The **OpenCL Extension** Specification

*Version: 2.1*

*Document Revision: 4*

Khronos OpenCL Working Group

*Editor: Aaftab Munshi*
9. OPTIONAL EXTENSIONS ..................................................................................................................................... 4

9.1 Compiler Directives for Optional Extensions ...........................................................................................................5

9.2 Getting OpenCL API Extension Function Pointers .......................................................................................................6

9.3 Availability of OpenCL 2.0 KHR Extensions ...................................................................................................................8

9.4 Sharing of CL / GL MSAA Textures ...............................................................................................................................8
   9.4.1 Additions to Chapter 9.7 of the OpenCL 2.0 Extension Specification ..............................................................8
   9.4.2 Additions to Chapter 5 of the OpenCL 2.0 Specification ..................................................................................9
   9.4.3 Additions to Chapter 6 of the OpenCL C Specification ...................................................................................9
   9.4.4 Additions to Chapter 1 of the OpenCL C++ Specification .................................................................................10
   9.4.5 Add the following built-in functions to section 6.13.14.3 – BuiltIn Image Read Functions ......................10
   9.4.6 Add the following built-in functions to section 2.12.3 of OpenCL C++ Specification ...........................13
   9.4.7 Add the following to both section 6.13.14.3 of OpenCL C Specification and section 2.12.3 of OpenCL
       C++ Specification ..................................................................................................................................................19
   9.4.8 Add the following built-in functions to section 6.13.14.5 of OpenCL C Specification ......................20
   9.4.9 Add the following built-in functions to section 2.12.6 of OpenCL C++ specification ...........................21

9.5 Subgroup Independent Forward Progress ..................................................................................................................23
   9.5.1.1 Additions to section 3.2 – Execution Model .................................................................................................23

9.6 Named barriers for subgroups ......................................................................................................................................24
   9.6.1 Additions to the OpenCL C++ Language ...........................................................................................................24
   9.6.1.1 Additions to section 1.1.3 Other Built-in Data Types .................................................................................24
   9.6.1.2 Add a new section 2.14 Named Barrier Functions .......................................................................................25

9.7 Priority Hints .........................................................................................................................................................25
   9.7.1 Host-side API modifications ............................................................................................................................26

9.8 Throttle Hints .......................................................................................................................................................26
   9.8.1 Host-side API modifications ............................................................................................................................26

INDEX - APIS ......................................................................................................................................................28
9. Optional Extensions

This document describes the list of optional features supported by OpenCL 2.1. Optional extensions may be supported by some OpenCL devices. Optional extensions are not required to be supported by a conformant OpenCL implementation, but are expected to be widely available; they define functionality that is likely to move into the required feature set in a future revision of the OpenCL specification. A brief description of how OpenCL extensions are defined is provided below.

For OpenCL extensions approved by the OpenCL working group, the following naming conventions are used:

- A unique name string of the form "cl_khr_<name>" is associated with each extension. If the extension is supported by an implementation, this string will be present in the CL_PLATFORM_EXTENSIONS string defined in table 4.1 or CL_DEVICE_EXTENSIONS string described in table 4.3.

- All API functions defined by the extension will have names of the form cl<FunctionName>KHR.

- All enumerants defined by the extension will have names of the form CL_<enum_name>_KHR.

OpenCL extensions approved by the OpenCL working group can be promoted to required core features in later revisions of OpenCL. When this occurs, the extension specifications are merged into the core specification. Functions and enumerants that are part of such promoted extensions will have the KHR affix removed. OpenCL implementations of such later revisions must also export the name strings of promoted extensions in the CL_PLATFORM_EXTENSIONS or CL_DEVICE_EXTENSIONS string, and support the KHR-affixed versions of functions and enumerants as a transition aid.

For vendor extensions, the following naming conventions are used:

- A unique name string of the form "cl_<vendor_name>_<name>" is associated with each extension. If the extension is supported by an implementation, this string will be present in the CL_PLATFORM_EXTENSIONS string described in table 4.1 or CL_DEVICE_EXTENSIONS string described in table 4.3.

- All API functions defined by the vendor extension will have names of the form cl<FunctionName><vendor_name>.

---

1 This document describes section 9 of the OpenCL 2.0 specification. Any reference to section 1.x – 8.x or tables 1.x – 8.x in this document refer to sections and tables described in the OpenCL 2.0 API and OpenCL C specifications.
All enumerants defined by the vendor extension will have names of the form CL_<enum_name>_<vendor_name>.

## 9.1 Compiler Directives for Optional Extensions

The `#pragma OPENCL EXTENSION` directive controls the behavior of the OpenCL compiler with respect to extensions. The `#pragma OPENCL EXTENSION` directive is defined as:

```c
#pragma OPENCL EXTENSION extension_name : behavior
#pragma OPENCL EXTENSION all : behavior
```

where `extension_name` is the name of the extension. The `extension_name` will have names of the form `cl_khr_<name>` for an extension approved by the OpenCL working group and will have names of the form `cl_<vendor_name>_<name>` for vendor extensions. The token `all` means that the behavior applies to all extensions supported by the compiler. The `behavior` can be set to one of the following values given by the table below.

<table>
<thead>
<tr>
<th>behavior</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable</td>
<td>Behave as specified by the extension <code>extension_name</code>. Report an error on the <code>#pragma OPENCL EXTENSION</code> if the <code>extension_name</code> is not supported, or if <code>all</code> is specified.</td>
</tr>
<tr>
<td>disable</td>
<td>Behave (including issuing errors and warnings) as if the extension <code>extension_name</code> is not part of the language definition. If <code>all</code> is specified, then behavior must revert back to that of the non-extended core version of the language being compiled to. Warn on the <code>#pragma OPENCL EXTENSION</code> if the extension <code>extension_name</code> is not supported.</td>
</tr>
</tbody>
</table>

The `#pragma OPENCL EXTENSION` directive is a simple, low-level mechanism to set the behavior for each extension. It does not define policies such as which combinations are appropriate; those must be defined elsewhere. The order of directives matter in setting the behavior for each extension. Directives that occur later override those seen earlier. The `all` variant sets the behavior for all extensions, overriding all previously issued extension directives, but only if the `behavior` is set to `disable`.

The initial state of the compiler is as if the directive

```c
#pragma OPENCL EXTENSION all : disable
```

was issued, telling the compiler that all error and warning reporting must be done according to this specification, ignoring any extensions.
Every extension which affects the OpenCL language semantics, syntax or adds built-in functions to the language must create a preprocessor `#define` that matches the extension name string. This `#define` would be available in the language if and only if the extension is supported on a given implementation.

**Example:**

An extension which adds the extension string "\texttt{cl_khr_3d_image_writes}" should also add a preprocessor `#define` called `cl_khr_3d_image_writes`. A kernel can now use this preprocessor `#define` to do something like:

```c
#ifdef cl_khr_3d_image_writes
    // do something using the extension
#else
    // do something else or #error!
#endif
```

### 9.2 Getting OpenCL API Extension Function Pointers

The function

```c
void* clGetExtensionFunctionAddressForPlatform2(
    cl_platform_id platform,
    const char *funcname)
```

returns the address of the extension function named by `funcname` for a given `platform`. The pointer returned should be cast to a function pointer type matching the extension function’s definition defined in the appropriate extension specification and header file. A return value of `NULL` indicates that the specified function does not exist for the implementation or `platform` is not a valid platform. A non-NULL return value for `clGetExtensionFunctionAddressForPlatform` does not guarantee that an extension function is actually supported by the platform. The application must also make a corresponding query using `clGetPlatformInfo(platform, CL_PLATFORM_EXTENSIONS, …)` or `clGetDeviceInfo(device, CL_DEVICE_EXTENSIONS, …)` to determine if an extension is supported by the OpenCL implementation.

`clGetExtensionFunctionAddressForPlatform` may not be queried for core (non-extension) functions in OpenCL. For functions that are queryable with

\footnote{Since there is no way to qualify the query with a device, the function pointer returned must work for all implementations of that extension on different devices for a platform. The behavior of calling a device extension function on a device not supporting that extension is undefined.}
clGetExtensionFunctionAddressForPlatform, implementations may choose to also export those functions statically from the object libraries implementing those functions. However, portable applications cannot rely on this behavior.

Function pointer typedefs must be declared for all extensions that add API entrypoints. These typedefs are a required part of the extension interface, to be provided in an appropriate header (such as cl_ext.h if the extension is an OpenCL extension, or cl_gl_ext.h if the extension is an OpenCL / OpenGL sharing extension).

The following convention must be followed for all extensions affecting the host API:

```c
#ifndef extension_name
#define extension_name 1

// all data typedefs, token #defines, prototypes, and
// function pointer typedefs for this extension

// function pointer typedefs must use the
// following naming convention
typedef CL_API_ENTRY return_type
(CL_API_CALL *cextension_func_nameTAG_fn)(...);
#endif // extension_name
```

where TAG can be KHR, EXT or vendor-specific.

Consider, for example, the cl_khr_gl_sharing extension. This extension would add the following to cl_gl_ext.h:

```c
#ifndef cl_khr_gl_sharing
#define cl_khr_gl_sharing 1

// all data typedefs, token #defines, prototypes, and
// function pointer typedefs for this extension
#define CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR -1000
#define CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR 0x2006
#define CL_DEVICES_FOR_GL_CONTEXT_KHR 0x2007
#define CL_GL_CONTEXT_KHR 0x2008
#define CL_EGL_DISPLAY_KHR 0x2009
#define CL_GLX_DISPLAY_KHR 0x200A
#define CL_WGL_HDC_KHR 0x200B
#define CL_CGL_SHAREGROUP_KHR 0x200C

// function pointer typedefs must use the
// following naming convention
typedef CL_API_ENTRY cl_int
(CL_API_CALL *clGetGLContextInfoKHR_fn)(
    const cl_context_properties * /* properties */,
    cl_gl_context_info /* param_name */,
    size_t /* param_value_size */,
...);```
void * /* param_value */,
size_t * /* param_value_size_ret */);

#endif // cl_khr_gl_sharing

9.3 Availability of OpenCL 2.0 KHR Extensions
This is described in the section F.4 of OpenCL API Specification.

9.4 Sharing of CL / GL MSAA Textures
This extension extends the CL / GL sharing (i.e. the cl_khr_gl_sharing_extension) defined in
section 9.7 to allow a CL image to be created from a GL multi-sampled (a.k.a. MSAA) texture
(color or depth).

This extension name is cl_khr_gl_msaa_sharing. This extension requires
cl_khr_gl_depth_images.

9.4.1 Additions to Chapter 9.7 of the OpenCL 2.0 Extension
Specification

Allow texture_target argument to clCreateFromGLTexture to be
GL_TEXTURE_2D_MULTISAMPLE or GL_TEXTURE_2D_MULTISAMPLE_ARRAY.

If texture_target is GL_TEXTURE_2D_MULTISAMPLE, clCreateFromGLTexture creates an
OpenCL 2D multi-sample image object from an OpenGL 2D multi-sample texture.

If texture_target is GL_TEXTURE_2D_MULTISAMPLE_ARRAY, clCreateFromGLTexture
creates an OpenCL 2D multi-sample array image object from an OpenGL 2D multi-sample
texture.

Multi-sample CL image objects can only be read from a kernel. Multi-sample CL image objects
cannot be used as arguments to clEnqueueReadImage, clEnqueueWriteImage,
clEnqueueCopyImage, clEnqueueCopyImageToBuffer, clEnqueueCopyBufferToImage,
clEnqueueMapImage and clEnqueueFillImage and will return a CL_INVALID_OPERATION
error.

Add the following entry to table 9.5:

<table>
<thead>
<tr>
<th>cl_gl_texture_info</th>
<th>Return Type</th>
<th>Info. returned in param_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_GL_NUM_SAMPLES</td>
<td>GLint</td>
<td>The samples argument passed to glTexImage2DMultisample or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>glTexImage3DMultisample.</td>
</tr>
</tbody>
</table>
**9.4.2 Additions to Chapter 5 of the OpenCL API Specification**

The formats described in tables 5.8.a and 5.8.b of the OpenCL 2.0 specification and the additional formats added to this table described in section 9.12.1 also support CL images created from a GL multi-sampled color or depth texture.

Update text that describes arg_value argument to clSetKernelArg with the following:

If the argument is a multi-sample 2D image, the arg_value entry must be a pointer to a multi-sample image object. If the argument is a multi-sample 2D depth image, the arg_value entry must be a pointer to a multisample depth image object. If the argument is a multi-sample 2D image array, the arg_value entry must be a pointer to a multi-sample image array object. If the argument is a multi-sample 2D depth image array, the arg_value entry must be a pointer to a multi-sample depth image array object.

Updated error code text for clSetKernelArg is:

Add the following text:

CL_INVALID_MEM_OBJECT for an argument declared to be a multi-sample image, multi-sample image array, multi-sample depth image or a multi-sample depth image array and the argument value specified in arg_value does not follow the rules described above for a depth memory object or memory array object argument.

**9.4.3 Additions to Chapter 6 of the OpenCL C Specification**

Add the following new data types to table 6.3 in section 6.1.3 of the OpenCL 2.0 specification

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image2d_msaa_t</td>
<td>A 2D multi-sample color image. Refer to section 6.13.14 for a detailed description of the built-in functions that use this type.</td>
</tr>
<tr>
<td>image2d_array_msaa_t</td>
<td>A 2D multi-sample color image array. Refer to section 6.13.14 for a detailed description of the built-in functions that use this type.</td>
</tr>
<tr>
<td>image2d_msaa_depth_t</td>
<td>A 2D multi-sample depth image. Refer to section 6.13.14 for a detailed description of the built-in functions that use this type.</td>
</tr>
</tbody>
</table>
image2d_array_msaa_depth_t | A 2D multi-sample depth image array. Refer to section 6.13.14 for a detailed description of the built-in functions that use this type.

### 9.4.4 Additions to Chapter 1 of the OpenCL C++ Specification

Add the following new data types to table 6.3 in section 6.1.3 of the OpenCL 2.0 specification

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image2d_msaa</td>
<td>A 2D multi-sample color image. Refer to section 2.12.3 for a detailed description of the built-in functions that use this type.</td>
</tr>
<tr>
<td>image2d_array_msaa</td>
<td>A 2D multi-sample color image array. Refer to section 2.12.3 for a detailed description of the built-in functions that use this type.</td>
</tr>
<tr>
<td>image2d_msaa_depth</td>
<td>A 2D multi-sample depth image. Refer to section 2.12.3 for a detailed description of the built-in functions that use this type.</td>
</tr>
<tr>
<td>image2d_array_msaa_depth</td>
<td>A 2D multi-sample depth image array. Refer to section 2.12.3 for a detailed description of the built-in functions that use this type.</td>
</tr>
</tbody>
</table>

### 9.4.5 Add the following built-in functions to section 6.13.14.3 – BuiltIn Image Read Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float4 read_imagef</td>
<td>Use the coordinate (coord.x, coord.y) and sample to do an element lookup in the 2D image object specified by image.</td>
</tr>
<tr>
<td></td>
<td>read_imagef returns floating-point values in the range [0.0 … 1.0] for image objects created with image_channel_data_type set to one of the pre-defined packed formats or CL_UNORM_INT8, or CL_UNORM_INT16.</td>
</tr>
<tr>
<td></td>
<td>read_imagef returns floating-point values in the range [-1.0 … 1.0] for image objects created with image_channel_data_type set to CL_SNORM_INT8,</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>int4 read_imagei</strong> (image2d_msaa_t image, int2 coord, int sample)</td>
<td>Use the coordinate (coord.x, coord.y) and sample to do an element lookup in the 2D image object specified by image. <strong>read_imagei</strong> and <strong>read_imageui</strong> return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. <strong>read_imagei</strong> can only be used with image objects created with <strong>image_channel_data_type</strong> set to one of the following values: CL_SIGNED_INT8, CL_SIGNED_INT16 and CL_SIGNED_INT32. If the <strong>image_channel_data_type</strong> is not one of the above values, the values returned by <strong>read_imagei</strong> are undefined. <strong>read_imageui</strong> can only be used with image objects created with <strong>image_channel_data_type</strong> set to one of the following values: CL_UNSIGNED_INT8, CL_UNSIGNED_INT16 and CL_UNSIGNED_INT32. If the <strong>image_channel_data_type</strong> is not one of the above values, the values returned by <strong>read_imageui</strong> are undefined.</td>
</tr>
<tr>
<td><strong>uint4 read_imageui</strong> (image2d_msaa_t image, int2 coord, int sample)</td>
<td>Use the coordinate (coord.x, coord.y) and sample to do an element lookup in the 2D image object specified by image. <strong>read_imagei</strong> and <strong>read_imageui</strong> return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. <strong>read_imagei</strong> can only be used with image objects created with <strong>image_channel_data_type</strong> set to one of the following values: CL_SIGNED_INT8, CL_SIGNED_INT16 and CL_SIGNED_INT32. If the <strong>image_channel_data_type</strong> is not one of the above values, the values returned by <strong>read_imagei</strong> are undefined. <strong>read_imageui</strong> can only be used with image objects created with <strong>image_channel_data_type</strong> set to one of the following values: CL_UNSIGNED_INT8, CL_UNSIGNED_INT16 and CL_UNSIGNED_INT32. If the <strong>image_channel_data_type</strong> is not one of the above values, the values returned by <strong>read_imageui</strong> are undefined.</td>
</tr>
<tr>
<td><strong>float4 read_imagef</strong> (image2d_array_msaa_t image, int4 coord, int sample)</td>
<td>Use coord.xy and sample to do an element lookup in the 2D image identified by coord.z in the 2D image array specified by image. <strong>read_imagef</strong> returns floating-point values in the range [0.0 … 1.0] for image objects created with <strong>image_channel_data_type</strong> set to one of the pre-</td>
</tr>
</tbody>
</table>
defined packed formats or CL_UNORM_INT8, or CL_UNORM_INT16.

**read_imagef** returns floating-point values in the range [-1.0 … 1.0] for image objects created with `image_channel_data_type` set to CL_SNORM_INT8, or CL_SNORM_INT16.

**read_imagef** returns floating-point values for image objects created with `image_channel_data_type` set to CL_HALF_FLOAT or CL_FLOAT.

Values returned by **read_imagef** for image objects with `image_channel_data_type` values not specified in the description above are undefined.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int4 read_imagei</strong> (</td>
<td></td>
</tr>
<tr>
<td>image2d_array_msaa_t image,</td>
<td></td>
</tr>
<tr>
<td>int4 coord,</td>
<td></td>
</tr>
<tr>
<td>int sample)</td>
<td></td>
</tr>
<tr>
<td><strong>uint4 read_imageui</strong> (</td>
<td></td>
</tr>
<tr>
<td>image2d_array_msaa_t image,</td>
<td></td>
</tr>
<tr>
<td>int4 coord,</td>
<td></td>
</tr>
<tr>
<td>int sample)</td>
<td></td>
</tr>
<tr>
<td><strong>float read_imagef</strong> (</td>
<td></td>
</tr>
<tr>
<td>Use the coordinate (coord.x, coord.y) and sample to</td>
<td></td>
</tr>
</tbody>
</table>
image2d_msaa_depth_t image,
int2 coord,
int sample)

do an element lookup in the 2D depth image object specified by image.

**read_imagef**

returns a floating-point value in the range [0.0 … 1.0] for depth image objects created with *image_channel_data_type* set to CL_UNORM_INT16 or CL_UNORM_INT24.

**read_imagef**

returns a floating-point value for depth image objects created with *image_channel_data_type* set to CL_FLOAT.

Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

float read_imagef (  
image2d_array_msaa_depth_t image,
int4 coord,
int sample)

Use *coord.xy* and *sample* to do an element lookup in the 2D image identified by *coord.z* in the 2D depth image array specified by *image*.

**read_imagef**

returns a floating-point value in the range [0.0 … 1.0] for depth image objects created with *image_channel_data_type* set to CL_UNORM_INT16 or CL_UNORM_INT24.

**read_imagef**

returns a floating-point value for depth image objects created with *image_channel_data_type* set to CL_FLOAT.

Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

### 9.4.6 Add the following built-in functions to section 2.12.3 of OpenCL C++ Specification

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>

Last Revision Date: 1/29/15
T image2d_msaa<T>::sample ( int2 coord, int sample)

Use the coordinate (coord.x, coord.y) and sample to do an element lookup in the 2D image object specified by image.

If image_channel_type::float32 returns floating-point values in the range [0.0 … 1.0] for image objects created with image_channel_data_type set to one of the pre-defined packed formats or CL_UNORM_INT8, or CL_UNORM_INT16.

If image_channel_type::float32 returns floating-point values in the range [-1.0 … 1.0] for image objects created with image_channel_data_type set to CL_SNORM_INT8, or CL_SNORM_INT16.

If image_channel_type::float32 returns floating-point values for image objects created with image_channel_data_type set to CL_HALF_FLOAT or CL_FLOAT.

Values returned by If image_channel_type::float32 for image objects with image_channel_data_type values not specified in the description above are undefined.

Use the coordinate (coord.x, coord.y) and sample to do an element lookup in the 2D image object specified by image.

If image_channel_type::sint32 and image_channel_type::uint32 return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer.

If image_channel_type::sint32 can only be used with image objects created with image_channel_data_type set to one of the following values: CL_SIGNED_INT8, CL_SIGNED_INT16 and CL_SIGNED_INT32.

If the image_channel_type is not one of the above values, the values returned are undefined.

If image_channel_type::sint32 can only be used with image objects created with...
image_channel_data_type set to one of the following values:
CL_UNSIGNED_INT8,
CL_UNSIGNED_INT16 and
CL_UNSIGNED_INT32.
If the image_channel_type is not one of the above values, the values returned are undefined.
T image2d_array_msaa<T>::sample(
    int4 coord,
    int sample)

Use `coord.xy` and `sample` to do an element lookup in the 2D image identified by `coord.z` in the 2D image array specified by `image`.

If `image_channel_type::float32` returns floating-point values in the range [0.0 … 1.0] for image objects created with `image_channel_data_type` set to one of the pre-defined packed formats or CL_UNORM_INT8, or CL_UNORM_INT16.

If `image_channel_type::float32` returns floating-point values in the range [-1.0 … 1.0] for image objects created with `image_channel_data_type` set to CL_SNORM_INT8, or CL_SNORM_INT16.

If `image_channel_type::float32` returns floating-point values for image objects created with `image_channel_data_type` set to CL_HALF_FLOAT or CL_FLOAT.

If `image_channel_type::float32` values returned by for image objects with `image_channel_type` values not specified in the description above are undefined. Use `coord.xy` and `sample` to do an element lookup in the 2D image identified by `coord.z` in the 2D image array specified by `image`.

If `image_channel_type::sint32` or `image_channel_type::uint32` return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer.

If `image_channel_type::sint32` can only be used with image objects created with `image_channel_data_type` set to one of the following values:
CL_SIGNED_INT8,
CL_SIGNED_INT16 and
CL_SIGNED_INT32.
If the `image_channel_type` is not one of the above values, the values returned by are undefined.

If `image_channel_type::uint32` can only be used with image objects created with `image_channel_data_type` set to one of the
following values:
CL_UNSIGNED_INT8,
CL_UNSIGNED_INT16 and
CL_UNSIGNED_INT32.
If `image_channel_type::sint32` and the
`image_data_type` is not one of the above values, the
values returned by are undefined.
| T image2d_msaa_depth<T>::sample( | Use the coordinate (coord.x, coord.y) and sample to do an element lookup in the 2D depth image object specified by image. |
| int2 coord, int sample) | If image_channel_type::float32 returns a floating-point value in the range [0.0 … 1.0] for depth image objects created with image_channel_data_type set to CL_UNORM_INT16 or CL_UNORM_INT24. |
| | If image_channel_type::float32 returns a floating-point value for depth image objects created with image_channel_data_type set to CL_FLOAT. |
| | If image_channel_type::float32, values returned by for image objects with image_channel_type values not specified in the description above are undefined. |

| T image2d_array_msaa_depth<T>::sample( | Use coord.xy and sample to do an element lookup in the 2D image identified by coord.z in the 2D depth image array specified by image. |
| int4 coord, int sample) | If image_channel_type::float32 returns a floating-point value in the range [0.0 … 1.0] for depth image objects created with image_channel_data_type set to CL_UNORM_INT16 or CL_UNORM_INT24. |
| | If image_channel_type::float32 returns a floating-point value for depth image objects created with image_channel_data_type set to CL_FLOAT. |
| | If image_channel_type::float32, values returned by for image objects with image_data_type values not specified in the description above are undefined. |

### 9.4.7 Add the following to both section 6.13.14.3 of OpenCL C Specification and section 2.12.3 of OpenCL C++ Specification

NOTE: When a multisample image is accessed in a kernel, the access takes one vector of integers describing which pixel to fetch and an integer corresponding to the sample numbers describing which sample within the pixel to fetch. sample identifies the sample position in the multi-sample image.
For best performance, we recommend that `sample` be a literal value so it is known at compile time and the OpenCL compiler can perform appropriate optimizations for multisample reads on the device.

No standard sampling instructions are allowed on the multisample image. Accessing a coordinate outside the image and/or a sample that is outside the number of samples associated with each pixel in the image is undefined.

### 9.4.8 Add the following built-in functions to section 6.13.14.5 of OpenCL C Specification

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int <code>get_image_width</code> (image2d_msaa_t image)</td>
<td>Return the image width in pixels.</td>
</tr>
<tr>
<td>int <code>get_image_width</code> (image2d_array_msaa_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_width</code> (image2d_msaa_depth_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_width</code> (image2d_array_msaa_depth_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_height</code> (image2d_msaa_t image)</td>
<td>Return the image height in pixels.</td>
</tr>
<tr>
<td>int <code>get_image_height</code> (image2d_array_msaa_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_height</code> (image2d_msaa_depth_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_height</code> (image2d_array_msaa_depth_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_channel_data_type</code> (image2d_msaa_t image)</td>
<td>Return the channel data type.</td>
</tr>
<tr>
<td>int <code>get_image_channel_data_type</code> (image2d_array_msaa_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_channel_data_type</code> (image2d_msaa_depth_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_channel_data_type</code> (image2d_array_msaa_depth_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_channel_order</code> (image2d_msaa_t image)</td>
<td>Return the image channel order.</td>
</tr>
<tr>
<td>int <code>get_image_channel_order</code> (image2d_array_msaa_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_channel_order</code> (image2d_msaa_depth_t image)</td>
<td></td>
</tr>
<tr>
<td>int <code>get_image_channel_order</code> (image2d_array_msaa_depth_t image)</td>
<td></td>
</tr>
</tbody>
</table>
9.4.9  Add the following built-in functions to section 2.12.6 of OpenCL C++ specification

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int image2d_msaa&lt;T&gt;::width ()</td>
<td>Return the image width in pixels.</td>
</tr>
<tr>
<td>int image2d_array_msaa&lt;T&gt;::width ()</td>
<td></td>
</tr>
<tr>
<td>int image2d_msaa_depth&lt;T&gt;::width ()</td>
<td></td>
</tr>
<tr>
<td>int image2d_array_msaa_depth&lt;T&gt;::width ()</td>
<td></td>
</tr>
<tr>
<td>int image2d_msaa&lt;T&gt;::height ()</td>
<td>Return the image height in pixels.</td>
</tr>
<tr>
<td>int image2d_array_msaa&lt;T&gt;::height ()</td>
<td></td>
</tr>
<tr>
<td>int image2d_msaa_depth&lt;T&gt;::height ()</td>
<td></td>
</tr>
<tr>
<td>int image2d_array_msaa_depth&lt;T&gt;::height ()</td>
<td></td>
</tr>
<tr>
<td>int image2d_msaa&lt;T&gt;::channel_data_type ()</td>
<td>Return the channel data type.</td>
</tr>
<tr>
<td>int image2d_array_msaa&lt;T&gt;::channel_data_type ()</td>
<td></td>
</tr>
<tr>
<td>int image2d_msaa_depth&lt;T&gt;::channel_data_type ()</td>
<td></td>
</tr>
<tr>
<td>int image2d_array_msaa_depth&lt;T&gt;::channel_data_type ()</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><code>channel_data_type()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>int image2d_msaa&lt;T&gt;::channel_order()</code></td>
<td>Return the image channel order.</td>
</tr>
<tr>
<td><code>int image2d_array_msaa&lt;T&gt;::channel_order()</code></td>
<td></td>
</tr>
<tr>
<td><code>int image2d_msaa_depth&lt;T&gt;::channel_order()</code></td>
<td></td>
</tr>
<tr>
<td><code>int image2d_array_msaa_depth&lt;T&gt;::channel_order()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>int image2d_array_msaa&lt;T&gt;::array_size()</code></td>
<td>Return the number of images in the 2D image array.</td>
</tr>
<tr>
<td><code>int image2d_array_msaa_depth&lt;T&gt;::array_size()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>int image2d_msaa&lt;T&gt;::num_samples()</code></td>
<td>Return the number of samples in the 2D MSAA image</td>
</tr>
<tr>
<td><code>int image2d_array_msaa&lt;T&gt;::num_samples()</code></td>
<td></td>
</tr>
<tr>
<td><code>int image2d_msaa_depth&lt;T&gt;::num_samples()</code></td>
<td></td>
</tr>
<tr>
<td><code>int image2d_array_msaa_depth&lt;T&gt;::num_samples()</code></td>
<td></td>
</tr>
</tbody>
</table>
9.5  Subgroup Independent Forward Progress

This extension adds support for implementation-controlled subgroups. Subgroups behave similarly to workgroups with their own sets of builtins and synchronization primitives. Subgroups within a workgroup are independent, make forward progress with respect to each other and may map to optimized hardware structures where that makes sense.

If this extension is supported by an implementation, the string `cl_khr_sub_group_forward_progress` will be present in the `CL_PLATFORM_EXTENSIONS` string described in *table 4.1*.

This is a behavioral change and exposes no other features. By querying the extension the user is made aware of a guarantee that the system of offering.

9.5.1.1 Additions to section 3.2 – Execution Model

Sub-groups execute concurrently within a given work-group and make independent forward progress with respect to each other even in the absence of work-group barrier operations. Sub-groups are able to internally synchronize using barrier operations without synchronizing with each other.
9.6 Named barriers for subgroups

Barrier operations that cover subsets of the OpenCL work-group have been requested both from inside and from outside the working group. The ability to perform barrier synchronization on flexible portions of an execution domain is present in a wide range of languages. Unfortunately, OpenCL 1.x had no concept of a subset of a work-group that could be portably synchronized independently of other parts of the work-group. Forward progress rules for work-items do not support this, for example. The sub-group provides an opportunity to change this. In the presence of the forward progress feature exposed as an extension query, sub-groups make forward progress independently of each other, in effect each mapping to a hardware thread or some multiple of one, such that synchronizing sets of sub-groups becomes a practical strategy. It is possible under OpenCL 2.0 rules to implement spin locks and other primitives such that a set of sub-groups would spin wait on a flag variable. However, OpenCL C does not expose all possible underlying operations that might be used to make spin-based synchronization more efficient on many architectures, and certainly can not guarantee that non-spin-based synchronization (such as hardware barriers) be used to implement partial synchronization. As a result the user-supplied methods of implementing such synchronization are likely to be inefficient. By providing a partial synchronization abstraction in the OpenCL C language vendors may optimize the implementation to make its use practical on their hardware. In all cases due to the forward progress guarantees of sub-groups a naive library implementation that spins will be possible and thus this does not add any hardware requirements, only making opportunities for hardware optimization. Encoding of inter-sub-group synchronization within the builtin library would allow an implementation of synchronization that does not depend on explicit sub-group forward progress, and that can utilize more efficient synchronization infrastructure in the hardware.

This extension requires cl_khr_sub_group_forward_progress extension to be available and enabled.

If this extension is supported by an implementation, the string cl_khr_sub_group_named_barrier will be present in the CL_PLATFORM_EXTENSIONS string described in table 4.1.

9.6.1 Additions to the OpenCL C++ Language

9.6.1.1 Additions to section 1.1.3 Other Built-in Data Types

work_group_named_barrier

An opaque synchronization object used for providing flexible barrier behavior between sub-groups within a work-group. An implementation shall support at least 8 named barriers per work-group.

Barriers are not copy constructable, nor are they move constructable.
9.6.1.2 Add a new section 2.14 Named Barrier Functions

A named barrier is specified through construction with the number of participant subgroups:

```cpp
explicit
work_group_named_barrier::work_group_named_barrier(uint sub_group_count);
```

Initialize a new named barrier object to synchronize sub group count sub-groups in the current work-group. Construction of a named-barrier object is a work-group operation and hence must be called uniformly across the work-group. sub_group_count must be uniform across the work-group.

Named barrier objects can be reconstructed and assigned to underlying entities by the compiler, or reused. Reused barriers will always be the same size and act in phases such that when each participating sub-group has waited the wait count is set to 0 and the entire process can start again. The internal wait count will cycle through the range from 0 to sub_group_count in each phase of use of the barrier.

Named barrier objects can only be constructed within kernels, not within arbitrary functions.

```cpp
void work_group_named_barrier::wait();
```

All work-items in a sub-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier. This function must be encountered by all work-items in a sub-group executing the kernel.

These rules apply to ND-ranges implemented with uniform and non-uniform workgroups. If `wait` is called inside a conditional statement, then all work-items within the subgroup must enter the conditional if any work-item in the sub-group enters the conditional statement and executes the call to `wait`. If `wait` is called inside a loop, all work-items within the sub-group must execute the wait operation for each iteration of the loop before any are allowed to continue execution beyond the call to `wait`. The `wait` function causes the entire sub-group to wait until sub group count subgroups have waited on the named barrier, where sub group count is the initialization value passed to the call to the constructor of the named barrier. Once the wait count equals sub group count, any sub-groups waiting at the named barrier will be released and the barrier’s wait count reset to 0. In addition, wait synchronizes according to the memory model as a read-modify-write atomic operation on the underlying named_barrier object with scope memory scope work group and memory order memory order acquire release. The act of being released acts as a read operation of the wait count at 0, with memory scope work group and memory order acquire.

9.7 Priority Hints

This extension adds priority hints for OpenCL, but does not specify the scheduling behaviour or minimum guarantees. It is expected that the the user guides associated with each implementation which supports this extension describe the scheduling behaviour guaranteed.

If this extension is supported by an implementation, the string `cl_khr_priority_hints` will be present in the CL_PLATFORM_EXTENSIONS string described in table 4.1.
9.7.1 Host-side API modifications

The function `clCreateCommandQueueWithProperties` (Section 5.1) is extended to support a priority value as part of the `properties` argument.

The priority property applies to OpenCL command queues that belong to the same OpenCL context.

The `properties` field accepts the `CL_QUEUE_PRIORITY_KHR` property, with a value of type `cl_queue_priority_khr` which can be one of:

- `CL_QUEUE_PRIORITY_HIGH_KHR`
- `CL_QUEUE_PRIORITY_MED_KHR`
- `CL_QUEUE_PRIORITY_LOW_KHR`

If `CL_QUEUE_PRIORITY_KHR` is not specified then the default priority is `CL_QUEUE_PRIORITY_MED_KHR`.

To the error section for `clCreateCommandQueueWithProperties`, the following is added:

* `CL_INVALID_QUEUE_PROPERTIES` if the `CL_QUEUE_PRIORITY_KHR` property is specified and the queue is a `CL_QUEUE_ON_DEVICE`.

9.8 Throttle Hints

This extension adds throttle hints for OpenCL, but does not specify the throttling behaviour or minimum guarantees. It is expected that the user guide associated with each implementation which supports this extension describe the throttling behaviour guaranteed.

If this extension is supported by an implementation, the string `cl_khr_throttle_hints` will be present in the `CL_PLATFORM_EXTENSIONS` string described in table 4.1.

Note that the throttle hint is orthogonal to functionality defined in `cl_khr_priority_hints` extension. For example, a task may have high priority (`CL_QUEUE_PRIORITY_HIGH_KHR`) but should at the same time be executed at an optimized throttle setting (`CL_QUEUE_THROTTLE_LOW_KHR`).

9.8.1 Host-side API modifications

The function `clCreateCommandQueueWithProperties` (Section 5.1) is extended to support a new `CL_QUEUE_THROTTLE_KHR` value as part of the `properties` argument.

The `properties` field accepts the following values:

- `CL_QUEUE_THROTTLE_HIGH_KHR` (full throttle, i.e., OK to consume more energy)
- `CL_QUEUE_THROTTLE_MED_KHR` (normal throttle)
- `CL_QUEUE_THROTTLE_LOW_KHR` (optimized/lowest energy consumption)
If \texttt{CL_QUEUE\_THROTTL\_KHR} is not specified then the default priority is \texttt{CL\_QUEUE\_THROTTL\_MED\_KHR}.

To the error section for \texttt{clCreateCommandQueueWithProperties}, the following is added:

* \texttt{CL\_INVALID\_QUEUE\_PROPERTIES} if the \texttt{CL\_QUEUE\_PRIORITY\_KHR} property is specified
and the queue is a \texttt{CL\_QUEUE\_ON\_DEVICE}.
Index - APIs

clGetExtensionFunctionAddressForPlatform, 6