



# The **OpenCL** C++ Wrapper API

*Version: 1.1*

*Document Revision: 04*

Khronos OpenCL Working Group

*Author:* Benedict R. Gaster

1.	Introduction .....	4
2.	C++ Platform layer.....	4
2.1	Querying Platform Info .....	4
2.2	Devices .....	5
2.3	Contexts .....	7
3.	C++ Runtime layer .....	10
3.1	Memory Objects.....	10
3.2	Buffer Objects .....	12
3.3	Images .....	13
3.3.1	Image 2D objects.....	14
3.3.2	Image 3D objects.....	16
3.4	Samplers.....	17
3.5	Programs .....	19
3.6	Kernels.....	24
3.7	Events.....	27
3.8	User Events .....	30
3.9	Command Queues.....	31
4.	Exceptions .....	58
5.	Using the C++ API with Standard Template Library .....	60

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be re-formatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group web-site should be included whenever possible with specification distributions.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Khronos, OpenKODE, OpenKOGS, OpenVG, OpenMAX, OpenSL ES, glFX and OpenWF are trademarks of the Khronos Group Inc. COLLADA is a trademark of Sony Computer Entertainment Inc. used by permission by Khronos. OpenGL and OpenML are registered trademarks and the OpenGL ES logo is a trademark of Silicon Graphics Inc. used by permission by Khronos. OpenCL is a registered trademark of Apple Inc. used by permission by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

# 1. Introduction

This specification describes the OpenCL C++ Wrapper API and is intended to be read alongside The OpenCL 1.1 Specification. The wrapper is designed to be built on top of the OpenCL 1.1 C API and is not a replacement. It is expected that any implementation of the C++ Wrapper API will make calls to underlying C API and it is assumed that the C API is a compliant implementation of the OpenCL 1.1 Specification platform and runtime API.

The interface is contained within a single C++ header file *cl.hpp* and all definitions are contained within the namespace *cl*. There is no additional requirement to include *cl.h* and to use either the C++ or original C API it is enough to simply include *cl.hpp*.

The C++ API corresponds closely to the underlying C API and introduces no additional execution overhead.

The API is divided into a number of classes that have a corresponding OpenCL C type, for example, there is a *cl::Memory* class that maps to *cl\_mem* in OpenCL C. When possible C++ inheritance is used to provide an extra level of type correctness and abstraction, for example *cl::Buffer* derives from the base class *cl::Memory* but represents the 1D memory subclass of all possible OpenCL memory objects.

The following sections describe each of class in detail.

## 2. C++ Platform layer

### 2.1 Querying Platform Info

The class *cl::Platform* provides functionality for working with OpenCL platforms. The list of platforms available can be obtained using the following static method<sup>1</sup>

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

*platforms* is a vector of OpenCL platforms found.

**cl::Platform::get** returns CL\_SUCCESS if the function is executed successfully. 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)
```

gets specific information about the OpenCL platform. The information that can be queried is specified in table 4.1.

---

<sup>1</sup> The C++ types VECTOR\_CLASS and STRING\_CLASS are described in section XXX.  
Last Revision Date: 6/14/2010

*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 will be returned. If *param* is NULL, it is ignored.

**cl::Platform::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 4.1.

*name* is a compile time argument 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 list of devices available on a platform can be obtained using the following method

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

*type* is a bitfield that identified the type of OpenCL device. The *type* can be used to query specific OpenCL devices or all OpenCL devices available. The valid values for *type* are specified in table 4.2.

*devices* returns a vector of OpenCL devices found. If *devices* argument is NULL, this argument is ignored.

**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\_DEVICE\_NOT\_FOUND if no OpenCL devices that matched *type* were found.

## 2.2 Devices

The 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 ID.

*device* is an OpenCL device.

The method

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

gets specific information about the OpenCL device. The information that can be queried is specified in table 4.3 and in conjunction with table 1<sup>2</sup>.

<b>cl_device_info</b>	<b>Return Type</b>
CL_DEVICE_MAX_WORK_ITEM_SIZES	VECTOR_CLASS\$<::size_t>
CL_DEVICE_NAME	STRING_CLASS
CL_DEVICE_VENDOR	STRING_CLASS
CL_DEVICE_PROFILE	STRING_CLASS
CL_DEVICE_VERSION	STRING_CLASS
CL_DRIVER_VERSION	STRING_CLASS
CL_DEVICE_OPENCL_C_VERSION	STRING_CLASS
CL_DEVICE_EXTENSIONS	STRING_CLASS

Table 1 Difference in return type for table 4.3 and `cl::Context::getInfo`

*T* is a compile time argument that is the return for the specific information being queried and 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 will be returned. If *param* is NULL, it is ignored.

**cl::Device::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 4.3 and in conjunction with table 1.

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

---

<sup>2</sup> Table 4.3 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::Device::getInfo** functions.

**cl::device::getInfo** returns the appropriate value for a given *name* as specified in table 4.3.

## 2.3 Contexts

The 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 * errinfo,  
                        const void * private_info_size,  
                        ::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 4.4. *properties* can be NULL in which case the platform that is selected is implementation-defined.

*pfn\_notify* is a callback function that can be registered by the application. This callback function will be 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 this 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* will be passed as the *user\_data* argument when *pfn\_notify* is called. *user\_data* can be NULL.

*err* will return 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 *err* is set to CL\_SUCCESS if the context is created 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 equal to 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 * errinfo,
                        const void * private_info_size,
                        ::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 device(s) to use.

*type* is a bit-field that identifies the type of device and is described in table 4.2 in section 4.2.

*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 that can be registered by the application. This callback function will be 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 this 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* will be passed as the *user\_data* argument when *pfn\_notify* is called. *user\_data* can be NULL.

*err* will return 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 *err* is set to CL\_SUCCESS if the context is created 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. The information that can be queried is specified in table 4.5 and in conjunction with table 2<sup>3</sup>.

cl_context_info	Return Type
CL_CONTEXT_DEVICES	VECTOR_CLASS<cl::Device>
CL_CONTEXT_PROPERTIES	VECTOR_CLASS<cl_context_properties>

Table 2: Difference in return type for table 4.5 and **cl::Context::getInfo**.

*T* is a compile time argument that is the return for the specific information being queried and corresponds to the values in tables 4.5.

*name* is an enumeration constant that identifies the context information being queried. It can be one of the values specified in table 4.5.

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

**cl::Context::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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)
```

<sup>3</sup> Table 2 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::Context::getInfo** functions.

gets specific information about the OpenCL context. The information that can be queried is specified in table 4.5 and in conjunction with table 2.

*name* is a compile time argument is an enumeration constant that identifies the context information being queried. It can be one of the values specified in table 4.5.

**cl::Context::getInfo** returns the appropriate value for a given *name* as specified in table 4.5.

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 is used to specify allocation and usage information about the image memory object being created and is described in table 5.3.

*image\_type* describes the image type and 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 are returned. Each entry describes a instance of the class *cl::ImageFormat*, itself 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 if the function is executed successfully. 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

The class *cl::Memory* provides a base class for working with OpenCL memory objects and 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. The information that can be queried is specified in table 5.9 and in conjunction with table 3<sup>4</sup>.

<b>cl_memory_info</b>	<b>Return Type</b>
CL_MEM_CONTEXT	cl::Context

Table 3: Difference in return type for table 5.9 and **cl::Memory::getInfo**

*T* is a compile time argument that is the return for the specific information being queried and corresponds to the values in tables 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 will be returned. If *param* is NULL, it is ignored.

**cl::Memory::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 5.9 and in conjunction with table 3.

*name* is a compile time argument 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::Memory::getInfo** returns the appropriate value for a given *name* as specified in table 5.9.

The method

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

---

<sup>4</sup> Table 3 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::Memory::getInfo** functions.

registers a user callback function that will be called when the memory object is deleted and its resources freed. See description of **clSetMemObjectDestructorCallback**, in section 5.4, for a detailed overview.

*pfn\_notify* is the callback function that can be 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 this callback function are:

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

*user\_data* will be 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 Buffer Objects

The 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 is used to specify allocation and usage information such as the memory arena that should be used to allocate the buffer object and how it will be used. Table 5.3 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 the buffer data that may already be allocated by the application. The size of the buffer that *host\_ptr* points to must be  $\geq$  *size* bytes.

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

**cl::Buffer::Buffer** creates a valid non-zero buffer object and *err* is set to CL\_SUCCESS if the buffer object is created 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)
```

can be used to create a new buffer object from an existing buffer object.

*flags* is a bit-field that is used to specify allocation and usage information about the image memory object being created and is described in table 5.3.

*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 that

*buffer\_create\_info* points to is described in table 5.4.

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

**cl::Buffer::createSubBuffer** returns CL\_SUCCESS, in *err* if the function is executed successfully. 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(s) specified in *buffer\_create\_info* (for a given *buffer\_create\_type*) is not valid or if *buffer\_create\_info* is NULL.

## 3.3 Images

The class *cl::Image: public Memory* provides a base class for working with OpenCL image objects and is used to build 2D and 3D and 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. The information that can be queried is specified in table 5.8.

*T* is a compile time argument that is the return for the specific information being queried and corresponds to the values in tables 5.8.

*name* is an enumeration constant that identifies the context information being queried. It can be one of the values specified in table 5.8.

*param* is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.8 will be returned. If *param* is NULL, it is ignored.

**cl::Memory::getImageInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 5.8.

*name* is a compile time argument is an enumeration constant that identifies the memory object information being queried. It can be one of the values specified in table 5.8.

**cl::Image::getImageInfo** returns the appropriate value for a given *name* as specified in table 5.8.

### 3.3.1 Image 2D objects

The 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 is used to specify allocation and usage information about the image memory object being created and is described in table 5.3.

*format* is a class<sup>5</sup> that describes format properties of the image to be allocated. Refer to section 5.3.1.1 for a detailed description of the image format descriptor.

*width*, and *height* are the width and height of the 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  $\geq \text{width} * \text{size of element in bytes}$  if *host\_ptr* is not NULL. If *host\_ptr* is not NULL and *row\_pitch* = 0, *row\_pitch* is calculated as  $\text{width} * \text{size of element in bytes}$ . 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  $\geq \text{row\_pitch} * \text{height}$ . 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.

*ret* will return an appropriate error code. If *ret* is NULL, no error code is returned.

**cl::Image2D::Image2D** returns a valid non-zero image object and *err* is set to CL\_SUCCESS if the image object is created 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.

---

<sup>5</sup> *cl::ImageFormat* is class a mapping for *cl\_image\_format* structure supported by the OpenCL implementation.  
Last Revision Date: 6/14/2010

- 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.

### 3.3.2 Image 3D objects

The 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 is used to specify allocation and usage information about the image memory object being created and is described in table 5.3.

*format* is a class<sup>6</sup> that describes format properties of the image to be allocated. Refer to section 5.3.1.1 for a detailed description of the image format descriptor.

*width*, and *height* are the width and height of the image in pixels. These must be values 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.

---

<sup>6</sup> *cl::ImageFormat* is class a mapping for *cl\_image\_format* structure supported by the OpenCL implementation.  
Last Revision Date: 6/14/2010



*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  $\geq \text{row\_pitch} * \text{height}$  if *host\_ptr* is not NULL. If *host\_ptr* is not NULL and *slice\_pitch* = 0, *slice\_pitch* is calculated as  $\text{row\_pitch} * \text{height}$ . 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  $\geq \text{slice\_pitch} * \text{depth}$ . 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.

*ret* will return an appropriate error code. If *ret* is NULL, no error code is returned.

*cl::Image3D::Image3D* returns a valid non-zero image object and *err* is set to CL\_SUCCESS if the image object is created 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*  $\leq 1$  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.

## 3.4 Samplers

The 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.

Last Revision Date: 6/14/2010

*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. This can be set to CL\_ADDRESS\_MIRRORED\_REPEAT, CL\_ADDRESS\_REPEAT, CL\_ADDRESS\_CLAMP\_TO\_EDGE, CL\_ADDRESS\_CLAMP and CL\_ADDRESS\_NONE.

*filtering\_mode* specifies the type of filter that must be applied when reading an image. This can be CL\_FILTER\_NEAREST, or CL\_FILTER\_LINEAR.

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

**cl::Sampler::Sampler** constructs a valid non-zero sampler object and *err* is set to CL\_SUCCESS if the sampler object is created 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 *addressing\_mode*, *filter\_mode* or *normalized\_coords* or combination of these argument values are 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. The information that can be queried is specified in table 5.10 and in conjunction with table 4<sup>7</sup>.

cl_sampler_info	Return Type
CL_SAMPLER_CONTEXT	cl::Context

Table 4: Difference in return type for table 5.10 and cl::Sampler::getInfo

*T* is a compile time argument that is the return for the specific information being queried and corresponds to the values in tables 5.10.

*name* is an enumeration constant that identifies the sampler information being queried. It can be one of the values specified in table 5.10.

---

<sup>7</sup> Table 4 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::Sampler::getInfo** functions.

*param* is a pointer to a memory location where the appropriate values for a given *name* as specified in table 5.10 will be returned.

If *param* is NULL, it is ignored.

**cl::Sampler::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 5.10 and in conjunction with table 4.

*name* is a compile time argument is an enumeration constant that identifies the sampler information being queried. It can be one of the values specified in table 5.10.

**cl::Sampler::getInfo** returns the appropriate value for a given *name* as specified in table 5.10.

## 3.5 Programs

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

The 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 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* will return an appropriate error code. If *err* is NULL, no error code is returned.

**cl::Program::Program** returns a valid program object and *err* is set to CL\_SUCCESS if the program object is created 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 any entry in sources contains a tuple with NULL or size of 0.
- 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 vector list of devices that are in context. *devices* must be of non-zero length. The binaries are loaded for devices specified in this list. The devices associated with the program object will be the list of devices specified by *devices*. The list of devices specified by *devices* must be devices associated with context.

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

*binary\_status* returns whether the program binary for each device specified in *devices* was loaded successfully or not. It is an array of *num\_devices* entries and returns CL\_SUCCESS in *binary\_status*[*i*] if binary was successfully loaded for device specified by *devices*[*i*]; otherwise CL\_INVALID\_BINARY in *binary\_status*[*i*] if program binary is not a valid binary for the specified device. If *binary\_status* is NULL, it is ignored.

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

**cl::Program::Program** returns a valid program object and *err* is set to CL\_SUCCESS if the program object is created 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.
- CL\_INVALID\_DEVICE if OpenCL devices listed in *devices* are not in the list of devices associated with context.
- CL\_INVALID\_VALUE if *binaries* is length 0 or if any entry in *binaries*[*i*] is not valid.
- CL\_INVALID\_BINARY if an invalid program binary was encountered for any device. *binary\_status* will return 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 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 (compilers and links) a program executable from the program source or binary for all the devices or a specific device(s) in the OpenCL context associated with program.

*devices* is a vector 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 this 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. The list of supported options is described in section 5.6.3.

*pfn\_notify* is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be 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* will be passed as an argument when *pfn\_notify* is called. *data* can be NULL.

**cl::Program::build** returns CL\_SUCCESS if the function is executed successfully. 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 **cl::Program::Program** 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 will be 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

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

gets specific information about the OpenCL Program. The information that can be queried is specified in table 5.11 and in conjunction with table 5<sup>8</sup>

<b>cl_program_info</b>	<b>Return Type</b>
CL_PROGRAM_CONTEXT	cl::Context
CL_PROGRAM_DEVICES	VECTOR_CLASS<cl_device_id>
CL_PROGRAM_SOURCE	STRING_CLASS
CL_PROGRAM_BINARY_SIZES	VECTOR_CLASS<::size_t>
CL_PROGRAM_BINARIES	VECTOR_CLASS<char*>

Table 5: Difference in return type for table 5.11 and **cl::Program::getInfo**

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

*name* is an enumeration constant that identifies the program 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 will be returned. If *param* is NULL, it is ignored.

**cl::Program::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 5.11 and in conjunction with table 5.

*name* is a compile time argument is an enumeration constant that identifies the program information being queried. It can be one of the values specified in table 5.11.

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

---

<sup>8</sup> Table 5 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::Program::getInfo** functions

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. The information that can be queried is specified in table 5.12 and in conjunction with table 6<sup>9</sup>

cl_program_info	Return Type
CL_PROGRAM_BUILD_OPTIONS	STRING_CLASS
CL_PROGRAM_BUILD_LOG	STRING_CLASS

Table 6: in return type for table 5.12 and cl::Program::getBuildInfo

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

*name* is an enumeration constant that identifies the program build 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 will be returned. If *param* is NULL, it is ignored.

**cl::Program::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 5.12 and in conjunction with table 6.

*name* is a compile time argument is an enumeration constant that identifies the program information being queried. It can be one of the values specified in table 5.12.

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

The method

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

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

---

<sup>9</sup> Table 5 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::Program::getBuildInfo** functions.

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

**cl::Program::createKernels** will return 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

The class *cl::Kernels* provides functionality for working with OpenCL kernels.

The constructor

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

will create 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* will return an appropriate error code. If *err* is NULL, no error code is returned.

**cl::Kernel::Kernel** returns a valid kernel object and *err* is set to CL\_SUCCESS if the kernel object is created 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. The information that can be queried is specified in table 5.13 and in conjunction with table 7<sup>10</sup>.

<b>cl_kernel_info</b>	<b>Return Type</b>
CL_KERNEL_FUNCTION_NAME	STRING_CLASS
CL_KERNEL_CONTEXT	cl::Context
CL_KERNEL_PROGRAM	cl::Program

Table 7: Difference in return type for table 5.13 and cl::Kernel::getInfo

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

*name* is an enumeration constant that identifies the kernel 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 will be returned. If *param* is NULL, it is ignored.

**cl::Kernel::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 5.13 and in conjunction with table 7.

*name* is a compile time argument is an enumeration constant that identifies the kernel information being queried. It can be one of the values specified in table 5.13.

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

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 specific to a device. The information that can be queried is specified in table 5.14 and in conjunction with table 8<sup>11</sup>

<sup>10</sup> Table 7 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::Kernel::getInfo** functions.

<sup>11</sup> Table 8 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::Kernel::getWorkGroupInfo** functions.

<b>cl_kernel_work_group_info</b>	<b>Return Type</b>
CL_KERNEL_COMPILE_WORK_GROUP_SIZE	cl::size_t <sup>12</sup>

Table 8: Difference in return type for table 5.14 and cl::Kernel::getWorkGroupInfo

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

*name* is an enumeration constant that identifies the kernel 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 will be returned. If *param* is NULL, it is ignored.

**cl::Kernel::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 5.14 and in conjunction with table 8.

*name* is a compile time argument is an enumeration constant that identifies the kernel information being queried. It can be one of the values specified in table 5.14.

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

The method

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

is used to set 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*<sup>13</sup>, which corresponds to an argument of \_\_local in the kernel object.

<sup>12</sup> 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 will be 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 as the 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.

## 3.7 Events

The 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. The information that can be queried is specified in table 5.15 and in conjunction with table 9<sup>14</sup>.

cl_event_info	Return Type
CL_EVENT_CONTEXT	Cl::Context
CL_EVENT_COMMAND_QUEUE	cl::CommandQueue

Table 9: Difference in return type for table 5.15 and cl::Event::getInfo

*T* is a compile time argument that is the return for the specific information being queried and corresponds to the values in tables 5.15.

*name* is an enumeration constant that identifies the event 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 will be returned. If *param* is NULL, it is ignored.

<sup>13</sup> 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.

<sup>14</sup> Table 9 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::Event::getInfo** functions

**cl::Event::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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 the OpenCL event. The information that can be queried is specified in table 5.15 and in conjunction with table 9.

*name* is a compile time argument is an enumeration constant that identifies the event information being queried. It can be one of the values specified in table 5.15.

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

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. The information that can be queried is specified in table 5.16.

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

*name* is an enumeration constant that identifies the profiling 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 will be returned. If *param* is NULL, it is ignored.

**cl::Event::getInfo** returns CL\_SUCCESS if the function is executed successfully. 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. The information that can be queried is specified in table 5.16.

*name* is a compile time argument is an enumeration constant that identifies the profiling information being queried. It can be one of the values specified in table 5.16.

Last Revision Date: 6/14/2010

**cl::Event::getProfilingInfo** returns the appropriate value for a given *name* as specified in table 5.16. 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 will be 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 will be called in the exact order that the execution status of a command changes.

*pfn\_notify* is the event callback function that can be 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 this 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 will be passed to *command\_exec\_status* instead.
- *user\_data* is a pointer to user supplied data.

*user\_data* will be passed as the *user\_data* argument when *pfn\_notify* is called. *user\_data* can be NULL.

**cl::Event::setCallback** returns CL\_SUCCESS if the function is executed successfully. 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

The 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* will return an appropriate error code. If *err* is NULL, no error code is returned.

**cl::UserEvent::UserEvent** returns a valid object and *err* is set to CL\_SUCCESS if the user event object is created 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.

Last Revision Date: 6/14/2010

- `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* will return 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

The class *cl::NDRange* provides functionality for working with global and local NDRanges as described in section 5.1. This is defined before command queues as it is a necessary type for certain enqueue commands, see the following for details.

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.

Last Revision Date: 6/14/2010

*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

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

The 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. 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.

Last Revision Date: 6/14/2010



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

**cl::CommandQueue::CommandQueue** returns a valid command-queue and *err* is set to CL\_SUCCESS if the command-queue is created 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 the OpenCL event. The information that can be queried is specified in table 5.2 and in conjunction with table 10<sup>15</sup>.

cl_command_queue_info	Return Type
CL_QUEUE_CONTEXT	cl::Context
CL_QUEUE_DEVICE	cl::Device

Table 10: Difference in return type for table 5.2 and cl::CommandQueue::getInfo

*T* is a compile time argument that is the return for the specific information being queried and corresponds to the values in tables 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 will be returned. If *param* is NULL, it is ignored.

**cl::CommandQueue::getInfo** returns CL\_SUCCESS if the function is executed successfully. Otherwise, it returns:

- CL\_INVALID\_VALUE if *name* is not one of the supported values.

The method

---

<sup>15</sup> Table 10 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the **cl::CommandQueue::getInfo** functions.

```
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. The information that can be queried is specified in table 5.2 and in conjunction with table 10.

*name* is a compile time argument 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)
```

```
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 command to read and from a buffer object to host memory or 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 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::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 if the function is executed successfully. 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)

```

```

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 if the function is executed successfully. 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 non zero 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 if the function is executed successfully. 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 *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 non zero 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 if the function is executed successfully. 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

```
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)
```

```
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 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 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 a non zero 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 if the function is executed successfully. 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 non zero 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 if the function is executed successfully. 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::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 non zero 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::enqueueCopyImageToBuffer** returns CL\_SUCCESS if the function is executed successfully. 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**(

Last Revision Date: 6/14/2010

```

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 *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 non zero 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::enqueueCopyBufferToImage** returns CL\_SUCCESS if the function is executed successfully. 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_map_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. 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 which 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 *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 non zero 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.

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

**cl::CommandQueue::enqueueMapBuffer** will return a pointer to the mapped region and *err* is set to CL\_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 Buffer& image,
    cl_bool blocking_map,
    cl_map_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. 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 which 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 *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 non zero 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.

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

**cl::CommandQueue::enqueueMapImage** will return a pointer to the mapped region and *err* is set to CL\_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 *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** will be 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 *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 non zero 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::CommandQueue::enqueueUnmapMemObject** returns CL\_SUCCESS if the function is executed successfully. 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 **cl::CommandQueue::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::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  $global[0] * \dots * global[work\_dim - 1]$ .

*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  $local[0] * \dots * local[work\_dim - 1]$ . 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  $local[0], \dots, local[work\_dim - 1]$  must be less than or equal to the corresponding values specified by `CL_DEVICE_MAX_WORK_ITEM_SIZES[0], \dots, CL_DEVICE_MAX_WORK_ITEM_SIZES[work\_dim - 1]`. The explicitly specified *local* will be used to determine how to break the global work-items specified by *global* into appropriate work-group instances. If *local* is specified, the values specified in  $global[0], \dots, global[work\_dim - 1]$  must be evenly divisible by the corresponding values specified in  $local[0], \dots, local[work\_dim - 1]$ .

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 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 *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 non zero 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::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 will be 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 will be 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 is used to declare 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 *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 non zero 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::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 argument list that *args* points to.

The data pointed to by *args.fst* and *args.snd* bytes in size will be copied and a pointer to this copied region will be 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 *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 non zero 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::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 then *commandqueue*. The marker command returns an *event* which can be used by to queue a wait on this marker event i.e. wait for all commands queued before the marker command 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::enqueueNDRangeKernel**, **cl::CommandQueue::enqueueTask**,  
**cl::CommandQueue::enqueueNativeKernel**,  
**cl::CommandQueue::enqueue{Read|Write}Map}{Buffer|Image}**,  
**cl::CommandQueue::enqueue{Read|Write}BufferRect**,  
**cl::CommandQueue::enqueueCopy{Buffer|Image}**, **cl::CommandQueue::enqueueCopyBufferRect**,  
**cl::CommandQueue::enqueueCopyBufferToImage**, **cl::CommandQueue::enqueueCopyImageToBuffer**  
or **cl::CommandQueue::enqueueMarker**.

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::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 and the C++ API provides the ability to use C++ exceptions to track and handle errors generated by the underlying OpenCL C API.

However, it is understood that the use of C++ exceptions is not universal and their use should to 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 use exception the user must explicitly define the preprocessor macro:

```
__CL_ENABLE_EXCEPTIONS
```

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

It is possible to override the default behavior for **cl::Error::what()** by defining the preprocessor macro:

```
__CL_USER_OVERRIDE_ERROR_STRINGS
```

and providing string constants for each of the preprocessor macros defined in Table 11.

Preprocessor macro name	Default value
GET_DEVICE_INFO_ERR	"clGetDeviceInfo"
GET_PLATFORM_INFO_ERR	"clGetPlatformInfo"
GET_DEVICE_IDS_ERR	"clGetDeviceIds"
GET_CONTEXT_INFO_ERR	"clGetContextInfo"
GET_EVENT_INFO_ERR	"clGetEventInfo"
GET_EVENT_PROFILE_INFO_ERR	"clGetEventProfileInfo"
GET_MEM_OBJECT_INFO_ERR	"clGetMemObjectInfo"
GET_IMAGE_INFO_ERR	"clGetImageInfo"
GET_SAMPLER_INFO_ERR	"clGetSampleInfo"
GET_KERNEL_INFO_ERR	"clGetKernelInfo"
GET_KERNEL_WORK_GROUP_INFO_ERR	"clGetKernelWorkGroupInfo"

Last Revision Date: 6/14/2010

__GET_PROGRAM_INFO_ERR	“clGetProgramInfo”
__GET_PROGRAM_BUILD_INFO_ERR	“clGetProgramBuildInfo”
__GET_COMMAND_QUEUE_INFO_ERR	“clGetCommandQueueInfo”
__CREATE_CONTEXT_FROM_TYPE_ERR	“clCreateContxtFromType”
__GET_SUPPORTED_IMAGE_FORMATS_ERR	“clGetSupportedImageFormats”
__CREATE_BUFFER_ERR	“clCreateBuffer”
__CREATE_SUBBUFFER_ERR	“clCreateSubBuffer”
__CREATE_GL_BUFFER_ERR	“clCreateGLBuffer”
__CREATE_IMAGE2D_ERR	“clCreateImage2D”
__CREATE_IMAGE3D_ERR	“clCreateImage3D”
__SET_MEM_OBJECT_DESTRUCTOR_CALLBACK_ERR	“clSetMemObjectDestructorCallback”
__CREATE_USER_EVENT_ERR	“clCreateUserEvent”
__SET_USER_EVENT_STATUS_ERR	“clSetUserEventStatus”
__SET_EVENT_CALLBACK_ERR	“clSetEventCallback”
__WAIT_FOR_EVENTS_ERR	“clWaitForEvents”
__CREATE_KERNEL_ERR	“clCreateKernel”
__SET_KERNEL_ARGS_ERR	“clSetKernelArgs”
__CREATE_PROGRAM_WITH_SOURCE_ERR	“clCreateProgramWithSource”
__CREATE_PROGRAM_WITH_BINARY_ERR	“clCreateProgramWithBinary”
__BUILD_PROGRAM_ERR	“clBuildProgram”
__CREATE_KERNELS_IN_PROGRAM_ERR	“clCreateKernelsInProgram”
__CREATE_COMMAND_QUEUE_ERR	“clCreateCommandQueue”
__SET_COMMAND_QUEUE_PROPERTY_ERR	“clSetCommandQueueProperty”
__ENQUEUE_READ_BUFFER_ERR	“clEnqueueReadBuffer”
__ENQUEUE_READ_BUFFER_RECT_ERR	“clEnqueueReadBufferRect”
__ENQUEUE_WRITE_BUFFER_ERR	“clEnqueueWriteBuffer”
__ENQUEUE_WRITE_BUFFER_RECT_ERR	“clEnqueueWriteBufferRect”
__ENQUEUE_COPY_BUFFER_ERR	“clEnqueueCopyBuffer”
__ENQUEUE_COPY_BUFFER_RECT_ERR	“clEnqueueCopyBufferRect”
__ENQUEUE_READ_IMAGE_ERR	“clEnqueueReadImage”
__ENQUEUE_WRITE_IMAGE_ERR	“clEnqueueWriteImage”
__ENQUEUE_COPY_IMAGE_ERR	“clEnqueueCopyImage”
__ENQUEUE_COPY_IMAGE_TO_BUFFER_ERR	“clEnqueueCopyImageToBuffer”
__ENQUEUE_COPY_BUFFER_TO_IMAGE_ERR	“clEnqueueCopyBufferToImage”
__ENQUEUE_MAP_BUFFER_ERR	“clEnqueueMapBuffer”
__ENQUEUE_MAP_IMAGE_ERR	“clEnqueueMapImage”
__ENQUEUE_UNMAP_MEM_OBJECT_ERR	“clEnqueueUnmapMemObject”
__ENQUEUE_NDRANGE_KERNEL_ERR	“clEnqueueNDRangeKernel”
__ENQUEUE_TASK_ERR	“clEnqueueTask”
__ENQUEUE_NATIVE_KERNEL	“clEnqueueNativeKernel”
__ENQUEUE_MARKER_ERR	“clEnqueueMarker”
__ENQUEUE_WAIT_FOR_EVENTS_ERR	“clEnqueueWaitForEvents”
__ENQUEUE_BARRIER_ERR	“clEnqueueBarriers”
__UNLOAD_COMPILER_ERR	“clUnloadCompiler”
__FLUSH_ERR	“clFlush”
__FINISH_ERR	“clFinish”

Table 11: Preprocessor error macros and their defaults.

## 5. Using the C++ API with Standard Template Library

While C++'s Standard Template library provides an excellent resource for quick access to many useful algorithms and containers, it is often not used due to compatibility issues across different toolchains and operating systems, among other reasons. The OpenCL C++ API uses vectors and strings in a number of places and by default will use *std::vector* and *std::string*, however, the developer has ability to not include these.

The C++ API provides replacements for both *std::vector* (*cl::vector*) and *std::string* (*cl::string*) or the developer has the option to use their own implementations.

By default, to avoid issues with backward compatibility, both *std::vector* and *std::string* are used. Either can be over ridden. For vectors an alternative version can be selected by defining the preprocessor macro:

```
__NO_STD_VECTOR
```

and in this case the following vector type is defined:

```
template cl::vector<typename T,  
                unsigned int N = __MAX_DEFAULT_VECTOR_SIZE>
```

This type shares the same interface as *std::vector* but has statically defined space requirements, which by default is set to a size of 10 elements. It is possible to manually override this allocation by defining the following 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
```

then neither *std::vector* or *cl::vector* classes will be used and instead the user must provide an implementation that matches the interface for *std::vector* and must provide the following preprocessor definition:

```
VECTOR_CLASS typeName
```

where *typeName* corresponds to the users vector class<sup>16</sup>.

For strings, if the preprocessor macro:

```
__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 allocated at creation, however, unlike *std::string* once created its size cannot

---

<sup>16</sup> C++ does not currently support typedef templates and thus the vector type must be given by its name only through the preprocessor macro **VECTOR\_CLASS**.

change. A developer can provide a replacement implementation for `std::string` by defining the preprocessor macro

```
__USE_DEV_STRING
```

and must provide an implementation that matches the interface for `std::string` and must provide a definition of the following typedef:

```
typedef stringType STRING_CLASS
```

where *stringType* must correspond to the user provided alternative for `std::string`.