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

Last Revision Date: 6/14/2010
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\(^1\)

\[
\text{static cl_int cl::Platform::get(VECTOR_CLASS<Platform> * platforms)}
\]

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

\[
\text{cl_int cl::Platform::getInfo(cl_platform_info name, STRING_CLASS * param)}
\]

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

---

\(^1\) The C++ types VECTOR_CLASS and STRING_CLASS are described in section XXX.
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

```cpp
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

```cpp
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

```cpp
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.

<table>
<thead>
<tr>
<th>cl_device_info</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_DEVICE_MAX_WORK_ITEM_SIZES</td>
<td>VECTOR_CLASS$&lt;::size_t&gt;</td>
</tr>
<tr>
<td>CL_DEVICE_NAME</td>
<td>STRING_CLASS</td>
</tr>
<tr>
<td>CL_DEVICE_VENDOR</td>
<td>STRING_CLASS</td>
</tr>
<tr>
<td>CL_DEVICE_PROFILE</td>
<td>STRING_CLASS</td>
</tr>
<tr>
<td>CL_DEVICE_VERSION</td>
<td>STRING_CLASS</td>
</tr>
<tr>
<td>CL_DRIVER_VERSION</td>
<td>STRING_CLASS</td>
</tr>
<tr>
<td>CL_DEVICE_OPENCL_C_VERSION</td>
<td>STRING_CLASS</td>
</tr>
<tr>
<td>CL_DEVICE_EXTENSIONS</td>
<td>STRING_CLASS</td>
</tr>
</tbody>
</table>

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.

---

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

Last Revision Date: 6/14/2010
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

```cpp
cl::Context::Context(VECTOR_CLASS<Device>& devices,
    cl_context_properties * properties = NULL,
    void (CL_CALLBACK * pfn_notify)(
        const char * errorinfo,
        const void * private_info_size,
        ::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:

- *errorinfo* 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

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

- errorinfo 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.
\texttt{cl::Context::Context} returns a valid object of type \texttt{cl::Context} and \texttt{err} is set to \texttt{CL_SUCCESS} if the context is created successfully. Otherwise, it returns one of the following error values in \texttt{err}:

- \texttt{CL_INVALID_PROPERTY} if context property name in \texttt{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.
- \texttt{CL_INVALID_VALUE} if \texttt{pfn_notify} is \texttt{NULL} but \texttt{user_data} is not \texttt{NULL}.
- \texttt{CL_INVALID_DEVICE_TYPE} if \texttt{type} is not a valid value.
- \texttt{CL_DEVICE_NOT_AVAILABLE} if no devices that match \texttt{type} and property values specified in \texttt{properties} are currently available.
- \texttt{CL_DEVICE_NOT_FOUND} if no devices that match \texttt{type} and property values specified in \texttt{properties} were found.
- \texttt{CL_OUT_OF_HOST_MEMORY} if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

\begin{verbatim}
template <typename T>
cl_int cl::Context::getInfo(cl_context_info name, 
                          T * param)
\end{verbatim}

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.

\begin{table}[h]
\begin{tabular}{|l|l|}
\hline
\textbf{cl\_context\_info} & \textbf{Return Type} \\
\hline
CL\_CONTEXT\_DEVICES & VECTOR\_CLASS<cl::Device> \\
CL\_CONTEXT\_PROPERTIES & VECTOR\_CLASS<cl\_context\_properties> \\
\hline
\end{tabular}
\caption{Difference in return type for table 4.5 and \texttt{cl::Context::getInfo}.}
\end{table}

\texttt{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.

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

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

\texttt{cl::Context::getInfo} returns \texttt{CL_SUCCESS} if the function is executed successfully. Otherwise, it returns:

- \texttt{CL_INVALID_VALUE} if \texttt{name} is not one of the supported values.

The method

\begin{verbatim}
template <cl_int name> typename
detail::param_traits<detail::cl\_context\_info, name>::param_type
cl::Context::getInfo(void)
\end{verbatim}

\footnote{Table 2 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the \texttt{cl::Context::getInfo} functions.}

Last Revision Date: 6/14/2010
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

```cpp
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

\[
\text{template <typename } T \text{>}
\]
\[
\text{cl_int cl::Memory::getInfo(cl\_context\_info name,}
\]
\[
\text{ \quad T \ast param)}
\]

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.

<table>
<thead>
<tr>
<th>cl_memory_info</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_MEM_CONTEXT</td>
<td>cl::Context</td>
</tr>
</tbody>
</table>

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.

\textbf{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

\[
\text{template <cl\_int } name \text{> typename}
\]
\[
\text{detail::param\_traits<detail::cl\_context\_info, name>::param\_type}
\]
\[
\text{cl::Memory::getInfo(void)}
\]

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.

\textbf{cl::Memory::getInfo} returns the appropriate value for a given \( name \) as specified in table 5.9.

The method

\[
\text{cl\_int cl::Memory::setDestructorCallback(}
\]
\[
\quad \text{void (CL\_CALLBACK \ast pfn\_notify)(cl\_mem memobj,}
\]
\[
\quad \text{ \quad \text{void \ast user\_data),}
\]
\[
\quad \text{void \ast user\_data = NULL)}
\]

\textsuperscript{4} Table 3 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the \textbf{cl::Memory::getInfo} functions.

Last Revision Date: 6/14/2010
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```cpp
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.

class 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\(^5\) 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 \(>= width \times size\) 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 \(\times size\) of element in bytes. If row_pitch is not 0, it must be a multiple of the image element size in bytes.

host_ptr is a pointer to the image data that may already be allocated by the application. The size of the buffer that host_ptr points to must be \(>= row_pitch \times height\). 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.

\(^5\) 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

```cpp
cl::Image3D 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\(^6\) 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.

\(^6\) `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 \( \geq \) row_pitch \( \times \) height if host_ptr is not NULL. If host_ptr is not NULL and slice_pitch = 0, slice_pitch is calculated as row_pitch \( \times \) 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 \( \geq \) slice_pitch \( \times \) 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

```cpp
class 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

```cpp
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.

<table>
<thead>
<tr>
<th>cl_sampler_info</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_SAMPLER_CONTEXT</td>
<td>cl::Context</td>
</tr>
</tbody>
</table>

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 specifieds in table 5.10.

---

Table 4 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the cl::Sampler::getInfo functions.

Last Revision Date: 6/14/2010
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

```cpp
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

```cpp
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

```cpp
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 5.8

<table>
<thead>
<tr>
<th>cl_program_info</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_PROGRAM_CONTEXT</td>
<td>cl::Context</td>
</tr>
<tr>
<td>CL_PROGRAM_DEVICES</td>
<td>VECTOR_CLASS&lt;cl_device_id&gt;</td>
</tr>
<tr>
<td>CL_PROGRAM_SOURCE</td>
<td>STRING_CLASS</td>
</tr>
<tr>
<td>CL_PROGRAM_BINARY_SIZES</td>
<td>VECTOR_CLASS&lt;::size_t&gt;</td>
</tr>
<tr>
<td>CL_PROGRAM_BINARIES</td>
<td>VECTOR_CLASS&lt;char *&gt;</td>
</tr>
</tbody>
</table>

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

```cpp
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.

---

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

Last Revision Date: 6/14/2010
The method

\[
\text{template <typename } T \text{>}
\]

\[
\text{cl_int cl::Program::getBuildInfo(cl_program_build_info name,}
\]

\[
T * \text{param)}
\]

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\(^9\)

<table>
<thead>
<tr>
<th>cl_program_info</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_PROGRAM_BUILD_OPTIONS</td>
<td>STRING_CLASS</td>
</tr>
<tr>
<td>CL_PROGRAM_BUILD_LOG</td>
<td>STRING_CLASS</td>
</tr>
</tbody>
</table>

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.

\text{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

\[
\text{template <cl_int name> typename}
\]

\[
detail::param_traits<detail::cl_program_info, name>::param_type
\]

\text{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.

\text{cl::Program::getBuildInfo} returns the appropriate value for a given \(name\) as specified in table 5.12.

The method

\[
\text{cl_int cl::Program::createKernels(const VECTOR\_CLASS<Kernel> * kernels)}
\]

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

\(^9\) Table 5 reflects differences in return types between the OpenCL C API and the OpenCL C++ API for the \text{cl::Program::getBuildInfo} functions.
kernels is a memory pointer to a vector where the kernel objects for kernels in the program will be returned.

c::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

```cpp
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

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

Last Revision Date: 6/14/2010
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.

<table>
<thead>
<tr>
<th>cl_kernel_info</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_KERNEL_FUNCTION_NAME</td>
<td>STRING_CLASS</td>
</tr>
<tr>
<td>CL_KERNEL_CONTEXT</td>
<td>cl::Context</td>
</tr>
<tr>
<td>CL_KERNEL_PROGRAM</td>
<td>cl::Program</td>
</tr>
</tbody>
</table>

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

```cpp
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

```cpp
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.

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

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

Last Revision Date: 6/14/2010
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

```cpp
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

```cpp
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**\(^\text{13}\), which corresponds to an argument of **__local** in the kernel object.

---

\(^{12}\) **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.

Last Revision Date: 6/14/2010
• 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

```cpp
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.

<table>
<thead>
<tr>
<th><code>cl_event_info</code></th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_EVENT_CONTEXT</td>
<td>Cl::Context</td>
</tr>
<tr>
<td>CL_EVENT_COMMAND_QUEUE</td>
<td>cl::CommandQueue</td>
</tr>
</tbody>
</table>

*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.

---

13 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.

14 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

```cpp
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

```cpp
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

```cpp
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.
**cl::Event::getProfilingInfo** returns the appropriate value for a given *name* as specified in table 5.16. The method

```cpp
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

```cpp
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:

---

Last Revision Date: 6/14/2010
- CL_INVALID_VALUE if `pfEvent_ptr` 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

```cpp
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

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

The method

```cpp
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

```cpp
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

```cpp
cl::NDRange::NDRange (::size_t size0, 
                      ::size_t size1)
```

returns a 2D range.
size0 describes the number of global or local work-items in dimension 0.

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

The constructor

\[
\text{cl::NDRange::NDRange}(\text{::size}_t \text{ size0}, \\
\text{::size}_t \text{ size1}, \\
\text{::size}_t \text{ size2})
\]

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

\[
\text{operator const ::size}_t \text{ cl::NDRange::*()} \text{ const}
\]

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

The method

\[
\text{::size}_t \text{ cl::NDRange::dimensions}() \text{ void}
\]

returns the number of dimensions defined in the range.

The class cl::CommandQueue provides functionality for working with OpenCL command-queues.

The constructor

\[
\text{cl::CommandQueue::CommandQueue}(
\text{ const Context& context,} \\
\text{ const Device& device,} \\
\text{ cl\_command\_queue\_properties properties = 0,} \\
\text{ cl\_int * err = NULL})
\]

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.
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

```cpp
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 1015.

<table>
<thead>
<tr>
<th>cl_command_queue_info</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_QUEUE_CONTEXT</td>
<td>cl::Context</td>
</tr>
<tr>
<td>CL_QUEUE_DEVICE</td>
<td>cl::Device</td>
</tr>
</tbody>
</table>

**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

---
15 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 CLDEVICE_MEM_BASE_ADDR_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

```c
cl_int cl::CommandQueue::enqueueReadBufferRect(
    const Buffer& buffer,
    cl_bool blocking_read,
    const size_t<3> buffer_offset,
    const size_t<3> host_offset,
```

Last Revision Date: 6/14/2010
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::CommandQueue::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::CommandQueue::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::CommandQueue::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.

Last Revision Date: 6/14/2010
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.

Last Revision Date: 6/14/2010
- 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::CommandQueue::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.
The method

```cpp
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`.

Last Revision Date: 6/14/2010
$dst_row_pitch$ 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 $region[1] \times dst_row_pitch$.

$events$ 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.

$\text{cl::CommandQueue::enqueueCopyBufferRect}$ 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

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

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

enqueue 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.

Last Revision Date: 6/14/2010
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.

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

```cpp
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:

Last Revision Date: 6/14/2010
The method

\[
\text{cl_int cl::CommandQueue::enqueueCopyImageToBuffer(}
\begin{array}{l}
\text{const Image& src\_image,} \\
\text{const Buffer& dst\_buffer,} \\
\text{const size_t<3>& src\_origin,} \\
\text{const size_t<3>& region,} \\
\text{const size_t dst\_offset,} \\
\text{const VECTOR\_CLASS<Event> * events = NULL,} \\
\text{Event * event = NULL)}
\end{array}
\]

enqueues a command to copy an image object to a buffer object. The OpenCL context associated with the command-queue, \textit{src\_image} and \textit{dst\_buffer} must be the same.

\textit{src\_image} is a valid image object.

\textit{dst\_buffer} is a valid buffer object.

\textit{src\_origin} defines the (x, y, z) offset in pixels in the image from where to copy. If \textit{src\_image} is a 2D image object and \textit{src\_origin}[2] is not equal to 0 or \textit{region}[2] is not equal to 1.

\textit{CL\_INVALID\_CONTEXT} if the context associated with the command-queue, \textit{src\_image} and \textit{dst\_image} are not the same or if the context associated with the command-queue and events in \textit{events} are not the same.

\textit{CL\_INVALID\_MEM\_OBJECT} if \textit{src\_image} and \textit{dst\_image} are not valid image objects.

\textit{CL\_IMAGE\_FORMAT\_MISMATCH} if \textit{src\_image} and \textit{dst\_image} do not use the same image format.

\textit{CL\_INVALID\_VALUE} if the 2D or 3D rectangular region specified by \textit{src\_origin} and \textit{src\_origin} + \textit{region} refers to a region outside \textit{src\_image}, or if the 2D or 3D rectangular region specified by \textit{dst\_origin} and \textit{dst\_origin} + \textit{region} refers to a region outside \textit{dst\_image}.

\textit{CL\_INVALID\_VALUE} if \textit{src\_image} is a 2D image object and \textit{src\_origin}[2] is not equal to 0 or \textit{region}[2] is not equal to 1.

\textit{CL\_INVALID\_VALUE} if \textit{dst\_image} is a 2D image object and \textit{dst\_origin}[2] is not equal to 0 or \textit{region}[2] is not equal to 1.

\textit{CL\_INVALID\_EVENT\_WAIT\_LIST} if event objects in \textit{events} are not valid events.

\textit{CL\_INVALID\_IMAGE\_SIZE} if image dimensions (image width, height, specified or compute row and/or slice pitch) for \textit{src\_image} are not supported by device associated with queue.

\textit{CL\_INVALID\_IMAGE\_SIZE} if image dimensions (image width, height, specified or compute row and/or slice pitch) for \textit{dst\_image} are not supported by device associated with queue.

\textit{CL\_MEM\_OBJECT\_ALLOCATION\_FAILURE} if there is a failure to allocate memory for data store associated with \textit{src\_image} or \textit{dst\_image}.

\textit{CL\_OUT\_OF\_RESOURCES} if there is a failure to allocate resources required by the OpenCL implementation on the device.

\textit{CL\_OUT\_OF\_HOST\_MEMORY} if there is a failure to allocate resources required by the OpenCL implementation on the host.
2D image object, the z value given by \textit{src\_origin}[2] must be 0.

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

\textit{dst\_offset} refers to the offset where to begin copying data into \textit{dst\_buffer}. The size in bytes of the region to be copied referred to as \textit{dst\_cb} is computed as width * height * depth * bytes/image element if \textit{src\_image} is a 3D image object and is computed as width * height * bytes/image element if \textit{src\_image} is a 2D image object.

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

\textit{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. \textit{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.

\textit{cl::CommandQueue::enqueueCopyImageToBuffer} returns \text{CL\_SUCCESS} if the function is executed successfully. Otherwise, it returns one of the following errors:

- \text{CL\_INVALID\_CONTEXT} if the context associated with the command-queue, \textit{src\_image} and \textit{dst\_buffer} are not the same or if the context associated with the command-queue and events in \textit{events} are not the same.
- \text{CL\_INVALID\_MEM\_OBJECT} if \textit{src\_image} is not a valid image object or \textit{dst\_buffer} is not a valid buffer object.
- \text{CL\_INVALID\_VALUE} if the 2D or 3D rectangular region specified by \textit{src\_origin} and \textit{src\_origin} + \textit{region} refers to a region outside \textit{src\_image}, or if the region specified by \textit{dst\_offset} and \textit{dst\_offset} + \textit{dst\_cb} to a region outside \textit{dst\_buffer}.
- \text{CL\_INVALID\_VALUE} if \textit{src\_image} is a 2D image object and \textit{src\_origin}[2] is not equal to 0 or \textit{region}[2] is not equal to 1.
- \text{CL\_INVALID\_EVENT\_WAIT\_LIST} if event objects in \textit{events} are not valid events.
- \text{CL\_MISALIGNED\_SUB\_BUFFER\_OFFSET} if \textit{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. \text{CL\_INVALID\_IMAGE\_SIZE} if image dimensions (image width, height, specified or compute row and/or slice pitch) for \textit{src\_image} are not supported by device associated with queue.
- \text{CL\_MEM\_OBJECT\_ALLOCATION\_FAILURE} if there is a failure to allocate memory for data store associated with \textit{src\_image} or \textit{dst\_buffer}.
- \text{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).
- \text{CL\_OUT\_OF\_RESOURCES} if there is a failure to allocate resources required by the OpenCL implementation on the device.
- \text{CL\_OUT\_OF\_HOST\_MEMORY} if there is a failure to allocate resources required by the OpenCL implementation on the host.

The method

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

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

src_buffer is a valid buffer object.

dst_image is a valid image object.

src_offset refers to the offset where to begin copying data from src_buffer.

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

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

events specifies events that need to complete before this particular command can be executed. If 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

```cpp
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.

Last Revision Date: 6/14/2010
**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 \((\text{offset}, \text{size})\) 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

```cpp
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.

Last Revision Date: 6/14/2010
• 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

```cpp
c_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).

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

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

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

local can also be a cl::NULLRange value in which case the OpenCL implementation will determine how to 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:

Last Revision Date: 6/14/2010
• 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

```c
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.

Last Revision Date: 6/14/2010
- 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

```cpp
c_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. \textit{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. \texttt{cl::CommandQueue::enqueueBarrier} can be used instead.

\textbf{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 \textit{command queue} and events in \textit{events} are not the same.
- CL\_INVALID\_VALUE if \textit{user_func} is NULL.
- CL\_INVALID\_VALUE if \textit{args.fst} is a NULL value and \textit{args.snd} > 0, or if \textit{args.fst} is a NULL value and then length of \textit{mem_objects} > 0.
- CL\_INVALID\_VALUE if \textit{args.fst} is not NULL and \textit{args.snd} is 0.
- CL\_INVALID\_VALUE if the length of \textit{mem_objects} > 0 and \textit{mem_locs} is NULL.
- CL\_INVALID\_VALUE if length of \textit{mem_objects} is 0 and \textit{mem_locs} is not NULL.
- CL\_INVALID\_OPERATION if \textit{device} cannot execute the native kernel.
- CL\_INVALID\_MEM\_OBJECT if one or more memory objects specified in \textit{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 \textit{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 \textit{kernel}.
- CL\_INVALID\_EVENT\_WAIT\_LIST if event objects in \textit{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 \textit{commandqueue}. The marker command returns an \textit{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.

\textbf{Cl::CommandQueue::enqueueMarker} returns CL\_SUCCESS if the function is successfully executed. Otherwise, it returns one of the following errors:

- CL\_INVALID\_VALUE if \textit{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 \textit{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)

Last Revision Date: 6/14/2010
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 be optional. In the case that exceptions are not used the resulting application must compile and work without exception support.

By default C++ exceptions are not enabled and the OpenCL error code is returned, or set, as per the underlying C API.

To 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.

<table>
<thead>
<tr>
<th>Preprocessor macro name</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>__GET_DEVICE_INFO_ERR</td>
<td>“clGetDeviceInfo”</td>
</tr>
<tr>
<td>__GET_PLATFORM_INFO_ERR</td>
<td>“clGetPlatformInfo”</td>
</tr>
<tr>
<td>__GET_DEVICE_IDS_ERR</td>
<td>“clGetDeviceIds”</td>
</tr>
<tr>
<td>__GET_CONTEXT_INFO_ERR</td>
<td>“clGetContextInfo”</td>
</tr>
<tr>
<td>__GET_EVENT_INFO_ERR</td>
<td>“clGetEventInfo”</td>
</tr>
<tr>
<td>__GET_EVENT_PROFILE_INFO_ERR</td>
<td>“clGetEventProfileInfo”</td>
</tr>
<tr>
<td>__GET_MEM_OBJECT_INFO_ERR</td>
<td>“clGetMemObjectInfo”</td>
</tr>
<tr>
<td>__GET_IMAGE_INFO_ERR</td>
<td>“clGetImageInfo”</td>
</tr>
<tr>
<td>__GET_SAMPLER_INFO_ERR</td>
<td>“clGetSamplerInfo”</td>
</tr>
<tr>
<td>__GET_KERNEL_INFO_ERR</td>
<td>“clGetKernelInfo”</td>
</tr>
<tr>
<td>__GET_KERNEL_WORK_GROUP_INFO_ERR</td>
<td>“clGetKernelWorkGroupInfo”</td>
</tr>
<tr>
<td>Macro Name</td>
<td>Function Name</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>__GET_PROGRAM_INFO_ERR</td>
<td><code>clGetProgramInfo</code></td>
</tr>
<tr>
<td>__GET_PROGRAM_BUILD_INFO_ERR</td>
<td><code>clGetProgramBuildInfo</code></td>
</tr>
<tr>
<td>__GET_COMMAND_QUEUE_INFO_ERR</td>
<td><code>clGetCommandQueueInfo</code></td>
</tr>
<tr>
<td>__CREATE_CONTEXT_FROM_TYPE_ERR</td>
<td><code>clCreateContextFromType</code></td>
</tr>
<tr>
<td>__GET_SUPPORTED_IMAGE_FORMATS_ERR</td>
<td><code>clGetSupportedImageFormats</code></td>
</tr>
<tr>
<td>__CREATE_BUFFER_ERR</td>
<td><code>clCreateBuffer</code></td>
</tr>
<tr>
<td>__CREATE_SUBBUFFER_ERR</td>
<td><code>clCreateSubBuffer</code></td>
</tr>
<tr>
<td>__CREATE_GL_BUFFER_ERR</td>
<td><code>clCreateGLBuffer</code></td>
</tr>
<tr>
<td>__CREATE_IMAGE2D_ERR</td>
<td><code>clCreateImage2D</code></td>
</tr>
<tr>
<td>__CREATE_IMAGE3D_ERR</td>
<td><code>clCreateImage3D</code></td>
</tr>
<tr>
<td>__SET_MEM_OBJECTDestructor_CALLBACK_ERR</td>
<td><code>clSetMemObjectDestructorCallback</code></td>
</tr>
<tr>
<td>__CREATE_USER_EVENT_ERR</td>
<td><code>clCreateUserEvent</code></td>
</tr>
<tr>
<td>__SET_USER_EVENT_STATUS_ERR</td>
<td><code>clSetUserEventStatus</code></td>
</tr>
<tr>
<td>__SET_EVENT_CALLBACK_ERR</td>
<td><code>clSetEventCallback</code></td>
</tr>
<tr>
<td>__WAIT_FOR_EVENTS_ERR</td>
<td><code>clWaitForEvents</code></td>
</tr>
<tr>
<td>__CREATE_KERNEL_ERR</td>
<td><code>clCreateKernel</code></td>
</tr>
<tr>
<td>__SET_KERNEL_ARGS_ERR</td>
<td><code>clSetKernelArgs</code></td>
</tr>
<tr>
<td>__CREATE_PROGRAM_WITH_SOURCE_ERR</td>
<td><code>clCreateProgramWithSource</code></td>
</tr>
<tr>
<td>__CREATE_PROGRAM_WITH_BINARY_ERR</td>
<td><code>clCreateProgramWithBinary</code></td>
</tr>
<tr>
<td>__BUILD_PROGRAM_ERR</td>
<td><code>clBuildProgram</code></td>
</tr>
<tr>
<td>__CREATE_KERNELS_IN_PROGRAM_ERR</td>
<td><code>clCreateKernelsInProgram</code></td>
</tr>
<tr>
<td>__CREATE_COMMAND_QUEUE_ERR</td>
<td><code>clCreateCommandQueue</code></td>
</tr>
<tr>
<td>__SET_COMMAND_QUEUE_PROPERTY_ERR</td>
<td><code>clSetCommandQueueProperty</code></td>
</tr>
<tr>
<td>__ENQUEUE_READ_BUFFER_ERR</td>
<td><code>clEnqueueReadBuffer</code></td>
</tr>
<tr>
<td>__ENQUEUE_READ_BUFFER RECT_ERR</td>
<td><code>clEnqueueReadBufferRect</code></td>
</tr>
<tr>
<td>__ENQUEUE_WRITE_BUFFER_ERR</td>
<td><code>clEnqueueWriteBuffer</code></td>
</tr>
<tr>
<td>__ENQUEUE_WRITE_BUFFER RECT_ERR</td>
<td><code>clEnqueueWriteBufferRect</code></td>
</tr>
<tr>
<td>__ENQUEUE_COPY_BUFFER_ERR</td>
<td><code>clEnqueueCopyBuffer</code></td>
</tr>
<tr>
<td>__ENQUEUE_COPY_BUFFER RECT_ERR</td>
<td><code>clEnqueueCopyBufferRect</code></td>
</tr>
<tr>
<td>__ENQUEUE_READ_IMAGE_ERR</td>
<td><code>clEnqueueReadImage</code></td>
</tr>
<tr>
<td>__ENQUEUE_WRITE_IMAGE_ERR</td>
<td><code>clEnqueueWriteImage</code></td>
</tr>
<tr>
<td>__ENQUEUE_COPY_IMAGE_ERR</td>
<td><code>clEnqueueCopyImage</code></td>
</tr>
<tr>
<td>__ENQUEUE_COPY_IMAGE TO BUFFER_ERR</td>
<td><code>clEnqueueCopyImageToBuffer</code></td>
</tr>
<tr>
<td>__ENQUEUE_COPY_BUFFER TO IMAGE_ERR</td>
<td><code>clEnqueueCopyBufferToImage</code></td>
</tr>
<tr>
<td>__ENQUEUE_MAP_BUFFER_ERR</td>
<td><code>clEnqueueMapBuffer</code></td>
</tr>
<tr>
<td>__ENQUEUE_MAP_IMAGE_ERR</td>
<td><code>clEnqueueMapImage</code></td>
</tr>
<tr>
<td>__ENQUEUE_UNMAP_MEM_OBJECT_ERR</td>
<td><code>clEnqueueUnmapMemObject</code></td>
</tr>
<tr>
<td>__ENQUEUE_NDRange KERNEL_ERR</td>
<td><code>clEnqueueNDRangeKernel</code></td>
</tr>
<tr>
<td>__ENQUEUE_TASK_ERR</td>
<td><code>clEnqueueTask</code></td>
</tr>
<tr>
<td>__ENQUEUE_NATIVE KERNEL</td>
<td><code>clEnqueueNativeKernel</code></td>
</tr>
<tr>
<td>__ENQUEUE_MARKER_ERR</td>
<td><code>clEnqueueMarker</code></td>
</tr>
<tr>
<td>__ENQUEUE_WAIT FOR EVENTS_ERR</td>
<td><code>clEnqueueWaitForEvents</code></td>
</tr>
<tr>
<td>__ENQUEUE_BARRIER ERR</td>
<td><code>clEnqueueBarriers</code></td>
</tr>
<tr>
<td>__UNLOAD_COMPILER_ERR</td>
<td><code>clUnloadCompiler</code></td>
</tr>
<tr>
<td>__FLUSH_ERR</td>
<td><code>clFlush</code></td>
</tr>
<tr>
<td>__FINISH_ERR</td>
<td><code>clFinish</code></td>
</tr>
</tbody>
</table>

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\(^\text{16}\).

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

\(^{16}\text{C++ does not currently support typedef templates and thus the vector type must be given by its name only through the preprocessor macro VECTOR_CLASS.}\)

Last Revision Date: 6/14/2010
change. A developer can provide a replacement implementation for std::string by defining the preprocessor macro

```cpp
__USE_DEV_STRING
```

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

```cpp
typedef stringType STRING_CLASS
```

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