

A NULL pointer is returned otherwise with one of the following error values returned in *err*:

- CL_INVALID_CONTEXT if context associated with the *command queue* and *buffer* are not the same or if the context associated with the *command queue* and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if *buffer* is not a valid buffer object.
- CL_INVALID_VALUE if region being mapped given by (*offset*, *size*) is out of bounds or if values specified in *map_flags* are not valid.
- CL_INVALID_EVENT_WAIT_LIST if event objects in *events* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_MAP_FAILURE if there is a failure to map the requested region into the host address space. This error cannot occur for buffer objects created with CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *buffer*.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The pointer returned maps a region starting at *offset* and is at least *size* bytes in size. The result of a memory access outside this region is undefined.

The method

void * **cl::CommandQueue::enqueueMapImage**(
const Image& *image*,
cl_bool *blocking_map*,
cl_map_flags *map_flags*,
::size_t<3>& *origin*,
::size_t<3>& *region*,
::size_t * *row_pitch*,
::size_t * *slice_pitch*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL,
cl_int * err = NULL)

enqueues a command to map a region in the image object given by *image* into the host address space and returns a pointer to this mapped region.

image is a valid image object. The OpenCL context associated with the *command queue* and *image* must be the same.

blocking_map indicates if the map operation is *blocking* or *non-blocking*.

If *blocking_map* is CL_TRUE, **cl::CommandQueue::enqueueMapImage** does not return until the specified region in *image* is mapped.

If *blocking_map* is CL_FALSE (i.e., the map operation is non-blocking), the pointer to the mapped region returned by **cl::CommandQueue::enqueueMapImage** cannot be used until the map command has completed. The *event* argument returns an event object that can be used to query the execution status of the map command. When the map command is completed, the application can access the contents of the mapped region using the pointer returned by **cl::CommandQueue::enqueueMapImage**.

map_flags is a bit-field and can be set to CL_MAP_READ to indicate that the region specified by (*origin*, *region*) in the image object is being mapped for reading, and/or CL_MAP_WRITE to indicate that the region specified by (*origin*, *region*) in the image object is being mapped for writing.

origin and *region* define the (*x*, *y*, *z*) offset in pixels and (*width*, *height*, *depth*) in pixels of the 2D or 3D rectangle region that is to be mapped. If *image* is a 2D image object, the *z* value given by *origin*[2] must be 0 and the *depth* value given by *region*[2] must be 1.

row_pitch returns the scan-line pitch in bytes for the mapped region. This must be a non-NULL value.

slice_pitch returns the size in bytes of each 2D slice for the mapped region. For a 2D image, zero is returned if this argument is not NULL. For a 3D image, *slice_pitch* must be a non-NULL value.

events specifies events that need to complete before this particular command can be executed. If e*vents* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not

NULL and a nonzero length, the list of events pointed to by *event*s must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

err returns an appropriate error code. If *err* is NULL, no error code is returned.

cl::CommandQueue::enqueueMapImage returns a pointer to the mapped region and sets *err* to CL_SUCCESS on success. It returns NULL otherwise, with one of the following error values returned in *err*:

- CL_INVALID_CONTEXT if context associated with the *command queue* and *image* are not the same or if context associated with the *command queue* and events in *events* are not the same.
- CL_INVALID_MEM_OBJECT if *image* is not a valid image object.
- CL_INVALID_VALUE if region being mapped given by (*origin*, *origin*+*region*) is out of bounds or if values specified in *map_flags* are not valid.
- CL_INVALID_VALUE if *image* is a 2D image object and *origin*[2] is not equal to 0 or *region*[2] is not equal to 1.
- CL_INVALID_VALUE if *row_pitch* is NULL.
- CL_INVALID_VALUE if *image* is a 3D image object and *slice_pitch* is NULL.
- CL_INVALID_EVENT_WAIT_LIST if event objects in events are not valid events.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *image* are not supported by device associated with *queue*.
- CL_MAP_FAILURE if there is a failure to map the requested region into the host address space. This error cannot occur for image objects created with CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *buffer*.
- CL_INVALID_OPERATION if the device associated with the *command queue* does not support images (i.e., CL_DEVICE_IMAGE_SUPPORT specified in table 4.3 is CL_FALSE).
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The pointer returned maps a 2D or 3D region starting at *origin* and is at least (*row_pitch * region[1] + region[0])* pixels in size for a 2D image, and is at least (*slice_pitch * region[2] +row_pitch * region[1] + region[0])* pixels in size for a 3D image. The result of a memory access outside this region is undefined.

If the image object is created with CL_MEM_USE_HOST_PTR set in *mem_flags*, the following will be true:

- The *host_ptr* specified in **cl::Image{2D|3D}** is guaranteed to contain the latest bits in the region being mapped when the **cl::CommandQueue::enqueueMapImage** command has completed.

- The pointer value returned by **cl::CommandQueue::enqueueMapImage** is derived from the *host_ptr* specified when the image object is created.

Mapped image objects are unmapped using **cl::CommandQueue::enqueueUnmapMemObject**. This is described in the following text.

cl_int **cl::CommandQueue::enqueueUnmapMemObject**(
const Memory& *memory*,
void * mapped_ptr,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to unmap a previously mapped region of a memory object. Reads or writes from the host using the pointer returned by **cl::CommandQueue::enqueueMapBuffer**, or **cl::CommandQueue::enqueueMapImage** are considered to be complete.

memobj is a valid memory object. The OpenCL context associated with *command_queue* and *memobj* must be the same.

mapped_ptr is the host address returned by a previous call to **cl::CommandQueue::enqueueMapBuffer**, or **cl::CommandQueue::enqueueMapImage** for *memobj*.

events specifies events that need to complete before this particular command can be executed. If e*vents* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *event*s must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command-queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete. **Cl::CommandQueue::enqueueBarrier** can be used instead.

Cl::CommandQueue::enqueueUnmapMemObject returns CL_SUCCESS on success. Otherwise it returns one of the following errors:

- CL_INVALID_MEM_OBJECT if *memobj* is not a valid memory object.
- CL_INVALID_VALUE if *mapped_ptr* is not a valid pointer returned by **cl::CommandQueue::enqueueMapBuffer**, or **clCommandQueue::enqueueMapImage** for *memobj*.
- CL_INVALID_EVENT_WAIT_LIST if event objects in events are not valid events.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.
- CL_INVALID_CONTEXT if context associated with the *command queue* and *memobj* are not the same or if the context associated with the *command queue* and events in *events* are not the same.

cl::CommandQueue::enqueueMapBuffer, and **cl::CommandQueue::enqueueMapImage** increments the mapped count of the memory object. The initial mapped count value of the memory object is zero. Multiple calls to **cl::CommandQueue::enqueueMapBuffer**, or **cl::CommandQueue::enqueueMapImage** on the same

memory object will increment this mapped count by appropriate number of calls.
cl::CommandQueue::enqueueUnmapMemObject decrements the mapped count of the memory object.

cl::CommandQueue::enqueueMapBuffer, and **cl::CommandQueue::enqueueMapImage** act as synchronization points for a region of the buffer object being mapped.

The method

cl_int **cl::CommandQueue::enqueueMarkerWithWaitList**(
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a marker command that waits for completion of a list of events.

events specifies a list of events. If NULL or empty, the marker command waits for the completion of all previously enqueued commands.

event returns an event object that identifies this command. It can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueMarkerWithWaitList returns CL_SUCCESS on success.

The method

cl_int **cl::CommandQueue::enqueueBarrierWithWaitList**(
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a barrier command that waits for completion of a list of events.

events specifies a list of events. If NULL or empty, the barrier command waits for the completion of all previously enqueued commands.

event returns an event object that identifies this command. It can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueBarrierWithWaitList returns CL_SUCCESS on success.

The method

cl_int **cl::CommandQueue::enqueueMigrateMemObjects**(
const VECTOR_CLASS<Memory> &*memObjects*,
cl_mem_migration_flags *flags*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command indicating with which device a set a memory objects should be associated.

memObjects specifies a set of memory objects.

flags specifies migration flags, as in table 5.10 of the *OpenCl Specification* Version 1.2. It must be nonempty.

events specifies a list of events on which the command waits. If NULL, the command does not wait.

event returns an event object that identifies this command. It can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueMigrateMemObjects returns CL_SUCCESS on success. Otherwise, it returns the error status from the underlying *clEnqueueMigrateMemObjects* call.

The method

cl_int **cl::CommandQueue::enqueueNDRangeKernel**(
const Kernel& *kernel*,
const NDRange& offset,
const NDRange& global,
const NDRange& local,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to execute a kernel on a device.

kernel is a valid kernel object. The OpenCL context associated with *kernel* and the *command queue* must be the same.

offset can be used to specify an array of *work_dim* unsigned values that describe the offset used to calculate the global ID of a work-item. If *offset* is cl::NULLRange, the global IDs start at offset (0, 0, … 0).

global points to an array of *work_dim* unsigned values that describe the number of global work-items in *work_dim* dimensions that will execute the kernel function. The total number of global work-items is computed as *global* [0] * … * *globa* [*work_dim* – 1].

local points to an array of *work_dim* unsigned values that describe the number of work-items that make up a work-group (also referred to as the size of the work-group) that will execute the kernel specified by *kernel*. The total number of work-items in a work-group is computed as *local*[0] * … * *local* [*work_dim* – 1]. The total number of work-items in the work-group must be less than or equal to the CL_DEVICE_MAX_WORK_GROUP_SIZE value specified in table 4.3 and the number of work-items specified in *local* [0], … *local* [*work_dim* – 1] must be less than or equal to the corresponding values specified by CL_DEVICE_MAX_WORK_ITEM_SIZES[0], …. CL_DEVICE_MAX_WORK_ITEM_SIZES[*work_dim* – 1]. The explicitly specified *local* determines how to break the global work-items specified by *global* into appropriate work-group instances. If *local* is specified, the values specified in *global* [0], … *global* [*work_dim* - 1] must be evenly divisible by the corresponding values specified in *local* [0], … *local* [*work_dim* – 1].

The work-group size to be used for *kernel* can also be specified in the program source using the __attribute__((reqd_work_group_size(X, Y, Z)))qualifier (refer to *section 6.8.2*). In this case the size of work group specified by *local* must match the value specified by the reqd_work_group_size attribute qualifier.

local can also be a cl::NULLRange value in which case the OpenCL implementation will determine how to be break the global work-items into appropriate work-group instances.

These work-group instances are executed in parallel across multiple compute units or concurrently on the same compute unit.

Each work-item is uniquely identified by a global identifier. The global ID, which can be read inside the kernel, is computed using the value given by *global_work_size* and *global_work_offset*. In addition, a work-item is also identified within a work-group by a unique local ID. The local ID, which can also be read by the kernel, is computed using the value given by *local_work_size*. The starting local ID is always (0, 0, … 0).

events specifies events that need to complete before this particular command can be executed. If e*vents* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *event*s must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete. c**l::CommandQueue::enqueueBarrier** can be used instead.

cl::CommandQueue::enqueueNDRangeKernel returns CL_SUCCESS if the kernel execution was successfully queued. Otherwise, it returns one of the following errors:

- CL_INVALID_PROGRAM_EXECUTABLE if there is no successfully built program executable available for device associated with the *command queue*.
- CL_INVALID_KERNEL if *kernel* is not a valid kernel object.
- CL_INVALID_CONTEXT if context associated with the *command queue* and *kernel* are not the same or if the context associated with the *command queue* and events in *events* are not the same.
- CL_INVALID_KERNEL_ARGS if the kernel argument values have not been specified.
- CL_INVALID_GLOBAL_WORK_SIZE if *global* is cl::NULLRange, or if any of the values specified in *global* [0], … *global* [*work_dim* – 1] are 0 or exceed the range given by the sizeof(size_t) for the device on which the kernel execution is enqueued.
- CL_INVALID_GLOBAL_OFFSET if the value specified in *global* + the corresponding values in *global* for any dimensions is greater than the sizeof(size t) for the device on which the kernel execution is enqueued.
- CL_INVALID_WORK_GROUP_SIZE if *local* is specified and number of work-items specified by *global* is not evenly divisible by size of work-group given by *local* or does not match the work-group size specified for *kernel* using the __attribute__((reqd_work_group_size(X, Y, Z))) qualifier in program source.

- CL_INVALID_WORK_GROUP_SIZE if *local* is specified and the total number of work-items in the work-group computed as *local* [0] * … *local* [*work_dim* – 1] is greater than the value specified by CL_DEVICE_MAX_WORK_GROUP_SIZE in table 4.3.

- CL_INVALID_WORK_GROUP_SIZE if *local* is NULL and the __attribute__((reqd_work_group_size(X, Y, Z))) qualifier declares the work-group size for *kernel* in the program source.

- CL_INVALID_WORK_ITEM_SIZE if the number of work-items specified in any of *local* [0], … *local* [*work_dim* – 1] is greater than the corresponding values specified by CL_DEVICE_MAX_WORK_ITEM_SIZES[0], …. CL_DEVICE_MAX_WORK_ITEM_SIZES[*work_dim* – 1].

- CL_MISALIGNED_SUB_BUFFER_OFFSET if a sub-buffer object is specified as the value for an argument that is a buffer object and the *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.

- CL_INVALID_IMAGE_SIZE if an image object is specified as an argument value and the image dimensions (image width, height, specified or compute row and/or slice pitch) are not supported by device associated with *queue*.

- CL_OUT_OF_RESOURCES if there is a failure to queue the execution instance of *kernel* on the command-queue because of insufficient resources needed to execute the kernel. For example, the explicitly specified *local* causes a failure to execute the kernel because of insufficient resources such as registers or local memory. Another example would be the number of read-only image args used in *kernel* exceed the CL_DEVICE_MAX_READ_IMAGE_ARGS value for device or the number of write-only image args used in *kernel* exceed the CL_DEVICE_MAX_WRITE_IMAGE_ARGS value for device or the number of samplers used in *kernel* exceed CL_DEVICE_MAX_SAMPLERS for device.

- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with image or buffer objects specified as arguments to *kernel*.

- CL_INVALID_EVENT_WAIT_LIST if event objects in events are not valid events.

- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::enqueueTask**(
const Kernel& *kernel*,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to execute a kernel on a device. The kernel is executed using a single work-item.

kernel is a valid kernel object. The OpenCL context associated with *kernel* and *command-queue* must be the same.

events specifies events that need to complete before this particular command can be executed. If e*vents* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *event*s must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete. c**l::CommandQueue::enqueueBarrier** can be used instead.

cl::CommandQueue::enqueueTask is equivalent to calling **cl::CommandQueue::enqueueNDRangeKernel** with *work_dim* = 1, *global* = NULLRange, *global* [0] set to 1 and *local* [0] set to 1.

Cl::CommandQueue::enqueueTask returns CL_SUCCESS if the kernel execution was successfully queued. Otherwise, it returns one of the following errors:

- CL_INVALID_PROGRAM_EXECUTABLE if there is no successfully built program executable available for device associated with the *command queue*.
- CL_INVALID_KERNEL if *kernel* is not a valid kernel object.
- CL_INVALID_CONTEXT if context associated with the *command queue* and *kernel* are not the same or if the context associated with the *command queue* and events in *events* are not the same.
- CL_INVALID_KERNEL_ARGS if the kernel argument values have not been specified.
- CL_INVALID_WORK_GROUP_SIZE if a work-group size is specified for *kernel* using the __attribute__((reqd_work_group_size(X, Y, Z))) qualifier in program source and is not (1, 1, 1).
- CL_MISALIGNED_SUB_BUFFER_OFFSET if a sub-buffer object is specified as the value for an argument that is a buffer object and the *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_INVALID_IMAGE_SIZE if an image object is specified as an argument value and the image dimensions (image width, height, specified or compute row and/or slice pitch) are not supported by device associated with *queue*.
- CL_OUT_OF_RESOURCES if there is a failure to queue the execution instance of *kernel* on the command-queue because of insufficient resources needed to execute the kernel.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with image or buffer objects specified as arguments to *kernel*.
- CL_INVALID_EVENT_WAIT_LIST if event objects in events are not valid events.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

```
cl_int cl::CommandQueue::enqueueNativeKernel(
        void (*user_func) (void *),
        std::pair<void*, ::size_t> args,
        const VECTOR_CLASS<Memory> * mem_objects = NULL,
```

const VECTOR_CLASS<const void *> * *mem_locs* = NULL,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to execute a native C/C++ function not compiled using the OpenCL compiler.

A native user function can only be executed on a command queue created on a device that has CL_EXEC_NATIVE_KERNEL capability set in CL_DEVICE_EXECUTION_CAPABILITIES as specified in table 4.3.

user_func is a pointer to a host-callable user function.

args is tuple containing a pointer to the args list that *user_func* should be called with and the is the size in bytes of the arggument list that *args* points to.

The data pointed to by *args.fst* and *args.snd* bytes in size is copied and a pointer to this copied region is passed to *user_func*. The copy needs to be done because the memory objects (cl_mem values) that *args* may contain need to be modified and replaced by appropriate pointers to global memory. When **cl::CommandQueue::enqueueNativeKernel** returns, the memory region pointed to by *args* can be reused by the application.

mem_objects is a list of valid buffer objects. The buffer object values specified in *mem_objects* are memory objects (*cl::Memory* values) returned by **cl::Buffer**.

mem_loc is a vector of appropriate locations that *args* points to where memory objects (*cl::Memory* values) are stored. Before the user function is executed, the memory object handles are replaced by pointers to global memory.

events specifies events that need to complete before this particular command can be executed. If e*vents* is NULL or of length zero, then this particular command does not wait on any event to complete. If *events* is not NULL and a nonzero length, the list of events pointed to by *event*s must be valid. The events specified in *events* act as synchronization points. The context associated with events in *events* and the command queue must be the same.

event returns an event object that identifies this particular copy command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete. c**l::CommandQueue::enqueueBarrier** can be used instead.

Cl::CommandQueue::enqueueNativeKernel returns CL_SUCCESS if the user function execution instance was successfully queued. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if context associated with the *command queue* and events in *events* are not the same.
- CL_INVALID_VALUE if *user_func* is NULL.
- CL_INVALID_VALUE if *args.fst* is a NULL value and *args.snd* > 0, or if *args.fst* is a NULL value and then length of *mem_objects* > 0.
- CL_INVALID_VALUE if *args.fst* is not NULL and *args.snd* is 0.

- CL_INVALID_VALUE if the length of *mem_objects* > 0 and *mem_locs* is NULL.
- CL_INVALID_VALUE if length of *mem_objects* is 0 and *mem_locs* is not NULL.
- CL_INVALID_OPERATION if *device* cannot execute the native kernel.
- CL_INVALID_MEM_OBJECT if one or more memory objects specified in *mem_objects* are not valid or are not buffer objects.
- CL_OUT_OF_RESOURCES if there is a failure to queue the execution instance of *kernel* on the command-queue because of insufficient resources needed to execute the kernel.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with buffer objects specified as arguments to *kernel*.
- CL_INVALID_EVENT_WAIT_LIST if event objects in events are not valid events.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::enqueueMarker**(Event * *event* = NULL)

enqueues a marker command to the command queue. The command returns an *event* which can be used to queue a wait on this marker (i.e., to wait for all commands queued before the marker to complete).

cl::CommandQueue::enqueueMarker returns CL_SUCCESS if the function is successfully executed. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *event* is a NULL value.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::enqueueWaitForEvents**(
const VECTOR_CLASS<Event>& *events*)

enqueues a wait for a specific event or a list of events to complete before any future commands queued in the command queue are executed. Each event in *events* must be a valid event object returned by a previous call to **cl::CommandQueue::enqueue***.

The events specified in *events* act as synchronization points. The context associated with events in *events* and then *command queue* must be the same.

cl::CommandQueue::enqueueWaitForEvents returns CL_SUCCESS if the function was successfully executed. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if the context associated with *command_queue* and events in *event_list* are not the same.
- CL_INVALID_VALUE if the length of *events* is 0.
- CL_INVALID_EVENT if event objects specified in *events* are not valid events.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::enqueueAcquireGLObjects**(
const VECTOR_CLASS<Memory>* *mem_objects* = NULL,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to acquire OpenCL memory objects created from OpenGL objects.

mem_objects is the list of OpenCL memory objects that correspond to OpenGL objects.

events specifies a list of events on which the command waits. If NULL or empty, the command waits for the completion of all previously enqueued commands.

event returns an event object that identifies this command. It can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueAcquireGLObjects returns CL_SUCCESS on success.

The method

cl_int **cl::CommandQueue::enqueueReleaseGLObjects**(
const VECTOR_CLASS<Memory>* *mem_objects* = NULL,
const VECTOR_CLASS<Event> * *events* = NULL,
Event * *event* = NULL)

enqueues a command to release OpenCL memory objects created from OpenGL objects.

mem_objects is the list of OpenCL memory objects that correspond to OpenGL objects.

events specifies a list of events on which the command waits. If NULL, the command waits for the completion of all previously enqueued commands.

event returns an event object that identifies this command. It can be used to query or queue a wait for this particular command to complete. *event* can be NULL, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

cl::CommandQueue::enqueueReleaseGLObjects returns CL_SUCCESS on success.

The method

cl_int **cl::CommandQueue::enqueueBarrier**(void)

enqueues a barrier operation. The **cl::CommandQueue::enqueueBarrier** command ensures that all queued commands in *command_queue* have finished execution before the next batch of commands can begin execution. The **cl::CommandQueue::enqueueBarrier** command is a synchronization point.

cl::CommandQueue::enqueueBarrier returns CL_SUCCESS if the function was executed successfully. It returns CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::flush**(void)

issues all previously queued OpenCL commands in the *command queue* to the device. **cl::CommandQueue::flush** only guarantees that all queued commands to *command queue* get issued to the appropriate device. There is no guarantee that they will be complete after **cl::CommandQueue::flush** returns.

cl::CommandQueue::flush returns CL_SUCCESS if the function call was executed successfully. Otherwise it returns CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

cl_int **cl::CommandQueue::finish**(void)

blocks until all previously queued OpenCL commands in the *command queue* are issued to the associated device and have completed. **cl::CommandQueue::finish** does not return until all queued commands in then *command queue* have been processed and completed. **cl::CommandQueue::finish** is also a synchronization point.

cl::CommandQueue::finish returns CL_SUCCESS if the function call was executed successfully. Otherwise it returns CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

4. Exceptions

The use of C++ exceptions can provide a structured approach to error handling for large applications. The C++ API provides the ability to use C++ exceptions to track and handle errors generated by the underlying OpenCL C API. However, the use of C++ exceptions is not universal and their use should be optional. In the case that exceptions are not used, the resulting application must compile and work without exception support.

By default, C++ exceptions are not enabled and the OpenCL error code is returned, or set, as per the underlying C API. To enable the use of exceptions, the user must explicitly define preprocessor macro:

__CL_ENABLE_EXCEPTIONS

Once enabled, an error (i.e., a value other than CL_SUCCESS) originally reported via return value instead will be reported by throwing an exception of class **cl::Error**. By default, the method **cl::Error::what()** returns a const pointer to a string naming the particular C API call that reported the error (e.g., "clGetDeviceInfo", "clGetPlatformInfo", and so on).

To override the default behavior for **cl::Error::what()**, define the preprocessor macro:

__CL_USER_OVERRIDE_ERROR_STRINGS

and provide string constants for each preprocessor macro defined in Table 11. Most macros apply to both OpenCL 1.1 and OpenCL 1.2; these have a blank Version column in the table. Macros marked V1.2 in the Version column are defined for OpenCL 1.2 only. Macros marked V1.1 in the Version column are defined for OpenCL 1.1 only; the corresponding OpenCL APIs are deprecated, and the user must define

__CL_USE_DEPRECATED_OPENCL_1_1_APIS

to use these V1.1 macros.

Preprocessor macro name	Default value	Version
__BUILD_PROGRAM_ERR	"clBuildProgram"	
__COMPILE_PROGRAM_ERR	"clCompileProgram"	V1.2
__COPY_ERR	"clCopy"	
__CREATE_BUFFER_ERR	"clCreateBuffer"	
__CREATE_COMMAND_QUEUE_ERR	"clCreateCommandQueue"	
__CREATE_CONTEXT_ERR	"clCreateContext"	
__CREATE_CONTEXT_FROM_TYPE_ERR	"clCreateContextFromType"	
__CREATE_GL_BUFFER_ERR	"clCreateFromGLBuffer"	
__CREATE_GL_RENDER_BUFFER_ERR	"clCreateFromGLBuffer"	
__CREATE_GL_TEXTURE_2D_ERR	"clCreateFromGLTexture2D"	V1.1
__CREATE_GL_TEXTURE_3D_ERR	"clCreateFromGLTexture3D"	V1.1
__CREATE_GL_TEXTURE_ERR	"clCreateFromGLTexture"	V1.2
__CREATE_IMAGE2D_ERR	"clCreateImage2D"	V1.1
__CREATE_IMAGE3D_ERR	"clCreateImage3D"	V1.1
__CREATE_IMAGE_ERR	"clCreateImage"	V1.2
__CREATE_KERNELS_IN_PROGRAM_ERR	"clCreateKernelsInProgram"	
__CREATE_KERNEL_ERR	"clCreateKernel"	
__CREATE_PROGRAM_WITH_BINARY_ERR	"clCreateProgramWithBinary"	
__CREATE_PROGRAM_WITH_BUILTIN_KERNELS_ERR	"clCreateProgramWithBuiltinKernels"	V1.2
__CREATE_PROGRAM_WITH_SOURCE_ERR	"clCreateProgramWithSource"	
__CREATE_SAMPLER_ERR	"clCreateSampler"	
__CREATE_SUBBUFFER_ERR	"clCreateSubBuffer"	
__CREATE_SUB_DEVICES_ERR	"clCreateSubDevicesEXT"	!V1.2
__CREATE_SUB_DEVICES_ERR	"clCreateSubDevices"	V1.2
__CREATE_USER_EVENT_ERR	"clCreateUserEvent"	
__ENQUEUE_ACQUIRE_GL_ERR	"clEnqueueAcquireGLObjects"	
__ENQUEUE_BARRIER_ERR	"clEnqueueBarrier"	V1.1
__ENQUEUE_COPY_BUFFER_ERR	"clEnqueueCopyBuffer"	
__ENQUEUE_COPY_BUFFER_RECT_ERR	"clEnqueueCopyBufferRect"	
__ENQUEUE_COPY_BUFFER_TO_IMAGE_ERR	"clEnqueueCopyBufferToImage"	
__ENQUEUE_COPY_IMAGE_ERR	"clEnqueueCopyImage"	
__ENQUEUE_COPY_IMAGE_TO_BUFFER_ERR	"clEnqueueCopyImageToBuffer"	
__ENQUEUE_FILL_BUFFER_ERR	"clEnqueueFillBuffer"	
__ENQUEUE_FILL_IMAGE_ERR	"clEnqueueFillImage"	

__ENQUEUE_MAP_BUFFER_ERR	“clEnqueueMapBuffer”	
__ENQUEUE_MAP_IMAGE_ERR	“clEnqueueMapImage”	
__ENQUEUE_MARKER_ERR	“clEnqueueMarker”	V1.1
__ENQUEUE_MIGRATE_MEM_OBJECTS_ERR	“clEnqueueMigrateMemObjects”	V1.2
__ENQUEUE_NATIVE_KERNEL	“clEnqueueNativeKernel”	
__ENQUEUE_NDRANGE_KERNEL_ERR	“clEnqueueNDRangeKernel”	
__ENQUEUE_READ_BUFFER_ERR	“clEnqueueReadBuffer”	
__ENQUEUE_READ_BUFFER_RECT_ERR	“clEnqueueReadBufferRect”	
__ENQUEUE_READ_IMAGE_ERR	“clEnqueueReadImage”	
__ENQUEUE_RELEASE_GL_ERR	“clEnqueueReleaseGLObjects”	
__ENQUEUE_TASK_ERR	“clEnqueueTask”	
__ENQUEUE_UNMAP_MEM_OBJECT_ERR	“clEnqueueUnmapMemObject”	
__ENQUEUE_WAIT_FOR_EVENTS_ERR	“clEnqueueWaitForEvents”	V1.1
__ENQUEUE_WRITE_BUFFER_ERR	“clEnqueueWriteBuffer”	
__ENQUEUE_WRITE_BUFFER_RECT_ERR	“clEnqueueWriteBufferRect”	
__ENQUEUE_WRITE_IMAGE_ERR	“clEnqueueWriteImage”	
__FINISH_ERR	“clFinish”	
__FLUSH_ERR	“clFlush”	
__GET_COMMAND_QUEUE_INFO_ERR	“clGetCommandQueueInfo”	
__GET_CONTEXT_INFO_ERR	“clGetContextInfo”	
__GET_DEVICE_IDS_ERR	“clGetDeviceIds”	
__GET_DEVICE_INFO_ERR	“clGetDeviceInfo”	
__GET_EVENT_INFO_ERR	“clGetEventInfo”	
__GET_EVENT_PROFILE_INFO_ERR	“clGetEventProfileInfo”	
__GET_GL_OBJECT_INFO_ERR	“clGetGLObjectInfo”	
__GET_IMAGE_INFO_ERR	“clGetImageInfo”	
__GET_KERNEL_ARG_INFO_ERR	“clGetKernelArgInfo”	V1.2
__GET_KERNEL_INFO_ERR	“clGetKernelInfo”	
__GET_KERNEL_WORK_GROUP_INFO_ERR	“clGetKernelWorkGroupInfo”	
__GET_MEM_OBJECT_INFO_ERR	“clGetMemObjectInfo”	
__GET_PLATFORM_IDS_ERR	“clGetPlatformIDs”	
__GET_PLATFORM_INFO_ERR	“clGetPlatformInfo”	
__GET_PROGRAM_BUILD_INFO_ERR	“clGetProgramBuildInfo”	
__GET_PROGRAM_INFO_ERR	“clGetProgramInfo”	
__GET_SAMPLER_INFO_ERR	“clGetSampleInfo”	
__GET_SUPPORTED_IMAGE_FORMATS_ERR	“clGetSupportedImageFormats”	
__IMAGE_DIMENSION_ERR	“Incorrect image dimensions”	V1.2
__RELEASE_ERR	“Release Object”	
__RETAIN_ERR	“Retain Object”	
__SET_COMMAND_QUEUE_PROPERTY_ERR	“clSetCommandQueueProperty”	
__SET_EVENT_CALLBACK_ERR	“clSetEventCallback”	
__SET_KERNEL_ARGS_ERR	“clSetKernelArgs”	
__SET_MEM_OBJECT_DESTRUCTOR_CALLBACK_ERR	“clSetMemObjectDestructorCallback”	
__SET_PRINTF_CALLBACK_ERR	“clSetPrintfCallback”	V1.2
__SET_USER_EVENT_STATUS_ERR	“clSetUserEventStatus”	
__UNLOAD_COMPILER_ERR	“clUnloadCompiler”	V1.1
__VECTOR_CAPACITY_ERR	“Vector capacity error”	
__WAIT_FOR_EVENTS_ERR	“clWaitForEvents”	

Table 13: Preprocessor error macros and their defaults.

5. Using the C++ API with the Standard Template Library

The C++ Standard Template Library is an excellent resource for quick access to many useful algorithms and containers, but it is often not used due to compatibility issues across different toolchains and operating systems, among other reasons. The OpenCL C++ wrapper API uses vectors and strings in a number of places. By default, it uses Standard Template Library vector and string classes **std::vector** and **std::string**. However, it also gives the developer the ability to not use these.

The C++ wrapper API provides replacements **cl::vector** for **std::vector** and **cl::string** for **std::string**. It also allows developers to use their own implementations instead.

By default, to avoid issues with backward compatibility, the C++ wrapper uses both **std::vector** and **std::string**. Either can be overridden, with **cl::vector** and **cl::type**; however, the developer should be aware that these types are deprecated. For vectors, the developer can select an alternative version by defining the preprocessor macro:

__NO_STD_VECTOR

In this case, the following vector type is defined:

template **cl::vector**<typename *T*,
unsigned int *N* = __MAX_DEFAULT_VECTOR_SIZE>

cl::vector shares the same interface as **std::vector**, but it has a statically defined space requirement, by default 10 elements. The developer can manually override this allocation by defining the preprocessor macro:

__MAX_DEFAULT_VECTOR_SIZE *N*

where *N* is the number of vector elements to use when allocating values of type **cl::vector**.

By defining the preprocessor macro:

__USE_DEV_VECTOR

neither **std::vector** nor **cl::vector** classes will be used. Instead, the user must provide the preprocessor definition:

VECTOR_CLASS *typeName*

where *typeName* corresponds to the user's vector class[4], with an implementation that matches the **std::vector** interface.

For strings, if the preprocessor macro:

[4] Few C++ compilers currently support typedef templates and thus the vector type must be given by its name only through the preprocessor macro VECTOR_CLASS.

__NO_STD_STRING

is defined, then the string type **cl::string** is used instead of **std::string**. Unlike **cl::vector,** the size of a given string is not defined statically but is allocated at creation. A developer can provide a replacement implementation for **std::string** by defining the preprocessor macro:

__USE_DEV_STRING

The developer must provide the following typedef:

typedef *stringType* **STRING_CLASS**

where *stringType* is the user provided alternative for **std::string**, with an implementation that matches the interface for **std::string.**

6. Index

This index lists each class, constructor, and method defined by the C++ wrapper API. Methods are listed under their fully qualified names; e.g., the many **getInfo** methods are listed under **cl::Context::getInfo**, **cl::Device::getinfo**, etc., not under **getInfo**.