The **OpenCL** Specification

Version: 1.1

Document Revision: 33

Khronos OpenCL Working Group

Editor: Aaftab Munshi

Copyright (c) 2008-2010 The Khronos Group Inc. All Rights Reserved.

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

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

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

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

Acknowledgements

The OpenCL specification is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Benedict Gaster, AMD
Bill Licea Kane, AMD
Ed Buckingham, AMD
Jan Civlin, AMD
Laurent Morichetti, AMD
Mark Fowler, AMD
Michael Houston, AMD
Michael Mantor, AMD
Norm Rubin, AMD
Victor Odintsov, AMD
Aaftab Munshi, Apple
Abe Stephens, Apple
Benjamin Lipchak, Apple
Bob Beretta, Apple
Daniel N. Gessel, Apple
David Black-Schaffer, Apple
Derek Gerstmann, Apple
Geoff Stahl, Apple
Ian Ollmann, Apple
Inam Rahman, Apple
Jeff Kidder, Apple
Jeremy Sandmel, Apple
John Stauffer, Apple
Kathleen Danielson, Apple
Michael Larson, Apple
MonPing Wang, Apple
Nate Begeman, Apple
Nick Burns, Apple
Nicolas Moss, Apple
Ralph Brunner, Apple
Stephen Canon, Apple
Tanya Lattner, Apple
Travis Brown, Apple
Anton Lokhmotov, ARM
Dave Shreiner, ARM
Hedley Francis, ARM
Robert Elliott, ARM
Scott Moyers, ARM

Tom Olson, ARM
Roger Nixon, Broadcom
Rob Barris, Blizzard
Alastair Donaldson, Codeplay
Andrew Richards, Codeplay
Andrew Brownsword, Electronic Arts
Eric Schenk, Electronic Arts
Erik Noreke, Ericsson
Jacob Strom, Ericsson
Teddie Stenvi, Ericsson
Zhengrong Yao, Ericsson
Brian Murray, Freescale
Avi Shapira, Graphic Remedy
Uri Shomroni, Graphic Remedy
Yaki Tebeka, Graphic Remedy
Barry Minor, IBM
Brian Watt, IBM
Dan Brokenshire, IBM
Greg Bellows, IBM
Joaquin Madruga, IBM
Mark Nutter, IBM
Joe Molleson, Imagination Technologies
Aaron Lefohn, Intel
Andrew Lauritzen, Intel
Craig Kolb, Intel
Geoff Berry, Intel
John Kessenich, Intel
Josh Fryman, Intel
Hong Jiang, Intel
Larry Seiler, Intel
Matt Pharr, Intel
Michael McCool, Intel
Murali Sundaresan, Intel
Ofer Rosenberg, Intel
Paul Lalonde, Intel
Stefanus Du Toit, Intel
Stephen Junkins, Intel
Tim Foley, Intel
Timothy Mattson, Intel
Bill Bush, Kestrel Institute
Lindsay Errington, Kestrel Institute
Jon Leech, Khronos
Benjamin Bergen, Los Alamos National Laboratory
Marcus Daniels, Los Alamos National Laboratory

Cormac Brick, Movidia
David Donohoe, Movidia
Jyrki Leskelä, Nokia
Kari Pulli, Nokia
Amit Rao, NVIDIA
Bastiaan Aarts, NVIDIA
Chris Cameron, NVIDIA
Christopher Lamb, NVIDIA
Ian Buck, NVIDIA
Jason Sanders, NVIDIA
Jaydeep Marathe, NVIDIA
Jian-Zhong Wang, NVIDIA
Kedar Patil, NVIDIA
Manjunath Kudlur, NVIDIA
Mark Harris, NVIDIA
Michael Gold, NVIDIA
Neil Trevett, NVIDIA
Vinod Grover, NVIDIA
Xiangyun Kong, NVIDIA
Yogesh Kini, NVIDIA
Yuan Lin, NVIDIA
Dairsie Latimer, Petapath
Neil Hickey, Petapath
Alex Bourd, QUALCOMM
Andrzej Mamona, QUALCOMM
Chihong Zhang, QUALCOMM
David Garcia, QUALCOMM
David Ligon, QUALCOMM
Robert Simpson, QUALCOMM
Jonathan Grant, Renesas
Yanjun Zhang, S3 Graphics
Robert Schulman, Seaweed Systems
Erwin Coumans, Sony
John Bates, Sony
Thierry Lepley, STMicroelectronics
Ajit Kamat, Symbian
Alan Ward, Texas Instruments
Madhukar Budagavi, Texas Instruments

1. Introduction

Modern processor architectures have embraced parallelism as an important pathway to increased performance. Facing technical challenges with higher clock speeds in a fixed power envelope, Central Processing Units (CPUs) now improve performance by adding multiple cores. Graphics Processing Units (GPUs) have also evolved from fixed function rendering devices into programmable parallel processors. As today's computer systems often include highly parallel CPUs, GPUs and other types of processors, it is important to enable software developers to take full advantage of these heterogeneous processing platforms.

Creating applications for heterogeneous parallel processing platforms is challenging as traditional programming approaches for multi-core CPUs and GPUs are very different. CPU-based parallel programming models are typically based on standards but usually assume a shared address space and do not encompass vector operations. General purpose GPU programming models address complex memory hierarchies and vector operations but are traditionally platform-, vendor- or hardware-specific. These limitations make it difficult for a developer to access the compute power of heterogeneous CPUs, GPUs and other types of processors from a single, multi-platform source code base. More than ever, there is a need to enable software developers to effectively take full advantage of heterogeneous processing platforms – from high performance compute servers, through desktop computer systems to handheld devices - that include a diverse mix of parallel CPUs, GPUs and other processors such as DSPs and the Cell/B.E. processor.

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.

OpenCL supports a wide range of applications, ranging from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient, close-to-the-metal programming interface, OpenCL will form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications. OpenCL is particularly suited to play an increasingly significant role in emerging interactive graphics applications that combine general parallel compute algorithms with graphics rendering pipelines.

OpenCL consists of an API for coordinating parallel computation across heterogeneous processors; and a cross-platform programming language with a well-specified computation environment. The OpenCL standard:

- Supports both data- and task-based parallel programming models
- Utilizes a subset of ISO C99 with extensions for parallelism
- Defines consistent numerical requirements based on IEEE 754
- Defines a configuration profile for handheld and embedded devices
- Efficiently interoperates with OpenGL, OpenGL ES and other graphics APIs

This document begins with an overview of basic concepts and the architecture of OpenCL, followed by a detailed description of its execution model, memory model and synchronization support. It then discusses the OpenCL platform and runtime API and is followed by a detailed description of the OpenCL C programming language. Some examples are given that describe sample compute use-cases and how they would be written in OpenCL. The specification is divided into a core specification that any OpenCL compliant implementation must support; a handheld/embedded profile which relaxes the OpenCL compliance requirements for handheld and embedded devices; and a set of optional extensions that are likely to move into the core specification in later revisions of the OpenCL specification.

2. Glossary

Application: The combination of the program running on the *host* and *OpenCL device*s.

Blocking and Non-Blocking Enqueue API calls: A *non-blocking enqueue API call* places a *command* on a *command-queue* and returns immediately to the host. The *blocking-mode enqueue API calls* do not return to the host until the command has completed.

Barrier: There are two types of *barriers* – a command-queue barrier and a work-group barrier.

- The OpenCL API provides a function to enqueue a *command-queue barrier* command. This *barrier* command ensures that all previously enqueued commands to a command-queue have finished execution before any following *commands* enqueued in the *command-queue* can begin execution.

- The OpenCL C programming language provides a built-in *work-group barrier* function. This *barrier* built-in function can be used by a *kernel* executing on a *device* to perform synchronization between *work-items* in a *work-group* executing the *kernel*. All the *work-items* of a *work-group* must execute the *barrier* construct before any are allowed to continue execution beyond the *barrier*.

Buffer Object: A memory object that stores a linear collection of bytes. Buffer objects are accessible using a pointer in a *kernel* executing on a *device*. Buffer objects can be manipulated by the host using OpenCL API calls. A *buffer object* encapsulates the following information:

- Size in bytes.
- Properties that describe usage information and which region to allocate from.
- Buffer data.

Command: The OpenCL operations that are submitted to a *command-queue* for execution. For example, OpenCL commands issue kernels for execution on a compute device, manipulate memory objects, etc.

Command-queue: An object that holds *commands* that will be executed on a specific *device*. The *command-queue* is created on a specific *device* in a *context*. *Commands* to a *command-queue* are queued in-order but may be executed in-order or out-of-order. *Refer to In-order Execution* and *Out-of-order Execution*.

Command-queue Barrier. See *Barrier*.

Compute Device Memory: This refers to one or more memories attached to the compute device.

Compute Unit: An OpenCL *device* has one or more *compute units*. A *work-group* executes on a single *compute unit*. A *compute unit* is composed of one or more *processing elements*. A *compute unit* may also include dedicated texture filtering units that can be accessed by its processing elements.

Concurrency: A property of a system in which a set of tasks in a system can remain active and make progress at the same time. To utilize concurrent execution when running a program, a programmer must identify the concurrency in their problem, expose it within the source code, and then exploit it using a notation that supports concurrency.

Constant Memory: A region of *global memory* that remains constant during the execution of a *kernel*. The *host* allocates and initializes memory objects placed into *constant memory*.

Context: The environment within which the *kernels* execute and the domain in which synchronization and memory management is defined. The *context* includes a set of *devices*, the memory accessible to those *devices*, the corresponding memory properties and one or more *command-queues* used to schedule execution of a *kernel(s)* or operations on *memory objects*.

Data Parallel Programming Model: Traditionally, this term refers to a programming model where concurrency is expressed as instructions from a single program applied to multiple elements within a set of data structures. The term has been generalized in OpenCL to refer to a model wherein a set of instructions from a single program are applied concurrently to each point within an abstract domain of indices.

Device: A *device* is a collection of *compute units*. A *command-queue* is used to queue *commands* to a *device*. Examples of *commands* include executing *kernels*, or reading and writing *memory objects*. OpenCL devices typically correspond to a GPU, a multi-core CPU, and other processors such as DSPs and the Cell/B.E. processor.

Event Object: An *event object* encapsulates the status of an operation such as a *command*. It can be used to synchronize operations in a context.

Event Wait List: An *event wait list* is a list of *event objects* that can be used to control when a particular *command* begins execution.

Framework: A software system that contains the set of components to support software development and execution. A *framework* typically includes libraries, APIs, runtime systems, compilers, etc.

Global ID: A *global ID* is used to uniquely identify a *work-item* and is derived from the number of *global work-items* specified when executing a *kernel*. The *global ID* is a N-dimensional value that starts at (0, 0, … 0). See also *Local ID*.

Global Memory: A memory region accessible to all *work-items* executing in a *context*. It is accessible to the *host* using *commands* such as read, write and map.

GL share group: A *GL share group* object manages shared OpenGL or OpenGL ES resources such as textures, buffers, framebuffers, and renderbuffers and is associated with one or more GL context objects. The *GL share group* is typically an opaque object and not directly accessible.

Handle: An opaque type that references an *object* allocated by OpenCL. Any operation on an *object* occurs by reference to that object's handle**.**

Host: The *host* interacts with the *context* using the OpenCL API.

Host pointer: A pointer to memory that is in the virtual address space on the *host*.

Illegal: Behavior of a system that is explicitly not allowed and will be reported as an error when encountered by OpenCL.

Image Object: A *memory object* that stores a two- or three- dimensional structured array. Image data can only be accessed with read and write functions. The read functions use a *sampler*.

The *image object* encapsulates the following information:

- Dimensions of the image.
- Description of each element in the image.
- Properties that describe usage information and which region to allocate from.
- Image data.

The elements of an image are selected from a list of predefined image formats.

Implementation Defined: Behavior that is explicitly allowed to vary between conforming implementations of OpenCL. An OpenCL implementor is required to document the implementation-defined behavior.

In-order Execution: A model of execution in OpenCL where the *commands* in a *command-queue* are executed in order of submission with each *command* running to completion before the next one begins. See *Out-of-order Execution*.

Kernel: A *kernel* is a function declared in a *program* and executed on an OpenCL *device*. A *kernel* is identified by the `__kernel` qualifier applied to any function defined in a *program*.

Kernel Object: A *kernel object* encapsulates a specific `__kernel` function declared in a *program* and the argument values to be used when executing this `__kernel` function.

Local ID: A *local ID* specifies a unique *work-item ID* within a given *work-group* that is executing a *kernel*. The *local ID* is a N-dimensional value that starts at (0, 0, … 0). See also *Global ID*.

Local Memory: A memory region associated with a *work-group* and accessible only by *work-items* in that *work-group*.

Marker: A *command* queued in a *command-queue* that can be used to tag all *commands* queued before the *marker* in the *command-queue*. The *marker* command returns an *event* which can be used by the *application* to queue a wait on the marker event i.e. wait for all commands queued before the *marker* command to complete.

Memory Objects: A *memory object* is a handle to a reference counted region of *global memory*. Also see *Buffer Object* and *Image Object*.

Memory Regions (or Pools): A distinct address space in OpenCL. *Memory regions* may overlap in physical memory though OpenCL will treat them as logically distinct. The *memory regions* are denoted as *private*, *local*, *constant,* and *global*.

Object: Objects are abstract representation of the resources that can be manipulated by the OpenCL API. Examples include *program objects*, *kernel objects*, and *memory objects*.

Out-of-Order Execution: A model of execution in which *commands* placed in the *work queue* may begin and complete execution in any order consistent with constraints imposed by *event wait lists* and *command-queue barrier*. See *In-order Execution*.

Platform: The *host* plus a collection of *devices* managed by the OpenCL *framework* that allow an application to share *resources* and execute *kernels* on *devices* in the *platform*.

Private Memory: A region of memory private to a *work-item*. Variables defined in one *work-item's private memory* are not visible to another *work-item*.

Processing Element: A virtual scalar processor. A work-item may execute on one or more processing elements.

Program: An OpenCL *program* consists of a set of *kernels*. *Programs* may also contain auxiliary functions called by the `__kernel` functions and constant data.

Program Object: A *program object* encapsulates the following information:

- A reference to an associated *context*.
- A *program* source or binary.
- The latest successfully built program executable, the list of *devices* for which the program executable is built, the build options used and a build log.
- The number of *kernel objects* currently attached.

Reference Count: The life span of an OpenCL object is determined by its *reference count*—an internal count of the number of references to the object. When you create an object in OpenCL, its *reference count* is set to one. Subsequent calls to the appropriate *retain* API (such as `clRetainContext`, `clRetainCommandQueue`) increment the *reference count*. Calls to the appropriate *release* API (such as `clReleaseContext`, `clReleaseCommandQueue`) decrement the *reference count*. After the *reference count* reaches zero, the object's resources are deallocated by OpenCL.

Relaxed Consistency: A memory consistency model in which the contents of memory visible to different *work-items* or *commands* may be different except at a *barrier* or other explicit synchronization points.

Resource: A class of *objects* defined by OpenCL. An instance of a *resource* is an *object*. The most common *resources* are the *context*, *command-queue*, *program objects*, *kernel objects*, and *memory objects*. Computational resources are hardware elements that participate in the action of advancing a program counter. Examples include the *host*, *devices*, *compute units* and *processing elements*.

Retain, Release: The action of incrementing (retain) and decrementing (release) the reference count using an OpenCL *object*. This is a book keeping functionality to make sure the system doesn't remove an *object* before all instances that use this *object* have finished. Refer to *Reference Count*.

Sampler: An *object* that describes how to sample an image when the image is read in the *kernel*. The image read functions take a *sampler* as an argument. The *sampler* specifies the image addressing-mode i.e. how out-of-range image coordinates are handled, the filtering mode, and whether the input image coordinate is a normalized or unnormalized value.

SIMD: Single Instruction Multiple Data. A programming model where a *kernel* is executed concurrently on multiple *processing elements* each with its own data and a shared program counter. All *processing elements* execute a strictly identical set of instructions.

SPMD: Single Program Multiple Data. A programming model where a *kernel* is executed concurrently on multiple *processing elements* each with its own data and its own program counter. Hence, while all computational resources run the same *kernel* they maintain their own instruction counter and due to branches in a *kernel*, the actual sequence of instructions can be quite different across the set of *processing elements*.

Task Parallel Programming Model: A programming model in which computations are expressed in terms of multiple concurrent tasks where a task is a *kernel* executing in a single *work-group* of size one. The concurrent tasks can be running different *kernels*.

Thread-safe: An OpenCL API call is considered to be *thread-safe* if the internal state as managed by OpenCL remains consistent when called simultaneously by multiple *host* threads. OpenCL API calls that are *thread-safe* allow an application to call these functions in multiple *host* threads without having to implement mutual exclusion across these *host* threads i.e. they are also re-entrant-safe.

Undefined: The behavior of an OpenCL API call, built-in function used inside a *kernel* or execution of a *kernel* that is explicitly not defined by OpenCL. A conforming implementation is not required to specify what occurs when an undefined construct is encountered in OpenCL.

Work-group: A collection of related *work-items* that execute on a single *compute unit*. The *work-items* in the group execute the same *kernel* and share *local memory* and *work-group barriers*.

Work-group Barrier. See *Barrier*.

Work-item: One of a collection of parallel executions of a *kernel* invoked on a *device* by a *command*. A *work-item* is executed by one or more *processing elements* as part of a *work-group* executing on a *compute unit*. A *work-item* is distinguished from other executions within the collection by its *global ID* and *local ID*.

2.1 OpenCL Class Diagram

Figure 2.1 describes the OpenCL specification as a class diagram using the Unified Modeling Language[1] (UML) notation. The diagram shows both nodes and edges which are classes and their relationships. As a simplification it shows only classes, and no attributes or operations. Abstract classes are annotated with "{abstract}". As for relationships it shows aggregations (annotated with a solid diamond), associations (no annotation), and inheritance (annotated with an open arrowhead). The cardinality of a relationship is shown on each end of it. A cardinality of "*****" represents "many", a cardinality of "**1**" represents "one and only one", a cardinality of "**0..1**" represents "optionally one", and a cardinality of "**1..***" represents "one or more". The navigability of a relationship is shown using a regular arrowhead.

---

[1] Unified Modeling Language (http://www.uml.org/) is a trademark of Object Management Group (OMG).

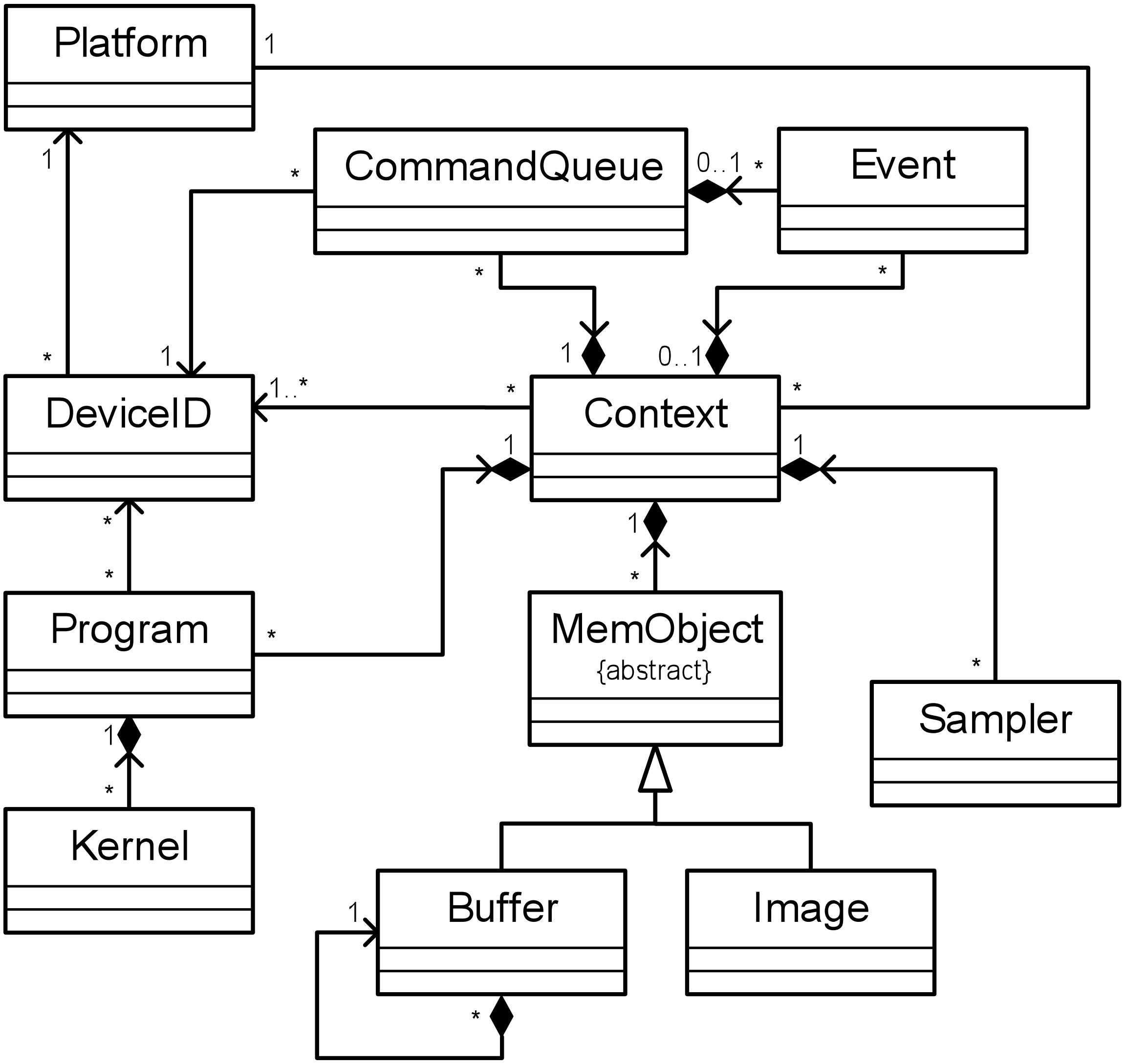


Figure 2.1 - *OpenCL UML Class Diagram*

3. The OpenCL Architecture

OpenCL is an open industry standard for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform. It is more than a language. OpenCL is a framework for parallel programming and includes a language, API, libraries and a runtime system to support software development. Using OpenCL, for example, a programmer can write general purpose programs that execute on GPUs without the need to map their algorithms onto a 3D graphics API such as OpenGL or DirectX.

The target of OpenCL is expert programmers wanting to write portable yet efficient code. This includes library writers, middleware vendors, and performance oriented application programmers. Therefore OpenCL provides a low-level hardware abstraction plus a framework to support programming and many details of the underlying hardware are exposed.

To describe the core ideas behind OpenCL, we will use a hierarchy of models:

- Platform Model
- Memory Model
- Execution Model
- Programming Model

3.1 Platform Model

The Platform model for OpenCL is defined in *figure 3.1*. The model consists of a **host** connected to one or more **OpenCL devices**. An OpenCL device is divided into one or more **compute units** (CUs) which are further divided into one or more **processing elements** (PEs). Computations on a device occur within the processing elements.

An OpenCL application runs on a host according to the models native to the host platform. The OpenCL application submits **commands** from the host to execute computations on the processing elements within a device. The processing elements within a compute unit execute a single stream of instructions as SIMD units (execute in lockstep with a single stream of instructions) or as SPMD units (each PE maintains its own program counter).

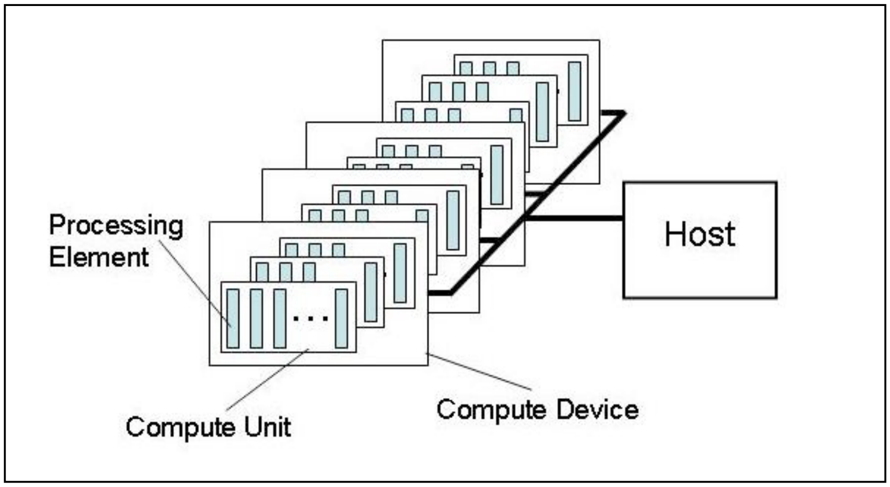


Figure 3.1: *Platform model … one host plus one or more compute devices each with one or more compute units each with one or more processing elements.*

3.1.1 Platform Mixed Version Support

OpenCL is designed to support devices with different capabilities under a single platform. This includes devices which conform to different versions of the OpenCL specification. There are three important version identifiers to consider for an OpenCL system: the platform version, the version of a device, and the version(s) of the OpenCL C language supported on a device.

The platform version indicates the version of the OpenCL runtime supported. This includes all of the APIs that the host can use to interact with the OpenCL runtime, such as contexts, memory objects, devices, and command queues.

The device version is an indication of the devices capabilities, separate from the runtime and compiler, as represented by the device info returned by **clGetDeviceInfo**. Examples of attributes associated with the device version are resource limits and extended functionality. The version returned corresponds to the highest version of the OpenCL spec for which the device is conformant, but is not higher than the platform version.

The language version for a device represents the OpenCL programming language features a developer can assume are supported on a given device. The version reported is the highest version of the language supported.

OpenCL C is designed to be backwards compatible, so a device is not required to support more than a single language version to be considered conformant. If multiple language versions are supported, the compiler defaults to using the highest language version supported for the device. The language version is not higher than the platform version, but may exceed the device version (see *section 5.6.3.5*).

3.2 Execution Model

Execution of an OpenCL program occurs in two parts: **kernels** that execute on one or more **OpenCL devices** and a **host program** that executes on the host. The host program defines the context for the kernels and manages their execution.

The core of the OpenCL execution model is defined by how the kernels execute. When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a **work-item** and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item.

Work-items are organized into **work-groups**. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

The index space supported in OpenCL is called an NDRange. An NDRange is an N-dimensional index space, where N is one, two or three. An NDRange is defined by an integer array of length N specifying the extent of the index space in each dimension starting at an offset index F (zero by default). Each work-item's global ID and local ID are N-dimensional tuples. The global ID components are values in the range from F, to F plus the number of elements in that dimension minus one.

Work-groups are assigned IDs using a similar approach to that used for work-item global IDs. An array of length N defines the number of work-groups in each dimension. Work-items are assigned to a work-group and given a local ID with components in the range from zero to the size of the work-group in that dimension minus one. Hence, the combination of a work-group ID and the local-ID within a work-group uniquely defines a work-item. Each work-item is identifiable in two ways; in terms of a global index, and in terms of a work-group index plus a local index within a work group.

For example, consider the 2-dimensional index space in *figure 3.2*. We input the index space for the work-items (G_x, G_y), the size of each work-group (S_x, S_y) and the global ID offset (F_x, F_y). The global indices define an G_x by G_y index space where the total number of work-items is the product of G_x and G_y. The local indices define a S_x by S_y index space where the number of work-items in a single work-group is the product of S_x and S_y. Given the size of each work-group and the total number of work-items we can compute the number of work-groups. A 2-dimensional index space is used to uniquely identify a work-group. Each work-item is identified by its global ID (g_x, g_y) or by the combination of the work-group ID (w_x, w_y), the size of each work-group (S_x, S_y) and the local ID (s_x, s_y) inside the workgroup such that

$$(g_x , g_y) = (w_x * S_x + s_x + F_x, w_y * S_y + s_y + F_y)$$

The number of work-groups can be computed as:

$$(W_x, W_y) = (G_x / S_x, G_y / S_y)$$

Given a global ID and the work-group size, the work-group ID for a work-item is computed as:

$$(w_x, w_y) = ((g_x - s_x - F_x) / S_x, (g_y - s_y - F_y) / S_y)$$

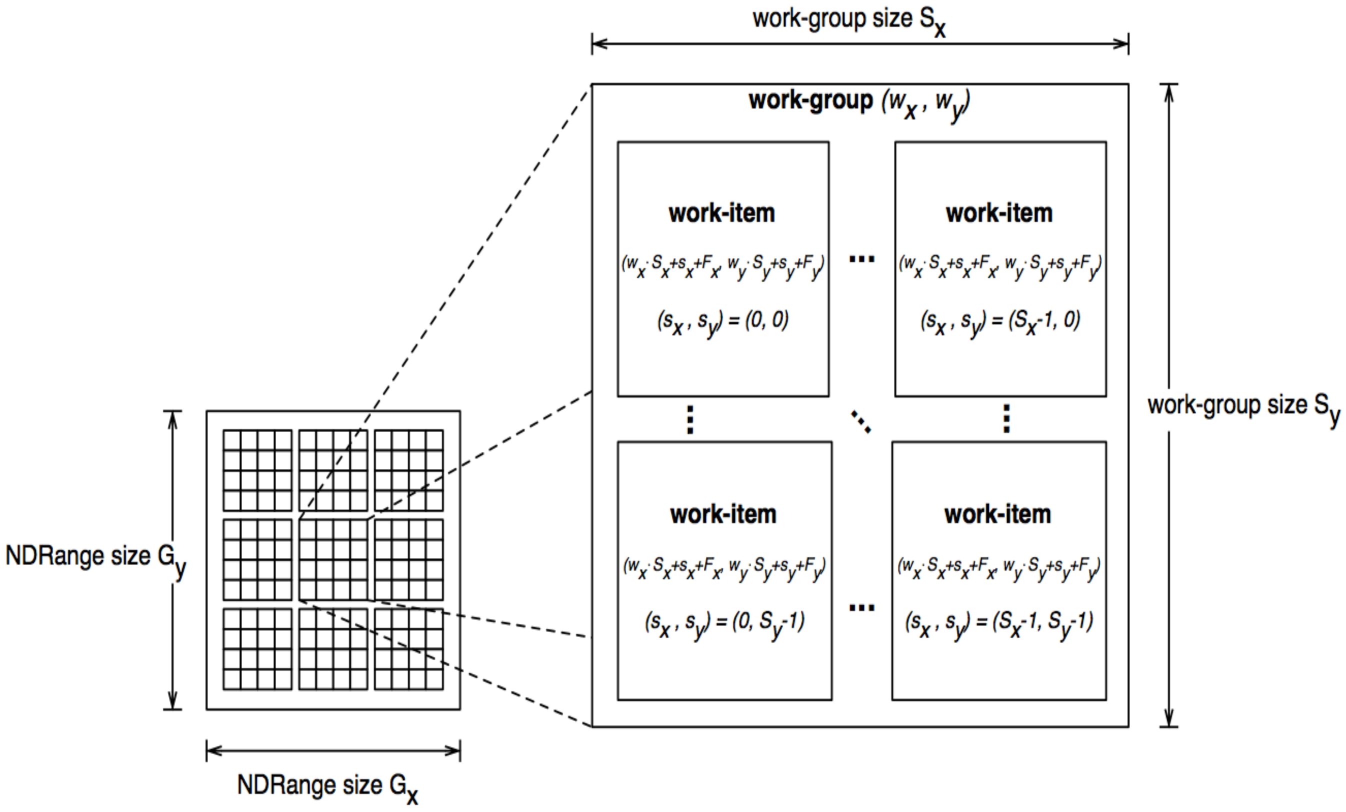


Figure 3.2 *An example of an NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs.*

A wide range of programming models can be mapped onto this execution model. We explicitly support two of these models within OpenCL; the **data parallel programming model** and the **task parallel programming model**.

3.2.1 Execution Model: Context and Command Queues

The host defines a context for the execution of the kernels. The context includes the following resources:

1. **Devices**: The collection of OpenCL devices to be used by the host.
2. **Kernels:** The OpenCL functions that run on OpenCL devices.
3. **Program Objects:** The program source and executable that implement the kernels.
4. **Memory Objects:** A set of memory objects visible to the host and the OpenCL devices. Memory objects contain values that can be operated on by instances of a kernel.

The context is created and manipulated by the host using functions from the OpenCL API. The host creates a data structure called a **command-queue** to coordinate execution of the kernels on the devices. The host places commands into the command-queue which are then scheduled onto the devices within the context. These include:

- **Kernel execution commands**: Execute a kernel on the processing elements of a device.
- **Memory commands**: Transfer data to, from, or between memory objects, or map and unmap memory objects from the host address space.
- **Synchronization commands**: Constrain the order of execution of commands.

The command-queue schedules commands for execution on a device. These execute asynchronously between the host and the device. Commands execute relative to each other in one of two modes:

- **In-order Execution**: Commands are launched in the order they appear in the command-queue and complete in order. In other words, a prior command on the queue completes before the following command begins. This serializes the execution order of commands in a queue.
- **Out-of-order Execution**: Commands are issued in order, but do not wait to complete before following commands execute. Any order constraints are enforced by the programmer through explicit synchronization commands.

Kernel execution and memory commands submitted to a queue generate event objects. These are used to control execution between commands and to coordinate execution between the host and devices.

It is possible to associate multiple queues with a single context. These queues run concurrently and independently with no explicit mechanisms within OpenCL to synchronize between them.

3.2.2 Execution Model: Categories of Kernels

The OpenCL execution model supports two categories of kernels:

- **OpenCL kernels** are written with the OpenCL C programming language and compiled with the OpenCL compiler. All OpenCL implementations support OpenCL kernels. Implementations may provide other mechanisms for creating OpenCL kernels.

- **Native kernels** are accessed through a host function pointer. Native kernels are queued for execution along with OpenCL kernels on a device and share memory objects with OpenCL kernels. For example, these native kernels could be functions defined in application code or exported from a library. Note that the ability to execute native kernels is an optional functionality within OpenCL and the semantics of native kernels are implementation-defined. The OpenCL API includes functions to query capabilities of a device(s) and determine if this capability is supported.

3.3 Memory Model

Work-item(s) executing a kernel have access to four distinct memory regions:

- **Global Memory**. This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.

- **Constant Memory**: A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

- **Local Memory**: A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.

- **Private Memory**: A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

Table 3.1 describes whether the kernel or the host can allocate from a memory region, the type of allocation (static i.e. compile time vs dynamic i.e. runtime) and the type of access allowed i.e. whether the kernel or the host can read and/or write to a memory region.

	Global	**Constant**	**Local**	**Private**
Host	Dynamic allocation Read / Write access	Dynamic allocation Read / Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read / Write access	Static allocation Read-only access	Static allocation Read / Write access	Static allocation Read / Write access

Table 3.1 *Memory Region - Allocation and Memory Access Capabilities*

The memory regions and how they relate to the platform model are described in *figure 3.3*.

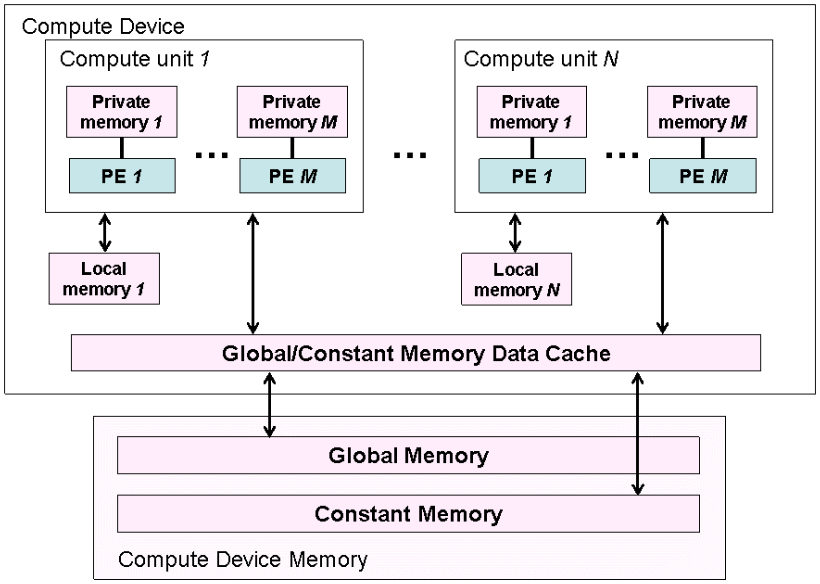


Figure 3.3: *Conceptual OpenCL device architecture with processing elements (PE), compute units and devices. The host is not shown.*

The application running on the host uses the OpenCL API to create memory objects in global memory, and to enqueue memory commands (described in *section 3.2.1*) that operate on these memory objects.

The host and OpenCL device memory models are, for the most part, independent of each other. This is by a necessity given that the host is defined outside of OpenCL. They do, however, at times need to interact. This interaction occurs in one of two ways: by explicitly copying data or by mapping and unmapping regions of a memory object.

To copy data explicitly, the host enqueues commands to transfer data between the memory object and host memory. These memory transfer commands may be blocking or non-blocking. The OpenCL function call for a blocking memory transfer returns once the associated memory resources on the host can be safely reused. For a non-blocking memory transfer, the OpenCL function call returns as soon as the command is enqueued regardless of whether host memory is safe to use.

The mapping/unmapping method of interaction between the host and OpenCL memory objects allows the host to map a region from the memory object into its address space. The memory map command may be blocking or non-blocking. Once a region from the memory object has been mapped, the host can read or write to this region. The host unmaps the region when accesses (reads and/or writes) to this mapped region by the host are complete.

3.3.1 Memory Consistency

OpenCL uses a relaxed consistency memory model; i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.

Within a work-item memory has load / store consistency. Local memory is consistent across work-items in a single work-group at a work-group barrier. Global memory is consistent across work-items in a single work-group at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing a kernel.

Memory consistency for memory objects shared between enqueued commands is enforced at a synchronization point.

3.4 Programming Model

The OpenCL execution model supports **data parallel** and **task parallel** programming models, as well as supporting hybrids of these two models. The primary model driving the design of OpenCL is data parallel.

3.4.1 Data Parallel Programming Model

A data parallel programming model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items. In a strictly data parallel model, there is a one-to-one mapping between the work-item and the element in a memory object over which a kernel can be executed in parallel. OpenCL implements a relaxed version of the data parallel programming model where a strict one-to-one mapping is not a requirement.

OpenCL provides a hierarchical data parallel programming model. There are two ways to specify the hierarchical subdivision. In the explicit model a programmer defines the total number of work-items to execute in parallel and also how the work-items are divided among work-groups. In the implicit model, a programmer specifies only the total number of work-items to execute in parallel, and the division into work-groups is managed by the OpenCL implementation.

3.4.2 Task Parallel Programming Model

The OpenCL task parallel programming model defines a model in which a single instance of a kernel is executed independent of any index space. It is logically equivalent to executing a kernel on a compute unit with a work-group containing a single work-item. Under this model, users express parallelism by:

- using vector data types implemented by the device,
- enqueuing multiple tasks, and/or
- enqueing native kernels developed using a programming model orthogonal to OpenCL.

3.4.3 Synchronization

There are two domains of synchronization in OpenCL:

- Work-items in a single work-group
- Commands enqueued to command-queue(s) in a single context

Synchronization between work-items in a single work-group is done using a work-group barrier. All the work-items of a work-group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all. There is no mechanism for synchronization between work-groups.

The synchronization points between commands in command-queues are:

- Command-queue barrier. The command-queue barrier ensures that all previously queued commands have finished execution and any resulting updates to memory objects are visible to subsequently enqueued commands before they begin execution. This barrier can only be used to synchronize between commands in a single command-queue.

- Waiting on an event. All OpenCL API functions that enqueue commands return an event that identifies the command and memory objects it updates. A subsequent command waiting on that event is guaranteed that updates to those memory objects are visible before the command begins execution.

3.5 Memory Objects

Memory objects are categorized into two types: *buffer* objects, and *image* objects. A *buffer* object stores a one-dimensional collection of elements whereas an *image* object is used to store a two- or three- dimensional texture, frame-buffer or image.

Elements of a *buffer* object can be a scalar data type (such as an int, float), vector data type, or a user-defined structure. An *image* object is used to represent a buffer that can be used as a texture or a frame-buffer. The elements of an image object are selected from a list of predefined image formats. The minimum number of elements in a memory object is one.

The fundamental differences between a *buffer* and an *image* object are:

- Elements in a *buffer* are stored in sequential fashion and can be accessed using a pointer by a kernel executing on a device. Elements of an *image* are stored in a format that is opaque to the user and cannot be directly accessed using a pointer. Built-in functions are provided by the OpenCL C programming language to allow a kernel to read from or write to an image.

- For a *buffer* object, the data is stored in the same format as it is accessed by the kernel, but in the case of an *image* object the data format used to store the image elements may not be the same as the data format used inside the kernel. Image elements are always a 4-component vector (each component can be a float or signed/unsigned integer) in a kernel. The built-in function to read from an image converts image element from the format it is stored into a 4-component vector. Similarly, the built-in function to write to an image converts the image element from a 4-component vector to the appropriate image format specified such as 4 8-bit elements, for example.

Memory objects are described by a **cl_mem** object. Kernels take memory objects as input, and output to one or more memory objects.

3.6 The OpenCL Framework

The OpenCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

- **OpenCL Platform layer**: The platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts.

- **OpenCL Runtime**: The runtime allows the host program to manipulate contexts once they have been created.

- **OpenCL Compiler**: The OpenCL compiler creates program executables that contain OpenCL kernels. The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism.

4. The OpenCL Platform Layer

This section describes the OpenCL platform layer which implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

4.1 Querying Platform Info

The list of platforms available can be obtained using the following function.

cl_int **clGetPlatformIDs** (cl_uint *num_entries*,
cl_platform_id **platforms*,
cl_uint **num_platforms*)

num_entries is the number of cl_platform_id entries that can be added to *platforms.* If *platforms* is not NULL, the *num_entries* must be greater than zero.

platforms returns a list of OpenCL platforms found. The cl_platform_id values returned in *platforms* can be used to identify a specific OpenCL platform. If *platforms* argument is NULL, this argument is ignored. The number of OpenCL platforms returned is the minimum of the value specified by *num_entries* or the number of OpenCL platforms available.

num_platforms returns the number of OpenCL platforms available. If *num_platforms* is NULL, this argument is ignored.

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

- CL_INVALID_VALUE if *num_entries* is equal to zero and *platforms* is not NULL or if both *num_platforms* and *platforms* are NULL.

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

The function

cl_int **clGetPlatformInfo** (cl_platform_id *platform*,
cl_platform_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

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

platform refers to the platform ID returned by **clGetPlatformIDs** or can be NULL. If *platform* is NULL, the behavior is implementation-defined.

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

param_value is a pointer to memory location where appropriate values for a given *param_name* as specified in *table 4.1* will be returned. If *param_value* is NULL, it is ignored.

param_value_size specifies the size in bytes of memory pointed to by *param_value*. This size in bytes must be >= size of return type specified in *table 4.1*.

param_value_size_ret returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is NULL, it is ignored.

cl_platform_info	**Return Type**	**Description**
CL_PLATFORM_PROFILE	char[][2]	OpenCL profile string. Returns the profile name supported by the implementation. The profile name returned can be one of the following strings: FULL_PROFILE – if the implementation supports the OpenCL specification (functionality defined as part of the core specification and does not require any extensions to be supported). EMBEDDED_PROFILE - if the implementation supports the OpenCL embedded profile. The embedded profile is defined to be a subset for each version of OpenCL. The embedded profile for OpenCL 1.1 is described in *section 10*.
CL_PLATFORM_VERSION	char[]	OpenCL version string. Returns the OpenCL version supported by the implementation. This version string

---

[2] A null terminated string is returned by OpenCL query function calls if the return type of the information being queried is a char[].

		has the following format: *OpenCL<space><major_version.minor_version><space><platform-specific information>* The *major_version.minor_version* value returned will be1.1.
CL_PLATFORM_NAME	char[]	Platform name string.
CL_PLATFORM_VENDOR	char[]	Platform vendor string.
CL_PLATFORM_EXTENSIONS	char[]	Returns a space separated list of extension names (the extension names themselves do not contain any spaces) supported by the platform. Extensions defined here must be supported by all devices associated with this platform.

Table 4.1. *OpenCL Platform Queries*

clGetPlatformInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors[3]:

- CL_INVALID_PLATFORM if *platform* is not a valid platform.
- CL_INVALID_VALUE if *param_name* is not one of the supported values or if size in bytes specified by *param_value_size* is < size of return type as specified in *table 4.1* and *param_value* is not a NULL value.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

4.2 Querying Devices

The list of devices available on a platform can be obtained using the following function.

cl_int **clGetDeviceIDs**[4] (cl_platform_id *platform*,
cl_device_type *device_type,*
cl_uint *num_entries*,
cl_device_id **devices*,
cl_uint **num_devices*)

---

[3] The OpenCL specification does not describe the order of precedence for error codes returned by API calls.

[4] **clGetDeviceIDs** may return all or a subset of the actual physical devices present in the platform and that match *device_type*.

platform refers to the platform ID returned by **clGetPlatformIDs** or can be NULL. If *platform* is NULL, the behavior is implementation-defined.

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

cl_device_type	**Description**
CL_DEVICE_TYPE_CPU	An OpenCL device that is the host processor. The host processor runs the OpenCL implementations and is a single or multi-core CPU.
CL_DEVICE_TYPE_GPU	An OpenCL device that is a GPU. By this we mean that the device can also be used to accelerate a 3D API such as OpenGL or DirectX.
CL_DEVICE_TYPE_ACCELERATOR	Dedicated OpenCL accelerators (for example the IBM CELL Blade). These devices communicate with the host processor using a peripheral interconnect such as PCIe.
CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the system.
CL_DEVICE_TYPE_ALL	All OpenCL devices available in the system.

Table 4.2. *List of OpenCL Device Categories*

num_entries is the number of cl_device entries that can be added to *devices*. If *devices* is not NULL, the *num_entries* must be greater than zero.

devices returns a list of OpenCL devices found. The cl_device_id values returned in *devices* can be used to identify a specific OpenCL device. If *devices* argument is NULL, this argument is ignored. The number of OpenCL devices returned is the minimum of the value specified by *num_entries* or the number of OpenCL devices whose type matches *device_type*.

num_devices returns the number of OpenCL devices available that match *device_type*. If *num_devices* is NULL, this argument is ignored.

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

- CL_INVALID_PLATFORM if *platform* is not a valid platform.
- CL_INVALID_DEVICE_TYPE if *device_type* is not a valid value.
- CL_INVALID_VALUE if *num_entries* is equal to zero and *devices* is not NULL or if both *num_devices* and *devices* are NULL.

- CL_DEVICE_NOT_FOUND if no OpenCL devices that matched *device_type* were found.
- 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 application can query specific capabilities of the OpenCL device(s) returned by **clGetDeviceIDs**. This can be used by the application to determine which device(s) to use.

The function

```
cl_int     clGetDeviceInfo (cl_device_id device,
                            cl_device_info param_name,
                            size_t param_value_size,
                            void *param_value,
                            size_t *param_value_size_ret)
```

gets specific information about an OpenCL device. The information that can be queried using **clGetDeviceInfo** is specified in *table 4.3*.

device is a device returned by **clGetDeviceIDs**.

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

param_value is a pointer to memory location where appropriate values for a given *param_name* as specified in *table 4.3* will be returned. If *param_value* is NULL, it is ignored.

param_value_size specifies the size in bytes of memory pointed to by *param_value*. This size in bytes must be >= size of return type specified in *table 4.3.*

param_value_size_ret returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is NULL, it is ignored.

cl_device_info	**Return Type**	**Description**
CL_DEVICE_TYPE	cl_device_type	The OpenCL device type. Currently supported values are: CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_ACCELERATOR, CL_DEVICE_TYPE_DEFAULT or a

		combination of the above.
CL_DEVICE_VENDOR_ID	cl_uint	A unique device vendor identifier. An example of a unique device identifier could be the PCIe ID.
CL_DEVICE_MAX_COMPUTE_UNITS	cl_uint	The number of parallel compute cores on the OpenCL device. The minimum value is 1.
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	cl_uint	Maximum dimensions that specify the global and local work-item IDs used by the data parallel execution model. (Refer to **clEnqueueNDRangeKernel**). The minimum value is 3.
CL_DEVICE_MAX_WORK_ITEM_SIZES	size_t []	Maximum number of work-items that can be specified in each dimension of the work-group to **clEnqueueNDRangeKernel**. Returns *n* size_t entries, where *n* is the value returned by the query for CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS. The minimum value is (1, 1, 1).
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t	Maximum number of work-items in a work-group executing a kernel using the data parallel execution model. (Refer to **clEnqueueNDRangeKernel**). The minimum value is 1.
CL_DEVICE_PREFERRED_ VECTOR_WIDTH_CHAR **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_SHORT** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_INT** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_LONG** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_FLOAT** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_DOUBLE** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_HALF**	cl_uint	Preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. If the **cl_khr_fp64** extension is not supported, CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE must return 0. If the **cl_khr_fp16** extension is not supported, CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF must return 0.
CL_DEVICE_NATIVE_ VECTOR_WIDTH_CHAR **CL_DEVICE_NATIVE_**	cl_uint	Returns the native ISA vector width. The vector width is defined as the number of scalar elements that can be

VECTOR_WIDTH_SHORT **CL_DEVICE_NATIVE_ VECTOR_WIDTH_INT** **CL_DEVICE_NATIVE_ VECTOR_WIDTH_LONG** **CL_DEVICE_NATIVE_ VECTOR_WIDTH_FLOAT** **CL_DEVICE_NATIVE_ VECTOR_WIDTH_DOUBLE** **CL_DEVICE_NATIVE_ VECTOR_WIDTH_HALF**		stored in the vector. If the **cl_khr_fp64** extension is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_ DOUBLE must return 0. If the **cl_khr_fp16** extension is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_ HALF must return 0.
CL_DEVICE_MAX_CLOCK_FREQUENCY	cl_uint	Maximum configured clock frequency of the device in MHz.
CL_DEVICE_ADDRESS_BITS	cl_uint	The default compute device address space size specified as an unsigned integer value in bits. Currently supported values are 32 or 64 bits.
CL_DEVICE_MAX_MEM_ALLOC_SIZE	cl_ulong	Max size of memory object allocation in bytes. The minimum value is max (1/4th of **CL_DEVICE_GLOBAL_MEM_SIZE** , 128*1024*1024)
CL_DEVICE_IMAGE_SUPPORT	cl_bool	Is CL_TRUE if images are supported by the OpenCL device and CL_FALSE otherwise.
CL_DEVICE_MAX_READ_IMAGE_ARGS	cl_uint	Max number of simultaneous image objects that can be read by a kernel. The minimum value is 128 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	cl_uint	Max number of simultaneous image objects that can be written to by a kernel. The minimum value is 8 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE2D_MAX_WIDTH	size_t	Max width of 2D image in pixels. The minimum value is 8192 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE2D_MAX_HEIGHT	size_t	Max height of 2D image in pixels. The minimum value is 8192 if

		CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE3D_MAX_WIDTH	size_t	Max width of 3D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE3D_MAX_HEIGHT	size_t	Max height of 3D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE3D_MAX_DEPTH	size_t	Max depth of 3D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_SAMPLERS	cl_uint	Maximum number of samplers that can be used in a kernel. Refer to *section 6.11.13* for a detailed description on samplers. The minimum value is 16 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_PARAMETER_SIZE	size_t	Max size in bytes of the arguments that can be passed to a kernel. The minimum value is 1024. For this minimum value, only a maximum of 128 arguments can be passed to a kernel.
CL_DEVICE_MEM_BASE_ADDR_ALIGN	cl_uint	Describes the alignment in bits of the base address of any allocated memory object.
CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE	cl_uint	The smallest alignment in bytes which can be used for any data type.
CL_DEVICE_SINGLE_FP_CONFIG	cl_device_ fp_config	Describes single precision floating-point capability of the device. This is a bit-field that describes one or more of the following values: CL_FP_DENORM – denorms are

		supported CL_FP_INF_NAN – INF and quiet NaNs are supported. CL_FP_ROUND_TO_NEAREST– round to nearest even rounding mode supported CL_FP_ROUND_TO_ZERO – round to zero rounding mode supported CL_FP_ROUND_TO_INF – round to +ve and –ve infinity rounding modes supported CL_FP_FMA – IEEE754-2008 fused multiply-add is supported. CL_FP_SOFT_FLOAT – Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. The mandated minimum floating-point capability is: CL_FP_ROUND_TO_NEAREST | CL_FP_INF_NAN.
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	cl_device_mem_cache_type	Type of global memory cache supported. Valid values are: CL_NONE, CL_READ_ONLY_CACHE and CL_READ_WRITE_CACHE.
CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE	cl_uint	Size of global memory cache line in bytes.
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE	cl_ulong	Size of global memory cache in bytes.
CL_DEVICE_GLOBAL_MEM_SIZE	cl_ulong	Size of global device memory in bytes.
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	cl_ulong	Max size in bytes of a constant buffer allocation. The minimum value is 64 KB.
CL_DEVICE_MAX_CONSTANT_ARGS	cl_uint	Max number of arguments declared with the __constant qualifier in a kernel. The minimum value is 8.
CL_DEVICE_LOCAL_MEM_TYPE	cl_device_local_mem_type	Type of local memory supported. This can be set to CL_LOCAL implying dedicated local memory storage such as SRAM, or CL_GLOBAL.

CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong	Size of local memory arena in bytes. The minimum value is 32 KB.
CL_DEVICE_ERROR_CORRECTION_SUPPORT	cl_bool	Is CL_TRUE if the device implements error correction for all accesses to compute device memory (global and constant). Is CL_FALSE if the device does not implement such error correction.
CL_DEVICE_HOST_UNIFIED_MEMORY	cl_bool	Is CL_TRUE if the device and the host have a unified memory subsystem and is CL_FALSE otherwise.
CL_DEVICE_PROFILING_TIMER_RESOLUTION	size_t	Describes the resolution of device timer. This is measured in nanoseconds. Refer to *section 5.12* for details.
CL_DEVICE_ENDIAN_LITTLE	cl_bool	Is CL_TRUE if the OpenCL device is a little endian device and CL_FALSE otherwise.
CL_DEVICE_AVAILABLE	cl_bool	Is CL_TRUE if the device is available and CL_FALSE if the device is not available.
CL_DEVICE_COMPILER_AVAILABLE	cl_bool	Is CL_FALSE if the implementation does not have a compiler available to compile the program source. Is CL_TRUE if the compiler is available. This can be CL_FALSE for the embedded platform profile only.
CL_DEVICE_EXECUTION_CAPABILITIES	cl_device_exec_capabilities	Describes the execution capabilities of the device. This is a bit-field that describes one or more of the following values: CL_EXEC_KERNEL – The OpenCL device can execute OpenCL kernels. CL_EXEC_NATIVE_KERNEL – The OpenCL device can execute native kernels.

		The mandated minimum capability is: CL_EXEC_KERNEL.
CL_DEVICE_QUEUE_PROPERTIES	cl_command_ queue_properties	Describes the command-queue properties supported by the device. This is a bit-field that describes one or more of the following values: CL_QUEUE_OUT_OF_ORDER_EXEC_ MODE_ENABLE CL_QUEUE_PROFILING_ENABLE These properties are described in *table 5.1*. The mandated minimum capability is: CL_QUEUE_PROFILING_ENABLE.
CL_DEVICE_PLATFORM	cl_platform_id	The platform associated with this device.
CL_DEVICE_NAME	char[]	Device name string.
CL_DEVICE_VENDOR	char[]	Vendor name string.
CL_DRIVER_VERSION	char[]	OpenCL software driver version string in the form *major_number.minor_number*
CL_DEVICE_PROFILE[5]	char[]	OpenCL profile string. Returns the profile name supported by the device. The profile name returned can be one of the following strings: FULL_PROFILE – if the device supports the OpenCL specification (functionality defined as part of the core specification and does not require any extensions to be supported). EMBEDDED_PROFILE - if the device supports the OpenCL embedded profile.

---

[5] The platform profile returns the profile that is implemented by the OpenCL framework. If the platform profile returned is FULL_PROFILE, the OpenCL framework will support devices that are FULL_PROFILE and may also support devices that are EMBEDDED_PROFILE. The compiler must be available for all devices i.e. CL_DEVICE_COMPILER_AVAILABLE is CL_TRUE. If the platform profile returned is EMBEDDED_PROFILE, then devices that are only EMBEDDED_PROFILE are supported.

CL_DEVICE_VERSION	char[]	OpenCL version string. Returns the OpenCL version supported by the device. This version string has the following format: *OpenCL<space><major_version.minor_version><space><vendor-specific information>* The *major_version.minor_version* value returned will be 1.1.
CL_DEVICE_OPENCL_C_VERSION	char[]	OpenCL C version string. Returns the highest OpenCL C version supported by the compiler for this device. This version string has the following format: *OpenCL<space>C<space><major_version.minor_version><space><vendor-specific information>* The *major_version.minor_version* value returned must be 1.1 if CL_DEVICE_VERSION is OpenCL 1.1. The *major_version.minor_version* value returned can be 1.0 or 1.1 if CL_DEVICE_VERSION is OpenCL 1.0. If OpenCL C 1.1 is returned, this implies that the language feature set defined in *section 6* of the OpenCL 1.1 specification is supported by the OpenCL 1.0 device.
CL_DEVICE_EXTENSIONS	char[]	Returns a space separated list of extension names (the extension names themselves do not contain any spaces) supported by the device. The list of extension names returned can be vendor supported extension names and one or more of the following Khronos approved extension names: **cl_khr_fp64** **cl_khr_int64_base_atomics** **cl_khr_int64_extended_atomics** **cl_khr_fp16** **cl_khr_gl_sharing** **cl_khr_gl_event**

		cl_khr_d3d10_sharing Please refer to *section 9* for a detailed description of these extensions.

Table 4.3. *OpenCL Device Queries*

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

- CL_INVALID_DEVICE if *device* is not valid.
- CL_INVALID_VALUE if *param_name* is not one of the supported values or if size in bytes specified by *param_value_size* is < size of return type as specified in *table 4.3* and *param_value* is not a NULL value.
- 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.

4.3 Contexts

The function

```
cl_context    clCreateContext (const cl_context_properties *properties,
                               cl_uint num_devices,
                               const cl_device_id *devices,
                               void (CL_CALLBACK *pfn_notify)(const char *errinfo,
                                                 const void *private_info, size_t cb,
                                                 void *user_data),
                               void *user_data,
                               cl_int *errcode_ret)
```

creates an OpenCL context. An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the 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.

cl_context_properties enum	Property value	Description
CL_CONTEXT_PLATFORM	cl_platform_id	Specifies the platform to use.

Table 4.4 *List of supported properties by* **clCreateContext**

num_devices is the number of devices specified in the *devices* argument.

devices is a pointer to a list of unique devices[6] returned by **clGetDeviceIDs** for a platform.

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

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

---

[6] Duplicate devices specified in *devices* are ignored.

- *user_data* is a pointer to user supplied data.

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

NOTE: There are a number of cases where error notifications need to be delivered due to an error that occurs outside a context. Such notifications may not be delivered through the *pfn_notify* callback. Where these notifications go is implementation-defined.

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

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

clCreateContext returns a valid non-zero context and *errcode_ret* is set to CL_SUCCESS if the context is created successfully. Otherwise, it returns a NULL value with the following error values returned in *errcode_ret*:

- CL_INVALID_PLATFORM if *properties* is NULL and no platform could be selected or if platform value specified in *properties* is not a valid platform.

- 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 NULL.

- CL_INVALID_VALUE if *num_devices* is 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 **clGetDeviceIDs**.

- 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 function

cl_context
clCreateContextFromType[7] (const cl_context_properties **properties,*
cl_device_type *device_type*,
void (CL_CALLBACK **pfn_notify*)(*const char *errinfo,*
*const void *private_info, size_t cb,*
*void *user_data*),
void **user_data*,
cl_int **errcode_ret*)

creates an OpenCL context from a device type that identifies the specific device(s) to use.

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 of supported properties is described in *table 4.4*. *properties* can also be NULL in which case the platform that is selected is implementation-defined.

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

pfn_notify and *user_data* are described in **clCreateContext**.

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

clCreateContextFromType returns a valid non-zero context and *errcode_ret* is set to CL_SUCCESS if the context is created successfully. Otherwise, it returns a NULL value with the following error values returned in *errcode_ret*:

- CL_INVALID_PLATFORM if *properties* is NULL and no platform could be selected or if platform value specified in *properties* is not a valid platform.
- CL_INVALID_PROPERTY if context property name in *properties* is not a supported property name, if the value specified for a supported property name is not valid, or if the same property name is specified more than once.
- CL_INVALID_VALUE if *pfn_notify* is NULL but *user_data* is not NULL.
- CL_INVALID_DEVICE_TYPE if *device_type* is not a valid value.
- CL_DEVICE_NOT_AVAILABLE if no devices that match *device_type* and property values specified in *properties* are currently available.

---

[7] **clCreateContextfromType** may return all or a subset of the actual physical devices present in the platform and that match *device_type*.

- CL_DEVICE_NOT_FOUND if no devices that match *device_type* and property values specified in *properties* were found.

- 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 function

cl_int **clRetainContext** (cl_context *context*)

increments the *context* reference count. **clRetainContext** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if *context* is not a valid OpenCL 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.

clCreateContext and **clCreateContextFromType** perform an implicit retain. This is very helpful for 3rd party libraries, which typically get a context passed to them by the application. However, it is possible that the application may delete the context without informing the library. Allowing functions to attach to (i.e. retain) and release a context solves the problem of a context being used by a library no longer being valid.

The function

cl_int **clReleaseContext** (cl_context *context*)

decrements the *context* reference count. **clReleaseContext** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT if *context* is not a valid OpenCL 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.

After the *context* reference count becomes zero and all the objects attached to *context* (such as memory objects, command-queues) are released, the *context* is deleted.

The function

```
cl_int          clGetContextInfo (cl_context context,
                                  cl_context_info param_name,
                                  size_t param_value_size,
                                  void *param_value,
                                  size_t *param_value_size_ret)
```

can be used to query information about a context.

context specifies the OpenCL context being queried.

param_name is an enumeration constant that specifies the information to query.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of return type as described in *table 4.5*.

param_value_size_ret returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is NULL, it is ignored.

The list of supported *param_name* values and the information returned in *param_value* by **clGetContextInfo** is described in *table 4.5*.

cl_context_info	**Return Type**	**Information returned in param_value**
CL_CONTEXT_REFERENCE_COUNT[8]	cl_uint	Return the *context* reference count.
CL_CONTEXT_NUM_DEVICES	cl_uint	Return the number of devices in *context*.
CL_CONTEXT_DEVICES	cl_device_id[]	Return the list of devices in *context*.
CL_CONTEXT_PROPERTIES	cl_context_properties[]	Return the *properties* argument specified in **clCreateContext** or **clCreateContextFromType**. If the *properties* argument specified in

---

[8] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

		clCreateContext or **clCreateContextFromType** used to create *context* is not NULL, the implementation must return the values specified in the *properties* argument. If the *properties* argument specified in **clCreateContext** or **clCreateContextFromType** used to create *context* is NULL, the implementation may return either a *param_value_size_ret* of 0 i.e. there is no context property value to be returned or can return a context property value of 0 (where 0 is used to terminate the context properties list) in the memory that *param_value* points to.

Table 4.5 *List of supported param_names by* **clGetContextInfo**

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

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if *param_name* is not one of the supported values or if size in bytes specified by *param_value_size* is < size of return type as specified in *table 4.5* and *param_value* is not a NULL value.
- 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.

5. The OpenCL Runtime

In this section we describe the API calls that manage OpenCL objects such as command-queues, memory objects, program objects, kernel objects for `__kernel` functions in a program and calls that allow you to enqueue commands to a command-queue such as executing a kernel, reading, or writing a memory object.

5.1 Command Queues

OpenCL objects such as memory, program and kernel objects are created using a context. Operations on these objects are performed using a command-queue. The command-queue can be used to queue a set of operations (referred to as commands) in order. Having multiple command-queues allows applications to queue multiple independent commands without requiring synchronization. Note that this should work as long as these objects are not being shared. Sharing of objects across multiple command-queues will require the application to perform appropriate synchronization. This is described in *Appendix A*.

The function

cl_command_queue **clCreateCommandQueue** (cl_context *context*,
cl_device_id *device*,
cl_command_queue_properties *properties*,
cl_int **errcode_ret*)

creates a command-queue on a specific device.

context must be a valid OpenCL context.

Command-Queue Properties	Description
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	Determines whether the commands queued in the command-queue are executed in-order or out-of-order. If set, the commands in the command-queue are executed out-of-order. Otherwise, commands are executed in-order. For a detailed description about CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, refer to *section 5.11*.
CL_QUEUE_PROFILING_ENABLE	Enable or disable profiling of commands in the command-queue. If set, the profiling of commands is enabled. Otherwise profiling of commands is disabled.

	For a detailed description, refer to *section 5.12*.

Table 5.1 *List of supported cl_command_queue_property values and description.*

device must be a device associated with *context*. It can either be in the list of devices specified when *context* is created using **clCreateContext** or have the same device type as device type specified when *context* is created using **clCreateContextFromType**.

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..

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

clCreateCommandQueue returns a valid non-zero command-queue and *errcode_ret* is set to CL_SUCCESS if the command-queue is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- 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 function

cl_int **clRetainCommandQueue** (cl_command_queue *command_queue*)

increments the *command_queue* reference count. **clRetainCommandQueue** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- 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.

clCreateCommandQueue performs an implicit retain. This is very helpful for 3rd party libraries, which typically get a command-queue passed to them by the application. However, it is possible that the application may delete the command-queue without informing the library. Allowing functions to attach to (i.e. retain) and release a command-queue solves the problem of a command-queue being used by a library no longer being valid.

The function

cl_int **clReleaseCommandQueue** (cl_command_queue *command_queue*)

decrements the *command_queue* reference count. **clReleaseCommandQueue** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- 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.

After the *command_queue* reference count becomes zero and all commands queued to *command_queue* have finished (eg. kernel executions, memory object updates etc.), the command-queue is deleted.

clReleaseCommandQueue performs an implicit flush to issue any previously queued OpenCL commands in *command_queue*.

The function

cl_int **clGetCommandQueueInfo** (cl_command_queue *command_queue*,
cl_command_queue_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

can be used to query information about a command-queue.

command_queue specifies the command-queue being queried.

param_name specifies the information to query.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.2*. If *param_value* is NULL, it is ignored.

param_value_size_ret returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is NULL, it is ignored.

The list of supported *param_name* values and the information returned in *param_value* by **clGetCommandQueueInfo** is described in *table 5.2*.

cl_command_queue_info	**Return Type**	**Information returned in param_value**
CL_QUEUE_CONTEXT	cl_context	Return the context specified when the command-queue is created.
CL_QUEUE_DEVICE	cl_device_id	Return the device specified when the command-queue is created.
CL_QUEUE_REFERENCE_COUNT[9]	cl_uint	Return the command-queue reference count.
CL_QUEUE_PROPERTIES	cl_command_queue_properties	Return the currently specified properties for the command-queue. These properties are specified by the *properties* argument in **clCreateCommandQueue**.

Table 5.2 *List of supported param_names by clGetCommandQueueInfo*

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_VALUE if *param_name* is not one of the supported values or if size in bytes specified by *param_value_size* is < size of return type as specified in *table 5.2* and *param_value* is not a NULL value.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.

---

[9] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

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

NOTE

It is possible that a device(s) becomes unavailable after a context and command-queues that use this device(s) have been created and commands have been queued to command-queues. In this case the behavior of OpenCL API calls that use this context (and command-queues) are considered to be implementation-defined. The user callback function, if specified, when the context is created can be used to record appropriate information in the *errinfo*, *private_info* arguments passed to the callback function when the device becomes unavailable.

5.2 Buffer Objects

A *buffer* object stores a one-dimensional collection of elements. Elements of a *buffer* object can be a scalar data type (such as an int, float), vector data type, or a user-defined structure.

5.2.1 Creating Buffer Objects

A **buffer object** is created using the following function

cl_mem **clCreateBuffer** (cl_context *context*,
cl_mem_flags *flags*,
size_t *size*,
void **host_ptr*,
cl_int **errcode_ret*)

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 possible values for *flags*:

cl_mem_flags	Description
CL_MEM_READ_WRITE	This flag specifies that the memory object will be read and written by a kernel. This is the default.
CL_MEM_WRITE_ONLY	This flag specifies that the memory object will be written but not read by a kernel. Reading from a buffer or image object created with CL_MEM_WRITE_ONLY inside a kernel is undefined.
CL_MEM_READ_ONLY	This flag specifies that the memory object is a read-only memory object when used inside a kernel. Writing to a buffer or image object created with CL_MEM_READ_ONLY inside a kernel is undefined.
CL_MEM_USE_HOST_PTR	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to use memory referenced by *host_ptr* as the storage bits for the memory object.

	OpenCL implementations are allowed to cache the buffer contents pointed to by *host_ptr* in device memory. This cached copy can be used when kernels are executed on a device. The result of OpenCL commands that operate on multiple buffer objects created with the same *host_ptr* or overlapping host regions is considered to be undefined.
CL_MEM_ALLOC_HOST_PTR	This flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory. CL_MEM_ALLOC_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.
CL_MEM_COPY_HOST_PTR	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by *host_ptr*. CL_MEM_COPY_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive. CL_MEM_COPY_HOST_PTR can be used with CL_MEM_ALLOC_HOST_PTR to initialize the contents of the cl_mem object allocated using host-accessible (e.g. PCIe) memory.

Table 5.3 *List of supported cl_mem_flags values*

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 >= *size* bytes.

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

clCreateBuffer returns a valid non-zero buffer object and *errcode_ret* is set to CL_SUCCESS if the buffer object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- 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[10].
- 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 function

```
cl_mem      clCreateSubBuffer (cl_mem buffer,
                               cl_mem_flags flags,
                               cl_buffer_create_type buffer_create_type,
                               const void *buffer_create_info,
                               cl_int *errcode_ret)
```

can be used to create a new buffer object (referred to as a sub-buffer object) from an existing buffer object.

buffer must be a valid buffer object and cannot be a sub-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*.

cl_buffer_create_type	**Description**
CL_BUFFER_CREATE_TYPE_REGION	Create a buffer object that represents a specific region in *buffer*.

---

[10] Implementations may return CL_INVALID_BUFFER_SIZE if *size* is greater than CL_DEVICE_MAX_MEM_ALLOC_SIZE value specified in *table 4.3* for all *devices* in context.

	buffer_create_info is a pointer to the following structure:

```
typedef struct _cl_buffer_region {
    size_t origin;
    size_t size;
} cl_buffer_region;
```

	(*origin*, *size*) defines the offset and size in bytes in *buffer*. If *buffer* is created with CL_MEM_USE_HOST_PTR, the *host_ptr* associated with the buffer object returned is *host_ptr* + *origin*. The buffer object returned references the data store allocated for *buffer* and points to a specific region given by (*origin*, *size*) in this data store. CL_INVALID_VALUE is returned in *errcode_ret* if the region specified by (*origin*, *size*) is out of bounds in *buffer*. CL_MISALIGNED_SUB_BUFFER_OFFSET is returned in *errcode_ret* if there are no devices in context associated with *buffer* for which the *origin* value is aligned to the CL_DEVICE_MEM_BASE_ADDR_ALIGN value.

Table 5.4 *List of supported names and values in clCreateSubBuffer.*

clCreateSubBuffer returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors in *errcode_ret*:

- CL_INVALID_MEM_OBJECT if *buffer* is not a valid buffer object or is a sub-buffer object.
- 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.

- 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.

NOTE:

The implementation may return the same **cl_mem** object with the reference count incremented appropriately for multiple calls to **clCreateSubBuffer** that use the same values for *buffer*, *flags*, *buffer_create_type* and *buffer_create_info* points to the same descriptor or descriptors that describe values that are exactly the same.

The result of OpenCL commands that read from and write to multiple sub-buffer objects created using **clCreateSubBuffer** with the same buffer object but represent overlapping regions in the buffer object is undefined. The result of OpenCL commands that read from and write to a buffer object and its sub-buffer object(s) created using **clCreateSubBuffer** with the same buffer object is undefined. OpenCL commands that only read from multiple sub-buffer objects created using **clCreateSubBuffer** with the same buffer object but represent overlapping regions in the buffer object or read from a buffer object and its sub-buffer objects should work as defined.

5.2.2 Reading, Writing and Copying Buffer Objects

The following functions enqueue commands to read from a buffer object to host memory or write to a buffer object from host memory.

```
cl_int  clEnqueueReadBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_read,
                             size_t offset,
                             size_t cb,
                             void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```

```
cl_int  clEnqueueWriteBuffer (cl_command_queue command_queue,
                              cl_mem buffer,
                              cl_bool blocking_write,
                              size_t offset,
                              size_t cb,
                              const void *ptr,
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)
```

command_queue refers to the command-queue in which the read / write command will be queued. *command_queue* and *buffer* must be created with the same OpenCL context.

buffer refers to a valid buffer object.

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

If *blocking_read* is CL_TRUE i.e. the read command is blocking, **clEnqueueReadBuffer** 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, **clEnqueueReadBuffer** 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 **clEnqueueWriteBuffer** call returns.

If *blocking_write* is CL_FALSE, the OpenCL implementation will use *ptr* to perform a non-blocking 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.

cb 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.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *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.

clEnqueueReadBuffer and **clEnqueueWriteBuffer** return CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- 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 *event_wait_list* 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*, *cb*) is out of bounds or if *ptr* is a NULL value.

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* 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_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the read and write operations are blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.

- 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 following functions enqueue commands to read a 2D or 3D rectangular region from a buffer object to host memory or write a 2D or 3D rectangular region to a buffer object from host memory.

cl_int **clEnqueueReadBufferRect** (cl_command_queue *command_queue*,
cl_mem *buffer*,
cl_bool *blocking_read*,
const size_t *buffer_origin*[3],
const size_t *host_origin*[3],
const size_t *region*[3],
size_t *buffer_row_pitch,*
size_t *buffer_slice_pitch*,
size_t *host_row_pitch*,
size_t *host_slice_pitch*,
void **ptr*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

cl_int **clEnqueueWriteBufferRect** (cl_command_queue *command_queue*,
cl_mem *buffer*,
cl_bool *blocking_write*,
const size_t *buffer_origin*[3],
const size_t *host_origin*[3],
const size_t *region*[3],
size_t *buffer_row_pitch,*
size_t *buffer_slice_pitch*,
size_t *host_row_pitch*,
size_t *host_slice_pitch*,
const void **ptr*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

command_queue refers to the command-queue in which the read / write command will be queued. *command_queue* and *buffer* must be created with the same OpenCL context.

buffer refers to a valid buffer object.

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

If *blocking_read* is CL_TRUE i.e. the read command is blocking, **clEnqueueReadBufferRect** 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, **clEnqueueReadBufferRect** 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 **clEnqueueWriteBufferRect** call returns.

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

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

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

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

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

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

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

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

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

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *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.

clEnqueueReadBufferRect and **clEnqueueWriteBufferRect** return CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- 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 *event_wait_list* 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_origin*, *region*) is out of bounds.

- CL_INVALID_VALUE if *ptr* is a NULL value.

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* 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_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the read and write operations are blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.

- 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.

NOTE:

Calling **clEnqueueReadBuffer** to read a region of the buffer object with the *ptr* argument value set to *host_ptr* + *offset*, where *host_ptr* is a pointer to the memory region specified when the buffer object being read is created with CL_MEM_USE_HOST_PTR, must meet the following requirements in order to avoid undefined behavior:

- All commands that use this buffer object or a memory object (buffer or image) created from this buffer object have finished execution before the read command begins execution.

- The buffer object or memory objects created from this buffer object are not mapped.

- The buffer object or memory objects created from this buffer object are not used by any command-queue until the read command has finished execution.

Calling **clEnqueueReadBufferRect** to read a region of the buffer object with the *ptr* argument value set to *host_ptr* and *host_origin*, *buffer_origin* values are the same, where *host_ptr* is a pointer to the memory region specified when the buffer object being read is created with CL_MEM_USE_HOST_PTR, must meet the same requirements given above for **clEnqueueReadBuffer**.

Calling **clEnqueueWriteBuffer** to update the latest bits in a region of the buffer object with the *ptr* argument value set to *host_ptr* + *offset*, where *host_ptr* is a pointer to the memory region specified when the buffer object being written is created with CL_MEM_USE_HOST_PTR, must meet the following requirements in order to avoid undefined behavior:

- The host memory region given by (*host_ptr* + *offset*, *cb*) contains the latest bits when the enqueued write command begins execution.

- The buffer object or memory objects created from this buffer object are not mapped.

- The buffer object or memory objects created from this buffer object are not used by any command-queue until the write command has finished execution.

Calling **clEnqueueWriteBufferRect** to update the latest bits in a region of the buffer object with the *ptr* argument value set to *host_ptr* and *host_origin*, *buffer_origin* values are the same, where *host_ptr* is a pointer to the memory region specified when the buffer object being written is created with CL_MEM_USE_HOST_PTR, must meet the following requirements in order to avoid undefined behavior:

- The host memory region given by (*buffer_origin region*) contains the latest bits when the enqueued write command begins execution.

- The buffer object or memory objects created from this buffer object are not mapped.

- The buffer object or memory objects created from this buffer object are not used by any command-queue until the write command has finished execution.

The function

```
cl_int  clEnqueueCopyBuffer (cl_command_queue command_queue,
                             cl_mem src_buffer,
                             cl_mem dst_buffer,
                             size_t src_offset,
                             size_t dst_offset,
                             size_t cb,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```

enqueues a command to copy a buffer object identified by *src_buffer* to another buffer object identified by *dst_buffer*.

command_queue refers to the command-queue in which the copy command will be queued. The OpenCL context associated with *command_queue*, *src_buffer* and *dst_buffer* must be the same.

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

dst_offset refers to the offset where to begin copying data into *dst_buffer*.

cb refers to the size in bytes to copy.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *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. **clEnqueueBarrier** can be used instead.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_buffer* and *dst_buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* 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*, *dst_offset*, *cb*, *src_offset* + *cb* or *dst_offset* + *cb* require accessing elements outside the *src_buffer* and *dst_buffer* buffer objects respectively.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* 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 function

cl_int **clEnqueueCopyBufferRect** (cl_command_queue *command_queue*,
cl_mem *src_buffer*,
cl_mem *dst_buffer*,
const size_t *src_origin*[3],
const size_t *dst_origin*[3],
const size_t *region*[3],
size_t *src_row_pitch,*
size_t *src_slice_pitch*,
size_t *dst_row_pitch*,
size_t *dst_slice_pitch*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

enqueues a command to copy a 2D or 3D rectangular region from the buffer object identified by *src_buffer* to a 2D or 3D region in the buffer object identified by *dst_buffer*.

command_queue refers to the command-queue in which the copy command will be queued. The OpenCL context associated with *command_queue*, *src_buffer* and *dst_buffer* must be the same.

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

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

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

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

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

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

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

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *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. **clEnqueueBarrier** can be used instead.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_buffer* and *dst_buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* 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_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_MEM_COPY_OVERLAP if *src_buffer* and *dst_buffer* are the same buffer object and the source and destination regions overlap.
- 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_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.

5.2.3 Mapping Buffer Objects

The function

void * **clEnqueueMapBuffer** (cl_command_queue *command_queue*,
cl_mem *buffer,*
cl_bool *blocking_map*,
cl_map_flags *map_flags*,
size_t *offset*,
size_t *cb*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*,
cl_int **errcode_ret*)

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.

command_queue must be a valid command-queue.

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

If *blocking_map* is CL_TRUE, **clEnqueueMapBuffer** 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 **clEnqueueMapBuffer** 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 **clEnqueueMapBuffer**.

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

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same.

event returns an event object that identifies this particular 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.

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

clEnqueueMapBuffer will return a pointer to the mapped region. The *errcode_ret* is set to CL_SUCCESS.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and *buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *buffer* is not a valid buffer object.
- CL_INVALID_VALUE if region being mapped given by (*offset*, *cb*) is out of bounds or if values specified in *map_flags* are not valid.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* 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_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the map operation is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.

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

NOTE:

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

- The *host_ptr* specified in **clCreateBuffer** is guaranteed to contain the latest bits in the region being mapped when the **clEnqueueMapBuffer** command has completed.

- The pointer value returned by **clEnqueueMapBuffer** will be derived from the *host_ptr* specified when the buffer object is created.

Mapped buffer objects are unmapped using **clEnqueueUnmapMemObject**. This is described in *section 5.4.2*.

5.3 Image Objects

An *image* object is used to store a one-, two- or three- dimensional texture, frame-buffer or image. The elements of an image object are selected from a list of predefined image formats. The minimum number of elements in a memory object is one.

5.3.1 Creating Image Objects

A 2D **image** object is created using the following function

```
cl_mem  clCreateImage2D (cl_context context,
                         cl_mem_flags flags,
                         const cl_image_format *image_format,
                         size_t image_width,
                         size_t image_height,
                         size_t image_row_pitch,
                         void *host_ptr,
                         cl_int *errcode_ret)
```

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*.

image_format is a pointer to a structure 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.

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

image_row_pitch is the scan-line pitch in bytes. This must be 0 if *host_ptr* is NULL and can be either 0 or >= *image_width* * size of element in bytes if *host_ptr* is not NULL. If *host_ptr* is not NULL and *image_row_pitch* = 0, *image_row_pitch* is calculated as *image_width* * size of element in bytes. If *image_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 >= *image_row_pitch* * *image_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.

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

clCreateImage2D returns a valid non-zero image object and *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

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

A 3D **image object** is created using the following function

cl_mem **clCreateImage3D (**cl_context *context*,
cl_mem_flags *flags*,
const cl_image_format **image_format*,
size_t *image_width*,
size_t *image_height*,
size_t *image_depth*,
size_t *image_row_pitch*,
size_t *image_slice_pitch*,
void **host_ptr*,
cl_int **errcode_ret*)

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*.

image_format is a pointer to a structure 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.

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

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

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

image_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 >= *image_row_pitch* * *image_height* if *host_ptr* is not NULL. If *host_ptr* is not NULL and *image_slice_pitch* = 0, *image_slice_pitch* is calculated as *image_row_pitch* * *image_height*. If *image_slice_pitch* is not 0, it must be a multiple of the *image_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 >= *image_slice_pitch* * *image_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.

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

clCreateImage3D returns a valid non-zero image object created and the *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if values specified in *flags* are not valid.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in *image_format* are not valid or if *image_format* is NULL.
- CL_INVALID_IMAGE_SIZE if *image_width*, *image_height* are 0 or if *image_depth* <= 1 or if they exceed values specified in CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_MAX_HEIGHT or CL_DEVICE_IMAGE3D_MAX_DEPTH respectively for all devices in *context* or if values specified by *image_row_pitch* and *image_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.

5.3.1.1 Image Format Descriptor

The image format descriptor structure is defined as

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

`image_channel_order` specifies the number of channels and the channel layout i.e. the memory layout in which channels are stored in the image. Valid values are described in *table 5.5*.

Enum values that can be specified in channel_order
CL_R, **CL_Rx** or **CL_A**
CL_INTENSITY. This format can only be used if channel data type = CL_UNORM_INT8, CL_UNORM_INT16, CL_SNORM_INT8, CL_SNORM_INT16, CL_HALF_FLOAT or CL_FLOAT.
CL_LUMINANCE. This format can only be used if channel data type = CL_UNORM_INT8, CL_UNORM_INT16, CL_SNORM_INT8, CL_SNORM_INT16, CL_HALF_FLOAT or CL_FLOAT.
CL_RG, **CL_RGx** or **CL_RA**
CL_RGB or **CL_RGBx**. This format can only be used if channel data type = CL_UNORM_SHORT_565, CL_UNORM_SHORT_555, or CL_UNORM_INT_101010.
CL_RGBA
CL_ARGB, CL_BGRA. This format can only be used if channel data type = CL_UNORM_INT8, CL_SNORM_INT8, CL_SIGNED_INT8 or CL_UNSIGNED_INT8.

Table 5.5 *List of supported Image Channel Order Values*

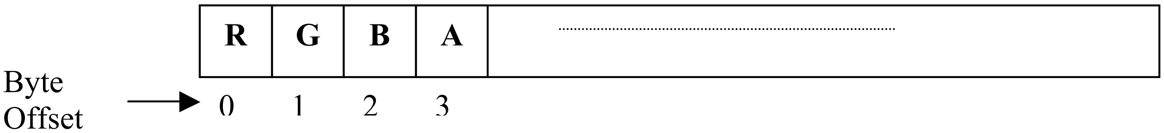
`image_channel_data_type` describes the size of the channel data type. The list of supported values is described in *table 5.6*. The number of bits per element determined by the `image_channel_data_type` and `image_channel_order` must be a power of two.

Image Channel Data Type	Description
CL_SNORM_INT8	Each channel component is a normalized signed 8-bit integer value
CL_SNORM_INT16	Each channel component is a normalized signed 16-bit integer value
CL_UNORM_INT8	Each channel component is a normalized unsigned 8-bit integer value
CL_UNORM_INT16	Each channel component is a normalized unsigned 16-bit integer value

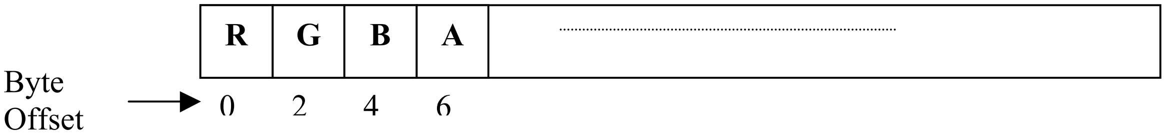
CL_UNORM_SHORT_565	Represents a normalized 5-6-5 3-channel RGB image. The channel order must be CL_RGB or CL_RGBx.
CL_UNORM_SHORT_555	Represents a normalized x-5-5-5 4-channel xRGB image. The channel order must be CL_RGB or CL_RGBx.
CL_UNORM_INT_101010	Represents a normalized x-10-10-10 4-channel xRGB image. The channel order must be CL_RGB or CL_RGBx.
CL_SIGNED_INT8	Each channel component is an unnormalized signed 8-bit integer value
CL_SIGNED_INT16	Each channel component is an unnormalized signed 16-bit integer value
CL_SIGNED_INT32	Each channel component is an unnormalized signed 32-bit integer value
CL_UNSIGNED_INT8	Each channel component is an unnormalized unsigned 8-bit integer value
CL_UNSIGNED_INT16	Each channel component is an unnormalized unsigned 16-bit integer value
CL_UNSIGNED_INT32	Each channel component is an unnormalized unsigned 32-bit integer value
CL_HALF_FLOAT	Each channel component is a 16-bit half-float value
CL_FLOAT	Each channel component is a single precision floating-point value

Table 5.6 *List of supported Image Channel Data Types*

For example, to specify a normalized unsigned 8-bit / channel RGBA image, `image_channel_order` = CL_RGBA, and `image_channel_data_type` = CL_UNORM_INT8. The memory layout of this image format is described below:



Similar, if `image_channel_order` = CL_RGBA and `image_channel_data_type` = CL_SIGNED_INT16, the memory layout of this image format is described below:



`image_channel_data_type` values of CL_UNORM_SHORT_565, CL_UNORM_SHORT_555 and CL_UNORM_INT_101010 are special cases of packed image formats where the channels of each element are packed into a single unsigned short or unsigned int. For these special packed image formats, the channels are normally packed with the first channel in the most significant bits of the bitfield, and successive channels occupying progressively less significant locations. For CL_UNORM_SHORT_565, `R` is in bits `15:11`, `G` is in bits `10:5` and `B` is in bits `4:0`. For CL_UNORM_SHORT_555, bit `15` is undefined, `R` is in bits `14:10`, `G` in bits `9:5` and `B` in bits `4:0`. For CL_UNORM_INT_101010, bits `31:30` are undefined, `R` is in bits `29:20`, `G` in bits `19:10` and `B` in bits `9:0`.

OpenCL implementations must maintain the minimum precision specified by the number of bits in `image_channel_data_type`. If the image format specified by `image_channel_order`, and `image_channel_data_type` cannot be supported by the OpenCL implementation, then the call to **clCreateImage2D or clCreateImage3D** will return a NULL memory object.

5.3.2 Querying List of Supported Image Formats

The function

cl_int **clGetSupportedImageFormats** (cl_context *context*,
cl_mem_flags *flags*,
cl_mem_object_type *image_type*,
cl_uint *num_entries*,
cl_image_format **image_formats*,
cl_uint **num_image_formats*)

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

- Context
- Image type – 2D, or 3D image.
- Image object allocation information

context is a valid OpenCL context on which the image object(s) will 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*.

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

num_entries specifies the number of entries that can be returned in the memory location given by *image_formats*.

image_formats is a pointer to a memory location where the list of supported image formats are returned. Each entry describes a *cl_image_format* structure supported by the OpenCL implementation. If *image_formats* is NULL, it is ignored.

num_image_formats is the actual number of supported image formats for a specific *context* and values specified by *flags*. If *num_image_formats* is NULL, it is ignored.

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

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if *flags* or *image_type* are not valid, or if *num_entries* is 0 and *image_formats* is not 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.

If CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_TRUE, the values assigned to CL_DEVICE_MAX_READ_IMAGE_ARGS, CL_DEVICE_MAX_WRITE_IMAGE_ARGS, CL_DEVICE_IMAGE2D_MAX_WIDTH, CL_DEVICE_IMAGE2D_MAX_HEIGHT, CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_MAX_HEIGHT, CL_DEVICE_IMAGE3D_MAX_DEPTH and CL_DEVICE_MAX_SAMPLERS by the implementation must be greater than or equal to the minimum values specified in *table 4.3*.

5.3.2.1 Minimum List of Supported Image Formats

For 2D and 3D image objects, the mandated minimum list of image formats that must be supported by all devices (for reading and writing) that support images is described in *table 5.7*.

image_num_channels	image_channel_order	image_channel_data_type
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32

		CL_HALF_FLOAT CL_FLOAT
4	CL_BGRA	CL_UNORM_INT8

Table 5.7 *Min. list of supported image formats.*

5.3.3 Reading, Writing and Copying Image Objects

The following functions 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.

cl_int **clEnqueueReadImage** (cl_command_queue *command_queue*,
cl_mem *image*,
cl_bool *blocking_read*,
const size_t *origin*[3],
const size_t *region*[3],
size_t *row_pitch*,
size_t *slice_pitch*,
void **ptr,*
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

cl_int **clEnqueueWriteImage** (cl_command_queue *command_queue*,
cl_mem *image*,
cl_bool *blocking_write*,
const size_t *origin*[3],
const size_t *region*[3],
size_t *input_row_pitch*,
size_t *input_slice_pitch*,
const void * *ptr*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

command_queue refers to the command-queue in which the read / write command will be queued. *command_queue* and *image* must be created with the same OpenCL context.

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 *non-blocking*.

If *blocking_read* is CL_TRUE i.e. the read command is blocking, **clEnqueueReadImage** 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, **clEnqueueReadImage** 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 **clEnqueueWriteImage** call returns.

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

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

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

row_pitch in **clEnqueueReadImage** and *input_row_pitch* in **clEnqueueWriteImage** 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 **clEnqueueReadImage** and *input_slice_pitch* in **clEnqueueWriteImage** 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.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *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.

clEnqueueReadImage and **clEnqueueWriteImage** return CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue* and *image* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if i*mage* 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_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* 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 *command_queue* does not support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the read and write operations are blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.
- 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.

NOTE:

Calling **clEnqueueReadImage** to read a region of the *image* with the *ptr* argument value set to *host_ptr* + (*origin[2] * image slice pitch* + *origin[1] * image row pitch* + *origin[0] * bytes per pixel*), where *host_ptr* is a pointer to the memory region specified when the *image* being read is created with CL_MEM_USE_HOST_PTR, must meet the following requirements in order to avoid undefined behavior:

- All commands that use this image object have finished execution before the read command begins execution.
- The *row_pitch* and *slice_pitch* argument values in **clEnqueueReadImage** must be set to the image row pitch and slice pitch.
- The image object is not mapped.
- The image object is not used by any command-queue until the read command has finished execution.

Calling **clEnqueueWriteImage** to update the latest bits in a region of the *image* with the *ptr* argument value set to *host_ptr* + (*origin[2] * image slice pitch* + *origin[1] * image row pitch* + *origin[0] * bytes per pixel*), where *host_ptr* is a pointer to the memory region specified when the *image* being written is created with CL_MEM_USE_HOST_PTR, must meet the following requirements in order to avoid undefined behavior:

- The host memory region being written contains the latest bits when the enqueued write command begins execution.
- The *input_row_pitch* and *input_slice_pitch* argument values in **clEnqueueWriteImage** must be set to the image row pitch and slice pitch.
- The image object is not mapped.
- The image object is not used by any command-queue until the write command has finished execution.

The function

cl_int **clEnqueueCopyImage** (cl_command_queue *command_queue*,
cl_mem *src_image*,
cl_mem *dst_image*,
const size_t *src_origin*[3],
const size_t *dst_origin*[3],
const size_t *region*[3],
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

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.

command_queue refers to the command-queue in which the copy command will be queued. 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.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *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. **clEnqueueBarrier** can be used instead.

It is currently a requirement that the *src_image* and *dst_image* image memory objects for **clEnqueueCopyImage** 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).

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_image* and *dst_image* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *src_image* and *dst_image* are not valid image objects.
- CL_IMAGE_FORMAT_MISMATCH if *src_image* and *dst_image* do not use the same image format.
- CL_INVALID_VALUE if the 2D or 3D rectangular region specified by *src_origin* and *src_origin* + *region* refers to a region outside *src_image*, or if the 2D or 3D rectangular region specified by *dst_origin* and *dst_origin* + *region* refers to a region outside *dst_image*.
- CL_INVALID_VALUE if *src_image* is a 2D image object and *src_origin*[2] is not equal to 0 or region[2] is not equal to 1.
- CL_INVALID_VALUE if *dst_image* is a 2D image object and *dst_origin*[2] is not equal to 0 or *region*[2] is not equal to 1.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *src_image* are not supported by device associated with *queue*.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *dst_image* are not supported by device associated with *queue*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_image* or *dst_image*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.

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

- CL_INVALID_OPERATION if the device associated with *command_queue* does not support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).

- CL_MEM_COPY_OVERLAP if *src_image* and *dst_image* are the same image object and the source and destination regions overlap.

5.3.4 Copying between Image and Buffer Objects

The function

cl_int **clEnqueueCopyImageToBuffer** (cl_command_queue *command_queue*,
cl_mem *src_image*,
cl_mem *dst_buffer*,
const size_t *src_origin*[3],
const size_t *region*[3],
size_t *dst_offset*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

enqueues a command to copy an image object to a buffer object.

command_queue must be a valid command-queue. The OpenCL context associated with *command_queue*, *src_image* and *dst_buffer* must be the same.

src_image is a valid image object.

dst_buffer is a valid buffer object.

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

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

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

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *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. **clEnqueueBarrier** can be used instead.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_image* and *dst_buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.

- CL_INVALID_MEM_OBJECT if *src_image* is not a valid image object or *dst_buffer* is not a valid buffer object.

- CL_INVALID_VALUE if the 2D or 3D rectangular region specified by *src_origin* and *src_origin* + *region* refers to a region outside *src_image*, or if the region specified by *dst_offset* and *dst_offset* + *dst_cb* to a region outside *dst_buffer*.

- CL_INVALID_VALUE if *src_image* is a 2D image object and *src_origin*[2] is not equal to 0 or *region*[2] is not equal to 1.

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.

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

- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *src_image* are not supported by device associated with *queue*.

- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_image* or *dst_buffer*.

- CL_INVALID_OPERATION if the device associated with *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 function

cl_int **clEnqueueCopyBufferToImage** (cl_command_queue *command_queue*,
cl_mem *src_buffer*,
cl_mem *dst_image*,
size_t *src_offset*,
const size_t *dst_origin*[3],
const size_t *region*[3],
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

enqueues a command to copy a buffer object to an image object.

command_queue must be a valid command-queue. The OpenCL context associated with *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.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *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. **clEnqueueBarrier** can be used instead.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_buffer* and *dst_image* are not the same or if the context associated with *command_queue* and events in *event_wait_list* 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_VALUE if *dst_image* is a 2D image object and *dst_origin*[2] is not equal to 0 or *region*[2] is not equal to 1.

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* 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 *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.

5.3.5 Mapping Image Objects

The function

void * **clEnqueueMapImage** (cl_command_queue *command_queue*,
cl_mem *image,*
cl_bool *blocking_map*,
cl_map_flags *map_flags*,
const size_t *origin*[3],
const size_t *region*[3],
size_t **image_row_pitch*,
size_t **image_slice_pitch*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*,
cl_int **errcode_ret*)

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.

command_queue must be a valid command-queue.

image is a valid image object. The OpenCL context associated with *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, **clEnqueueMapImage** 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 **clEnqueueMapImage** 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 **clEnqueueMapImage**.

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.

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

image_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, *image_slice_pitch* must be a non-NULL value.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before **clEnqueueMapImage** can be executed. If *event_wait_list* is NULL, then **clEnqueueMapImage** does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same.

event returns an event object that identifies this particular 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.

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

clEnqueueMapImage will return a pointer to the mapped region. The *errcode_ret* is set to CL_SUCCESS.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and *image* are not the same or if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *image* is not a valid image object.
- CL_INVALID_VALUE if region being mapped given by (*origin*, *origin+region*) is out of bounds or if values specified in *map_flags* are not valid.
- CL_INVALID_VALUE if *image* is a 2D image object and *origin*[2] is not equal to 0 or *region*[2] is not equal to 1.

- CL_INVALID_VALUE if *image_row_pitch* is NULL.

- CL_INVALID_VALUE if *image* is a 3D image object and *image_slice_pitch* is NULL.

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* 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_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the map operation is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.

- 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 *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 pointer returned maps a 2D or 3D region starting at *origin* and is at least (*image_row_pitch * region[1])* pixels in size for a 2D image, and is at least (*image_slice_pitch * region[2])* 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 **clCreateImage{2D|3D}** is guaranteed to contain the latest bits in the region being mapped when the **clEnqueueMapImage** command has completed.

- The pointer value returned by **clEnqueueMapImage** will be derived from the *host_ptr* specified when the image object is created.

Mapped image objects are unmapped using **clEnqueueUnmapMemObject**. This is described in *section 5.4.2*.

5.3.6 Image Object Queries

To get information that is common to all memory objects (buffer and image objects), use the **clGetMemObjectInfo** function described in *section 5.4.3*.

To get information specific to an image object created with **clCreateImage{2D|3D}**, use the following function

```
cl_int      clGetImageInfo (cl_mem image,
                            cl_image_info param_name,
                            size_t param_value_size,
                            void *param_value,
                            size_t *param_value_size_ret)
```

image specifies the image object being queried.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetImageInfo** is described in *table 5.8*.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.8*.

param_value_size_ret returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is NULL, it is ignored.

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.8* and *param_value* is not NULL.
- CL_INVALID_MEM_OBJECT if *image* is a not a valid image 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.

cl_image_info	**Return type**	**Info. returned in *param_value***
CL_IMAGE_FORMAT	cl_image_format	Return image format descriptor specified when *image* is created with **clCreateImage{2D|3D}**.
CL_IMAGE_ELEMENT_SIZE	size_t	Return size of each element of the image memory object given by *image*. An element is made up of *n* channels. The value of *n* is given in *cl_image_format* descriptor.
CL_IMAGE_ROW_PITCH	size_t	Return size in bytes of a row of elements of the image object given by *image*.
CL_IMAGE_SLICE_PITCH	size_t	Return size in bytes of a 2D slice for the 3D image object given by *image*. For a 2D image object this value will be 0.
CL_IMAGE_WIDTH	size_t	Return width of the image in pixels
CL_IMAGE_HEIGHT	size_t	Return height of the image in pixels
CL_IMAGE_DEPTH	size_t	Return depth of the image in pixels. For a 2D image object, depth = 0.

Table 5.8 *List of supported param_names by clGetImageInfo*

5.4 Querying, Unmapping, Retaining and Releasing Memory Objects

5.4.1 Retaining and Releasing Memory Objects

The function

cl_int **clRetainMemObject** (cl_mem *memobj*)

increments the *memobj* reference count. **clRetainMemObject** 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 (buffer or image 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.

clCreateBuffer, **clCreateSubBuffer**, and **clCreateImage{2D|3D}** perform an implicit retain. **clCreateSubBuffer** also performs an implicit retain on the memory object used to create the sub-buffer or image object.

The function

cl_int **clReleaseMemObject** (cl_mem *memobj*)

decrements the *memobj* reference count. **clReleaseMemObject** 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_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.

After the *memobj* reference count becomes zero and commands queued for execution on a command-queue(s) that use *memobj* have finished, the memory object is deleted.

The function

```
cl_int      clSetMemObjectDestructorCallback (cl_mem memobj,
                    void (CL_CALLBACK *pfn_notify)(cl_mem memobj,
                                                   void *user_data),
                    void *user_data)
```

registers a user callback function that will be called when the memory object is deleted and its resources freed. Each call to **clSetMemObjectDestructorCallback** registers the specified user callback function on a callback stack associated with *memobj*. The registered user callback functions are called in the reverse order in which they were registered. The user callback functions are called and then the memory object's resources are freed and the memory object is deleted. This provides a mechanism for the application (and libraries) using *memobj* to be notified when the memory referenced by *host_ptr*, specified when the memory object is created and used as the storage bits for the memory object, can be reused or freed.

memobj is a valid memory object.

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. *user_data* can be NULL.

clSetMemObjectDestructorCallback 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 *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.

NOTE: When the user callback function is called by the implementation, the contents of the memory region pointed to by *host_ptr* (if the memory object is created with

CL_MEM_USE_HOST_PTR) are undefined. The callback function is typically used by the application to either free or reuse the memory region pointed to by *host_ptr*.

The behavior of calling expensive system routines, OpenCL API calls to create contexts or command-queues, or blocking OpenCL operations from the following list below, in a callback is undefined.

> **clFinish**,
> **clWaitForEvents**,
> blocking calls to **clEnqueueReadBuffer**, **clEnqueueReadBufferRect**,
> **clEnqueueWriteBuffer**, **clEnqueueWriteBufferRect,**
> blocking calls to **clEnqueueReadImage** and **clEnqueueWriteImage,**
> blocking calls to **clEnqueueMapBuffer,**
> **clEnqueueMapImage**,
> blocking calls to **clBuildProgram**

If an application needs to wait for completion of a routine from the above list in a callback, please use the non-blocking form of the function, and assign a completion callback to it to do the remainder of your work. Note that when a callback (or other code) enqueues commands to a command-queue, the commands are not required to begin execution until the queue is flushed. In standard usage, blocking enqueue calls serve this role by implicitly flushing the queue. Since blocking calls are not permitted in callbacks, those callbacks that enqueue commands on a command queue should either call **clFlush** on the queue before returning or arrange for **clFlush** to be called later on another thread.

The user callback function may not call OpenCL APIs with the memory object for which the callback function is invoked and for such cases the behavior of OpenCL APIs is considered to be undefined.

5.4.2 Unmapping mapped memory objects

The function

> cl_int **clEnqueueUnmapMemObject** (cl_command_queue *command_queue*,
> cl_mem *memobj*,
> void **mapped_ptr*,
> cl_uint *num_events_in_wait_list*,
> const cl_event **event_wait_list*,
> cl_event **event*)

enqueues a command to unmap a previously mapped region of a memory object. Reads or writes from the host using the pointer returned by **clEnqueueMapBuffer** or **clEnqueueMapImage** are considered to be complete.

command_queue must be a valid command-queue.

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 **clEnqueueMapBuffer**, or **clEnqueueMapImage** for *memobj*.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before **clEnqueueUnmapMemObject** can be executed. If *event_wait_list* is NULL, then **clEnqueueUnmapMemObject** does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same.

event returns an event object that identifies this particular 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. **clEnqueueBarrier** can be used instead.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- 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 **clEnqueueMapBuffer**, or **clEnqueueMapImage** for *memobj*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or if *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- 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.
- CL_INVALID_CONTEXT if context associated with *command_queue* and *memobj* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.

clEnqueueMapBuffer, and **clEnqueueMapImage** increments the mapped count of the memory object. The initial mapped count value of the memory object is zero. Multiple calls to

clEnqueueMapBuffer, or **clEnqueueMapImage** on the same memory object will increment this mapped count by appropriate number of calls. **clEnqueueUnmapMemObject** decrements the mapped count of the memory object.

clEnqueueMapBuffer, and **clEnqueueMapImage** act as synchronization points for a region of the buffer object being mapped.

5.4.2.1 Behavior of OpenCL commands that access mapped regions of a memory object

The contents of the regions of a memory object mapped for writing (i.e. CL_MAP_WRITE is set in *map_flags* argument to **clEnqueueMapBuffer**, or **clEnqueueMapImage**) are considered to be undefined until this region is unmapped. Reads and writes by a kernel executing on a device to a memory region(s) mapped for writing are undefined.

Multiple command-queues can map a region or overlapping regions of a memory object for reading (i.e. *map_flags* = CL_MAP_READ). The contents of the regions of a memory object mapped for reading can also be read by kernels executing on a device(s). The behavior of writes by a kernel executing on a device to a mapped region of a memory object is undefined. Mapping (and unmapping) overlapped regions of a memory object for writing is undefined.

The behavior of OpenCL function calls that enqueue commands that write or copy to regions of a memory object that are mapped is undefined.

5.4.3 Memory Object Queries

To get information that is common to all memory objects (buffer and image objects), use the following function

cl_int **clGetMemObjectInfo** (cl_mem *memobj*,
cl_mem_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

memobj specifies the memory object being queried.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetMemObjectInfo** is described in *table 5.9*.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.9*.

param_value_size_ret returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is NULL, it is ignored.

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.9* and *param_value* is not NULL.

- CL_INVALID_MEM_OBJECT if *memobj* is a not a valid memory 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.

cl_mem_info	**Return type**	**Info. returned in *param_value***
CL_MEM_TYPE	cl_mem_object_type	Returns one of the following values: CL_MEM_OBJECT_BUFFER if *memobj* is created with **clCreateBuffer** or **clCreateSubBuffer**. CL_MEM_OBJECT_IMAGE2D if *memobj* is a 2D image object. CL_MEM_OBJECT_IMAGE3D if *memobj* is a 3D image object.
CL_MEM_FLAGS	cl_mem_flags	Return the *flags* argument value specified when *memobj* is created with **clCreateBuffer**, **clCreateSubBuffer**, or **clCreateImage{2D|3D}**.
CL_MEM_SIZE	size_t	Return actual size of the data store associated with *memobj* in bytes.
CL_MEM_HOST_PTR	void *	Return the *host_ptr* argument value specified when *memobj* is created.
CL_MEM_MAP_COUNT[11]	cl_uint	Map count.

---

[11] The map count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for debugging.

CL_MEM_REFERENCE_COUNT[12]	cl_uint	Return *memobj* reference count.
CL_MEM_CONTEXT	cl_context	Return context specified when memory object is created.
CL_MEM_ASSOCIATED_ MEMOBJECT	cl_mem	Return memory object from which *memobj* is created. This returns the memory object specified as *buffer* argument to **clCreateSubBuffer**. Otherwise a NULL value is returned.
CL_MEM_OFFSET	size_t	Return offset if *memobj* is a sub-buffer object created using **clCreateSubBuffer**. This return 0 if *memobj* is not a sub-buffer object.

Table 5.9 *List of supported param_names by clGetMemObjectInfo*

---

[12] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

5.5 Sampler Objects

A sampler object describes how to sample an image when the image is read in the kernel. The built-in functions to read from an image in a kernel take a sampler as an argument. The sampler arguments to the image read function can be sampler objects created using OpenCL functions and passed as argument values to the kernel or can be samplers declared inside a kernel. In this section we discuss how sampler objects are created using OpenCL functions.

5.5.1 Creating Sampler Objects

The function

```
cl_sampler    clCreateSampler (cl_context context,
                               cl_bool normalized_coords,
                               cl_addressing_mode addressing_mode,
                               cl_filter_mode filter_mode,
                               cl_int *errcode_ret)
```

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

context must be a valid OpenCL context.

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

addressing_mode specifies how out-of-range image coordinates are handled when reading from an image. 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.

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

clCreateSampler returns a valid non-zero sampler object and *errcode_ret* is set to CL_SUCCESS if the sampler object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- 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 function

cl_int **clRetainSampler** (cl_sampler *sampler*)

increments the *sampler* reference count. **clCreateSampler** performs an implicit retain. **clRetainSampler** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_SAMPLER if *sampler* is not a valid sampler 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 function

cl_int **clReleaseSampler** (cl_sampler *sampler*)

decrements the *sampler* reference count. The sampler object is deleted after the reference count becomes zero and commands queued for execution on a command-queue(s) that use *sampler* have finished. **clReleaseSampler** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_SAMPLER if *sampler* is not a valid sampler 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.

5.5.2 Sampler Object Queries

The function

```
cl_int      clGetSamplerInfo (cl_sampler sampler,
                              cl_sampler_info param_name,
                              size_t param_value_size,
                              void *param_value,
                              size_t *param_value_size_ret)
```

returns information about the sampler object.

sampler specifies the sampler being queried.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetSamplerInfo** is described in *table 5.10*.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.10.*

param_value_size_ret returns the actual size in bytes of data copied to *param_value*. If *param_value_size_ret* is NULL, it is ignored.

cl_sampler_info	Return Type	Info. returned in *param_value*
CL_SAMPLER_REFERENCE_COUNT[13]	cl_uint	Return the *sampler* reference count.
CL_SAMPLER_CONTEXT	cl_context	Return the context specified when the sampler is created.
CL_SAMPLER_NORMALIZED_COORDS	cl_bool	Return the normalized coords value associated with *sampler*.
CL_SAMPLER_ADDRESSING_MODE	cl_addressing_mode	Return the addressing mode value associated with *sampler*.
CL_SAMPLER_FILTER_MODE	cl_filter_mode	Return the filter mode value associated with *sampler*.

Table 5.10 *clGetSamplerInfo parameter queries.*

---

[13] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.10* and *param_value* is not NULL.
- CL_INVALID_SAMPLER if *sampler* is a not a valid sampler 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.

5.6 Program Objects

An OpenCL program consists of a set of kernels that are identified as functions declared with the `__kernel` qualifier in the program source. OpenCL programs may also contain auxiliary functions and constant data that can be used by `__kernel` functions. The program executable can be generated *online* or *offline* by the OpenCL compiler for the appropriate target device(s).

A program object encapsulates the following information:

- An associated context.
- A program source or binary.
- The latest successfully built program executable, the list of devices for which the program executable is built, the build options used and a build log.
- The number of kernel objects currently attached.

5.6.1 Creating Program Objects

The function

cl_program **clCreateProgramWithSource** (cl_context *context*,
cl_uint *count*,
const char ***strings*,
const size_t **lengths*,
cl_int **errcode_ret*)

creates a program object for a context, and loads the source code specified by the text strings in the *strings* array into the program object. The devices associated with the program object are the devices associated with *context*.

context must be a valid OpenCL context.

strings is an array of *count* pointers to optionally null-terminated character strings that make up the source code.

The *lengths* argument is an array with the number of chars in each string (the string length). If an element in *lengths* is zero, its accompanying string is null-terminated. If *lengths* is NULL, all strings in the *strings* argument are considered null-terminated. Any length value passed in that is greater than zero excludes the null terminator in its count.

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

clCreateProgramWithSource returns a valid non-zero program object and *errcode_ret* is set to

CL_SUCCESS if the program object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if *count* is zero or if *strings* or any entry in *strings* 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 function

cl_program **clCreateProgramWithBinary** (cl_context *context*,
cl_uint *num_devices*,
const cl_device_id **device_list*,
const size_t **lengths*,
const unsigned char ***binaries*,
cl_int **binary_status*,
cl_int **errcode_ret*)

creates a program object for a context, and loads the binary bits specified by *binary* into the program object.

context must be a valid OpenCL context.

device_list is a pointer to a list of devices that are in *context*. *device_list* must be a non-NULL value. The binaries are loaded for devices specified in this list.

num_devices is the number of devices listed in *device_list*.

The devices associated with the program object will be the list of devices specified by *device_list*. The list of devices specified by *device_list* must be devices associated with *context*.

lengths is an array of the size in bytes of the program binaries to be loaded for devices specified by *device_list*.

binaries is an array of pointers to program binaries to be loaded for devices specified by *device_list*. For each device given by *device_list*[i], the pointer to the program binary for that device is given by *binaries*[i] and the length of this corresponding binary is given by *lengths*[i]. *lengths*[i] cannot be zero and *binaries*[i] cannot be a NULL pointer.

The program binaries specified by *binaries* contain the bits that describe the program executable that will be run on the device(s) associated with *context*. The program binary can consist of either or both:

- Device-specific executable(s), and/or,
- Implementation-specific intermediate representation (IR) which will be converted to the device-specific executable.

binary_status returns whether the program binary for each device specified in *device_list* 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 *device_list[i]*; otherwise returns CL_INVALID_VALUE if *lengths[i]* is zero or if *binaries[i]* is a NULL value or 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.

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

clCreateProgramWithBinary returns a valid non-zero program object and *errcode_ret* is set to CL_SUCCESS if the program object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if *device_list* is NULL or *num_devices* is zero.
- CL_INVALID_DEVICE if OpenCL devices listed in *device_list* are not in the list of devices associated with *context*.
- CL_INVALID_VALUE if *lengths* or *binaries* are NULL or if any entry in *lengths*[i] is zero or *binaries*[i] is NULL.
- 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.

OpenCL allows applications to create a program object using the program source or binary and build appropriate program executables. This allows applications to determine whether they want to use the pre-built offline binary or load and compile the program source and use the executable compiled/linked online as the program executable. This can be very useful as it allows applications to load and build program executables online on its first instance for appropriate OpenCL devices in the system. These executables can now be queried and cached by the application. Future instances of the application launching will no longer need to compile and build the program executables. The cached executables can be read and loaded by the application, which can help significantly reduce the application initialization time.

The function

cl_int **clRetainProgram** (cl_program *program*)

increments the *program* reference count. **clCreateProgram** does an implicit retain. **clRetainProgram** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_PROGRAM if *program* is not a valid program 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 function

cl_int **clReleaseProgram** (cl_program *program*)

decrements the *program* reference count. The program object is deleted after all kernel objects associated with *program* have been deleted and the *program* reference count becomes zero. **clReleaseProgram** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_PROGRAM if *program* is not a valid program 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.

5.6.2 Building Program Executables

The function

cl_int **clBuildProgram** (cl_program *program*,
cl_uint *num_devices*,
const cl_device_id **device_list*,
const char **options*,
void *(*CL_CALLBACK **pfn_notify)(cl_program program,*
*void *user_data*),
void **user_data*)

builds (compiles & 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*. OpenCL allows program executables to be built using the source or the binary. **clBuildProgram** must be called for *program* created using either **clCreateProgramWithSource** or **clCreateProgramWithBinary** to build the program executable for one or more devices associated with *program*.

program is the program object.

device_list is a pointer to a list of devices associated with *program*. If *device_list* is a NULL value, the program executable is built for all devices associated with *program* for which a source or binary has been loaded. If *device_list* is a non-NULL value, the program executable is built for devices specified in this list for which a source or binary has been loaded.

num_devices is the number of devices listed in *device_list*.

options is a pointer to a null-terminated string of characters 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, **clBuildProgram** does not need to wait for the build to complete and can return immediately. If *pfn_notify* is NULL, **clBuildProgram** 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.

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

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

- CL_INVALID_PROGRAM if *program* is not a valid program object.

- CL_INVALID_VALUE if *device_list* is NULL and *num_devices* is greater than zero, or if *device_list* is not NULL and *num_devices* is zero.

- CL_INVALID_VALUE if *pfn_notify* is NULL but *user_data* is not NULL.

- CL_INVALID_DEVICE if OpenCL devices listed in *device_list* are not in the list of devices associated with *program*

- CL_INVALID_BINARY if *program* is created with **clCreateWithProgramBinary** and devices listed in *device_list* 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 *device_list* by a previous call to **clBuildProgram** for *program* has not completed.

- CL_COMPILER_NOT_AVAILABLE if *program* is created with **clCreateProgramWithSource** 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 **clBuildProgram** 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.

5.6.3 Build Options

The build options are categorized as pre-processor options, options for math intrinsics, options that control optimization and miscellaneous options. This specification defines a standard set of options that must be supported by an OpenCL compiler when building program executables online or offline. These may be extended by a set of vendor- or platform-specific options.

5.6.3.1 Preprocessor options

These options control the OpenCL preprocessor which is run on each program source before actual compilation.

-D *name*

Predefine *name* as a macro, with definition 1.

-D *name=definition*

The contents of *definition* are tokenized and processed as if they appeared during translation phase three in a `#define' directive. In particular, the definition will be truncated by embedded newline characters.

-D options are processed in the order they are given in the *options* argument to **clBuildProgram**.

-I *dir*

Add the directory *dir* to the list of directories to be searched for header files.

5.6.3.2 Math Intrinsics Options

These options control compiler behavior regarding floating-point arithmetic. These options trade off between speed and correctness.

-cl-single-precision-constant

Treat double precision floating-point constant as single precision constant.

-cl-denorms-are-zero

This option controls how single precision and double precision denormalized numbers are handled. If specified as a build option, the single precision denormalized numbers may be flushed to zero; double precision denormalized numbers may also be flushed to zero if the optional extension for double precision is supported. This is intended to be a performance hint and the OpenCL compiler can choose not to flush denorms to zero if the device supports single precision (or double precision) denormalized numbers.

This option is ignored for single precision numbers if the device does not support single precision denormalized numbers i.e. CL_FP_DENORM bit is not set in CL_DEVICE_SINGLE_FP_CONFIG.

This option is ignored for double precision numbers if the device does not support double precision or if it does support double precision but not double precision denormalized numbers i.e. CL_FP_DENORM bit is not set in CL_DEVICE_DOUBLE_FP_CONFIG.

This flag only applies for scalar and vector single precision floating-point variables and computations on these floating-point variables inside a program. It does not apply to reading from or writing to image objects.

5.6.3.3 Optimization Options

These options control various sorts of optimizations. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

-cl-opt-disable

This option disables all optimizations. The default is optimizations are enabled.

-cl-strict-aliasing

This option allows the compiler to assume the strictest aliasing rules.

The following options control compiler behavior regarding floating-point arithmetic. These options trade off between performance and correctness and must be specifically enabled. These options are not turned on by default since it can result in incorrect output for programs which depend on an exact implementation of IEEE 754 rules/specifications for math functions.

-cl-mad-enable

Allow `a * b + c` to be replaced by a `mad`. The `mad` computes `a * b + c` with reduced accuracy. For example, some OpenCL devices implement `mad` as truncate the result of `a * b` before adding it to `c`.

-cl-no-signed-zeros

Allow optimizations for floating-point arithmetic that ignore the signedness of zero. IEEE 754 arithmetic specifies the distinct behavior of `+0.0` and `-0.0` values, which then prohibits simplification of expressions such as `x+0.0` or `0.0*x` (even with -cl-finite-math only). This option implies that the sign of a zero result isn't significant.

-cl-unsafe-math-optimizations

Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid, (b) may violate IEEE 754 standard and (c) may violate the OpenCL numerical compliance requirements as defined in *section 7.4* for single-precision floating-point, *section 9.3.9* for double-precision floating-point, and edge case behavior in *section 7.5*. This option includes the -cl-no-signed-zeros and -cl-mad-enable options.

-cl-finite-math-only

Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or ±∞. This option may violate the OpenCL numerical compliance requirements defined in *section 7.4* for single-precision floating-point, *section 9.3.9* for double-precision floating-point, and edge case behavior in *section 7.5*.

-cl-fast-relaxed-math

Sets the optimization options -cl-finite-math-only and -cl-unsafe-math-optimizations. This allows optimizations for floating-point arithmetic that may violate the IEEE 754 standard and the OpenCL numerical compliance requirements defined in in *section 7.4* for single-precision floating-point, *section 9.3.9* for double-precision floating-point, and edge case behavior in *section 7.5*. This option causes the preprocessor macro `__FAST_RELAXED_MATH__` to be defined in the OpenCL program.

5.6.3.4 Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error. The following language-independent options do not enable specific warnings but control the kinds of diagnostics produced by the OpenCL compiler.

-w

Inhibit all warning messages.

-Werror

Make all warnings into errors.

5.6.3.5 Options Controlling the OpenCL C version

The following option controls the version of OpenCL C that the compiler accepts.

-cl-std=

Determine the OpenCL C language version to use. A value for this option must be provided. Valid values are:

CL1.1 – Support all OpenCL C programs that use the OpenCL C language features defined in *section 6* of the OpenCL 1.1 specification.

Calls to **clBuildProgram** with the -cl-std=CL1.1 option **will fail** to build the program executable for any devices with CL_DEVICE_OPENCL_C_VERSION = OpenCL C 1.0.

If the –cl-std build option is not specified, the CL_DEVICE_OPENCL_C_VERSION is used to select the version of OpenCL C to be used when building the program executable for each device.

5.6.4 Unloading the OpenCL Compiler

The function

cl_int **clUnloadCompiler** (void)

allows the implementation to release the resources allocated by the OpenCL compiler. This is a hint from the application and does not guarantee that the compiler will not be used in the future or that the compiler will actually be unloaded by the implementation. Calls to **clBuildProgram** after **clUnloadCompiler** will reload the compiler, if necessary, to build the appropriate program executable. This call currently always returns CL_SUCCESS.

5.6.5 Program Object Queries

The function

cl_int **clGetProgramInfo** (cl_program *program*,
cl_program_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

returns information about the program object.

program specifies the program object being queried.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetProgramInfo** is described in *table 5.11*.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.11.*

param_value_size_ret returns the actual size in bytes of data copied to *param_value*. If *param_value_size_ret* is NULL, it is ignored.

cl_program_info	Return Type	Info. returned in *param_value*
CL_PROGRAM_REFERENCE_ COUNT[14]	cl_uint	Return the *program* reference count.
CL_PROGRAM_CONTEXT	cl_context	Return the context specified when the program object is created
CL_PROGRAM_NUM_DEVICES	cl_uint	Return the number of devices associated with *program*.
CL_PROGRAM_DEVICES	cl_device_id[]	Return the list of devices associated

---

[14] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

		with the program object. This can be the devices associated with context on which the program object has been created or can be a subset of devices that are specified when a progam object is created using **clCreateProgramWithBinary**.
CL_PROGRAM_SOURCE	char[]	Return the program source code specified by **clCreateProgramWithSource**. The source string returned is a concatenation of all source strings specified to **clCreateProgramWithSource** with a null terminator. The concatenation strips any nulls in the original source strings. The actual number of characters that represents the program source code including the null terminator is returned in *param_value_size_ret*.
CL_PROGRAM_BINARY_SIZES	size_t[]	Returns an array that contains the size in bytes of the program binary for each device associated with *program*. The size of the array is the number of devices associated with *program*. If a binary is not available for a device(s), a size of zero is returned.
CL_PROGRAM_BINARIES	unsigned char *[]	Return the program binaries for all devices associated with *program*. For each device in *program*, the binary returned can be the binary specified for the device when *program* is created with **clCreateProgramWithBinary** or it can be the executable binary generated by **clBuildProgram**. If *program* is created with **clCreateProgramWithSource**, the binary returned is the binary generated by **clBuildProgram**. The bits returned can be an implementation-specific intermediate representation (a.k.a. IR) or device specific executable bits or both. The decision on which information is returned in the binary is up to the OpenCL implementation.

		param_value points to an array of `n` pointers allocated by the caller, where `n` is the number of devices associated with program. The buffer sizes needed to allocate the memory that these `n` pointers refer to can be queried using the CL_PROGRAM_BINARY_SIZES query as described in this table. Each entry in this array is used by the implementation as the location in memory where to copy the program binary for a specific device, if there is a binary available. To find out which device the program binary in the array refers to, use the CL_PROGRAM_DEVICES query to get the list of devices. There is a one-to-one correspondence between the array of `n` pointers returned by CL_PROGRAM_BINARIES and array of devices returned by CL_PROGRAM_DEVICES. If an entry value in the array is NULL, the implementation skips copying the program binary for the specific device identified by the array index.

Table 5.11 *clGetProgramInfo parameter queries.*

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.11* and *param_value* is not NULL.

- CL_INVALID_PROGRAM if *program* is a not a valid program 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 function

cl_int **clGetProgramBuildInfo** (cl_program *program*,
cl_device_id *device*,
cl_program_build_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

returns build information for each device in the program object.

program specifies the program object being queried.

device specifies the device for which build information is being queried. *device* must be a valid device associated with *program*.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetProgramBuildInfo** is described in *table 5.12*.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.12.*

param_value_size_ret returns the actual size in bytes of data copied to *param_value*. If *param_value_size_ret* is NULL, it is ignored.

cl_program_buid_info	**Return Type**	**Info. returned in *param_value***
CL_PROGRAM_BUILD_STATUS	cl_build_status	Returns the build status of *program* for a specific device as given by *device*. This can be one of the following: CL_BUILD_NONE. The build status returned if no build has been performed on the specified program object for *device*. CL_BUILD_ERROR. The build status returned if the last call to **clBuildProgram** on the specified program object for *device* generated an error. CL_BUILD_SUCCESS. The build

		status retrned if the last call to **clBuildProgram** on the specified program object for *device* was successful. CL_BUILD_IN_PROGRESS. The build status returned if the last call to **clBuildProgram** on the specified program object for *device* has not finished.
CL_PROGRAM_BUILD_OPTIONS	char[]	Return the build options specified by the *options* argument in **clBuildProgram** for *device*. If build status of *program* for *device* is CL_BUILD_NONE, an empty string is returned.
CL_PROGRAM_BUILD_LOG	char[]	Return the build log when **clBuildProgram** was called for *device*. If build status of *program* for *device* is CL_BUILD_NONE, an empty string is returned.

Table 5.12 *clGetProgramBuildInfo parameter queries.*

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

- CL_INVALID_DEVICE if *device* is not in the list of devices associated with *program*.
- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.12* and *param_value* is not NULL.
- CL_INVALID_PROGRAM if *program* is a not a valid program 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.

5.7 Kernel Objects

A kernel is a function declared in a program. A kernel is identified by the `__kernel` qualifier applied to any function in a program. A kernel object encapsulates the specific `__kernel` function declared in a program and the argument values to be used when executing this `__kernel` function.

5.7.1 Creating Kernel Objects

To create a kernel object, use the function

cl_kernel **clCreateKernel** (cl_program *program,*
const char **kernel_name*,
cl_int **errcode_ret*)

program is a program object with a successfully built executable.

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

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

clCreateKernel returns a valid non-zero kernel object and *errcode_ret* is set to CL_SUCCESS if the kernel object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- 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 *kernel_name* is not found in *program*.
- CL_INVALID_KERNEL_DEFINITION if the function definition for `__kernel` function given by *kernel_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 *kernel_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 function

cl_int **clCreateKernelsInProgram** (cl_program *program*,
cl_uint *num_kernels*,
cl_kernel **kernels*,
cl_uint **num_kernels_ret*)

creates kernel objects for all kernel functions in *program*. Kernel objects are not created for any `__kernel` functions in *program* that do not have the same function definition across all devices for which a program executable has been successfully built.

program is a program object with a successfully built executable.

num_kernels is the size of memory pointed to by *kernels* specified as the number of cl_kernel entries.

kernels is the buffer where the kernel objects for kernels in *program* will be returned. If *kernels* is NULL, it is ignored. If *kernels* is not NULL, *num_kernels* must be greater than or equal to the number of kernels in *program*.

num_kernels_ret is the number of kernels in *program*. If *num_kernels_ret* is NULL, it is ignored.

clCreateKernelsInProgram will return CL_SUCCESS if the kernel objects were successfully allocated. Otherwise, it returns one of the following errors:

- CL_INVALID_PROGRAM if *program* is not a valid program object.
- CL_INVALID_PROGRAM_EXECUTABLE if there is no successfully built executable for any device in *program*.
- CL_INVALID_VALUE if *kernels* is not NULL and *num_kernels* is less than the number of kernels 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.

Kernel objects can only be created once you have a program object with a valid program source or binary loaded into the program object and the program executable has been successfully built for one or more devices associated with program. No changes to the program executable are allowed while there are kernel objects associated with a program object. This means that calls to

clBuildProgram return CL_INVALID_OPERATION if there are kernel objects attached to a program object. The OpenCL context associated with *program* will be the context associated with *kernel*. The list of devices associated with *program* are the devices associated with *kernel*. Devices associated with a program object for which a valid program executable has been built can be used to execute kernels declared in the program object.

The function

cl_int **clRetainKernel** (cl_kernel *kernel*)

increments the *kernel* reference count. **clRetainKernel** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_KERNEL if *kernel* is not a valid kernel 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.

clCreateKernel or **clCreateKernelsInProgram** do an implicit retain.

The function

cl_int **clReleaseKernel** (cl_kernel *kernel*)

decrements the *kernel* reference count. **clReleaseKernel** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_KERNEL if *kernel* is not a valid kernel 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 kernel object is deleted once the number of instances that are retained to *kernel* become zero and the kernel object is no longer needed by any enqueued commands that use *kernel*.

5.7.2 Setting Kernel Arguments

To execute a kernel, the kernel arguments must be set.

The function

cl_int **clSetKernelArg** (cl_kernel *kernel*,
cl_uint *arg_index*,
size_t *arg_size*,
const void **arg_value*)

is used to set the argument value for a specific argument of a kernel.

kernel is a valid kernel object.

arg_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.

For example, consider the following kernel:

```
__kernel void
image_filter (int n, int m,
            __constant float *filter_weights,
            __read_only image2d_t src_image,
            __write_only image2d_t dst_image)
{
    ...
}
```

Argument index values for `image_filter` will be `0` for `n`, `1` for `m`, `2` for `filter_weights`, `3` for `src_image` and `4` for `dst_image`.

arg_value is a pointer to data that should be used as the argument value for argument specified by *arg_index*. The argument data pointed to by *arg_value* is copied and the *arg_value* pointer can therefore be reused by the application after **clSetKernelArg** returns. The argument value specified is the value used by all API calls that enqueue *kernel* (**clEnqueueNDRangeKernel** and **clEnqueueTask**) until the argument value is changed by a call to **clSetKernelArg** for *kernel*.

If the argument is a memory object (buffer or image), the *arg_value* entry will be a pointer to the appropriate buffer or image object. The memory object must be created with the context associated with the kernel object. A NULL value can also be specified if the argument is a buffer object in which case a NULL value will be used as the value for the argument declared as a pointer to *__global* or *__constant* memory in the kernel. If the argument is declared with the *__local* qualifier, the *arg_value* entry must be NULL. If the argument is of type *sampler_t*, the *arg_value* entry must be a pointer to the sampler object.

If the argument is declared to be a pointer of a built-in or user defined type with the __*global* or __*constant* qualifier, the memory object specified as argument value must be a buffer object (or NULL). If the argument is declared with the __*constant* qualifier, the size in bytes of the memory object cannot exceed CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE and the number of arguments declared with the __*constant* qualifier cannot exceed CL_DEVICE_MAX_CONSTANT_ARGS.

The memory object specified as argument value must be a 2D image object if the argument is declared to be of type *image2d_t*. The memory object specified as argument value must be a 3D image object if argument is declared to be of type *image3d_t*.

For all other kernel arguments, the *arg_value* entry must be a pointer to the actual data to be used as argument value.

arg_size specifies the size of the argument value. If the argument is a memory object, the size is the size of the buffer or image object type. For arguments declared with the __*local* qualifier, the size specified will be the size in bytes of the buffer that must be allocated for the __*local* argument. If the argument is of type *sampler_t*, the *arg_size* value must be equal to `sizeof(cl_sampler)`. For all other arguments, the size will be the size of argument type.

NOTE: A kernel object does not update the reference count for objects such as memory, sampler objects specified as argument values by **clSetKernelArg**, Users may not rely on a kernel object to retain objects specified as argument values to the kernel[15].

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

- CL_INVALID_KERNEL if *kernel* is not a valid kernel object.
- CL_INVALID_ARG_INDEX if *arg_index* is not a valid argument index.
- CL_INVALID_ARG_VALUE if *arg_value* specified is NULL for an argument that is not declared with the __*local* qualifier or vice-versa.
- CL_INVALID_MEM_OBJECT for an argument declared to be a memory object when the specified *arg_value* is not a valid memory object.
- CL_INVALID_SAMPLER for an argument declared to be of type *sampler_t* when the specified *arg_value* is not a valid sampler object.

---

[15] Implementations shall not allow cl_kernel objects to hold reference counts to cl_kernel arguments, because no mechanism is provided for the user to tell the kernel to release that ownership right. If the kernel holds ownership rights on kernel args, that would make it impossible for the user to tell with certainty when he may safely release user allocated resources associated with OpenCL objects such as the cl_mem backing store used with CL_MEM_USE_HOST_PTR.

- CL_INVALID_ARG_SIZE if *arg_size* does not match the size of the data type for an argument that is not a memory object or if the argument is a memory object and *arg_size* != `sizeof(cl_mem)` or if *arg_size* is zero and the argument is declared with the *__local* qualifier or if the argument is a sampler and *arg_size* != `sizeof(cl_sampler)`.

- 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.

5.7.3 Kernel Object Queries

The function

cl_int **clGetKernelInfo** (cl_kernel *kernel*,
cl_kernel_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

returns information about the kernel object.

kernel specifies the kernel object being queried.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetKernelInfo** is described in *table 5.13*.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.13*.

param_value_size_ret returns the actual size in bytes of data copied to *param_value*. If *param_value_size_ret* is NULL, it is ignored.

cl_kernel_info	**Return Type**	**Info. returned in *param_value***
CL_KERNEL_FUNCTION_NAME	char[]	Return the kernel function name.
CL_KERNEL_NUM_ARGS	cl_uint	Return the number of arguments to *kernel*.

CL_KERNEL_REFERENCE_COUNT[16]	cl_uint	Return the *kernel* reference count.
CL_KERNEL_CONTEXT	cl_context	Return the context associated with *kernel*.
CL_KERNEL_PROGRAM	cl_program	Return the program object associated with *kernel*.

Table 5.13 *clGetKernelInfo parameter queries.*

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.13* and *param_value* is not NULL.
- CL_INVALID_KERNEL if *kernel* is a not a valid kernel 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 function

cl_int **clGetKernelWorkGroupInfo** (cl_kernel *kernel*,
cl_device_id *device*,
cl_kernel_work_group_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

returns information about the kernel object that may be specific to a device.

kernel specifies the kernel object being queried.

device identifies a specific device in the list of devices associated with *kernel*. The list of devices is the list of devices in the OpenCL context that is associated with *kernel*. If the list of devices associated with *kernel* is a single device, *device* can be a NULL value.

---

[16] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetKernelWorkGroupInfo** is described in *table 5.14*.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.14*.

param_value_size_ret returns the actual size in bytes of data copied to *param_value*. If *param_value_size_ret* is NULL, it is ignored.

cl_kernel_work_group_info	**Return Type**	**Info. returned in *param_value***
CL_KERNEL_WORK_GROUP_SIZE	size_t	This provides a mechanism for the application to query the maximum work-group size that can be used to execute a kernel on a specific device given by *device*. The OpenCL implementation uses the resource requirements of the kernel (register usage etc.) to determine what this work-group size should be.
CL_KERNEL_COMPILE_ WORK_GROUP_SIZE	size_t[3]	Returns the work-group size specified by the `__attribute__((reqd_work_group_size(X, Y, Z)))` qualifier. Refer to *section 6.8.2*. If the work-group size is not specified using the above attribute qualifier (0, 0, 0) is returned.
CL_KERNEL_LOCAL_MEM_SIZE	cl_ulong	Returns the amount of local memory in bytes being used by a kernel. This includes local memory that may be needed by an implementation to execute the kernel, variables declared inside the kernel with the `__local` address qualifier and local memory to be allocated for arguments to the kernel declared as pointers with the `__local` address qualifier and whose size is specified with **clSetKernelArg**.

		If the local memory size, for any pointer argument to the kernel declared with the `__local` address qualifier, is not specified, its size is assumed to be 0.
CL_KERNEL_PREFERRED_ WORK_GROUP_SIZE_MULTIPLE	size_t	Returns the preferred multiple of work-group size for launch. This is a performance hint. Specifying a work-group size that is not a multiple of the value returned by this query as the value of the local work size argument to **clEnqueueNDRangeKernel** will not fail to enqueue the kernel for execution unless the work-group size specified is larger than the device maximum.
CL_KERNEL_PRIVATE_MEM_SIZE	cl_ulong	Returns the minimum amount of private memory, in bytes, used by each work-item in the kernel. This value may include any private memory needed by an implementation to execute the kernel, including that used by the language built-ins and variable declared inside the kernel with the `__private` qualifier.

Table 5.14 *clGetKernelWorkGroupInfo parameter queries.*

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

- CL_INVALID_DEVICE if *device* is not in the list of devices associated with *kernel* or if *device* is NULL but there is more than one device associated with *kernel*.

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.14* and *param_value* is not NULL.

- CL_INVALID_KERNEL if *kernel* is a not a valid kernel 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.

5.8 Executing Kernels

The function

```
cl_int    clEnqueueNDRangeKernel (cl_command_queue command_queue,
                                  cl_kernel kernel,
                                  cl_uint  work_dim,
                                  const size_t *global_work_offset,
                                  const size_t *global_work_size,
                                  const size_t *local_work_size,
                                  cl_uint num_events_in_wait_list,
                                  const cl_event *event_wait_list,
                                  cl_event *event)
```

enqueues a command to execute a kernel on a device.

command_queue is a valid command-queue. The kernel will be queued for execution on the device associated with *command_queue*.

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

work_dim is the number of dimensions used to specify the global work-items and work-items in the work-group. *work_dim* must be greater than zero and less than or equal to CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS.

global_work_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 *global_work_offset* is NULL, the global IDs start at offset (0, 0, … 0).

global_work_size[17] 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_work_size*[0] * … * *global_work_size*[*work_dim* – 1].

local_work_size 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

---

[17] The values specified in *global_work_size* + corresponding values specified in *global_work_offset* cannot exceed the range given by the `sizeof(size_t)` for the device on which the kernel execution will be enqueued. The `sizeof(size_t)` for a device can be determined using CL_DEVICE_ADDRESS_BITS in *table 4.3*. If, for example, CL_DEVICE_ADDRESS_BITS = 32, i.e. the device uses a 32-bit address space, size_t is a 32-bit unsigned integer and global_work_size values must be in the range 1 .. 2^32 - 1. Values outside this range return a CL_OUT_OF_RESOURCES error.

execute the kernel specified by *kernel*. The total number of work-items in a work-group is computed as *local_work_size*[0] * … * *local_work_size*[*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_work_size*[0], … *local_work_size*[*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_work_size* will be used to determine how to break the global work-items specified by *global_work_size* into appropriate work-group instances. If *local_work_size* is specified, the values specified in *global_work_size*[0], … *global_work_size*[*work_dim* - 1] must be evenly divisible by the corresponding values specified in *local_work_size*[0], … *local_work_size*[*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_work_size* must match the value specified by the reqd_work_group_size attribute qualifier.

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

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

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

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same.

event returns an event object that identifies this particular kernel execution instance. Event objects are unique and can be used to identify a particular kernel execution instance later on. If *event* is NULL, no event will be created for this kernel execution instance and therefore it will not be possible for the application to query or queue a wait for this particular kernel execution instance.

clEnqueueNDRangeKernel 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 *command_queue*.

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- CL_INVALID_KERNEL if *kernel* is not a valid kernel object.

- CL_INVALID_CONTEXT if context associated with *command_queue* and *kernel* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.

- CL_INVALID_KERNEL_ARGS if the kernel argument values have not been specified.

- CL_INVALID_WORK_DIMENSION if *work_dim* is not a valid value (i.e. a value between 1 and 3).

- CL_INVALID_GLOBAL_WORK_SIZE if *global_work_size* is NULL, or if any of the values specified in *global_work_size*[0], … *global_work_size*[*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_work_size* + the corresponding values in *global_work_offset* 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_work_size* is specified and number of work-items specified by *global_work_size* is not evenly divisible by size of work-group given by *local_work_size* 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_work_size* is specified and the total number of work-items in the work-group computed as *local_work_size*[0] * … *local_work_size*[*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_work_size* 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_work_size*[0], … *local_work_size*[*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_work_size* 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_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.

- 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 function

cl_int **clEnqueueTask** (cl_command_queue *command_queue*,
cl_kernel *kernel*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

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

command_queue is a valid command-queue. The kernel will be queued for execution on the device associated with *command_queue*.

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

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same.

event returns an event object that identifies this particular kernel execution instance. Event objects are unique and can be used to identify a particular kernel execution instance later on. If *event* is NULL, no event will be created for this kernel execution instance and therefore it will not be possible for the application to query or queue a wait for this particular kernel execution instance.

clEnqueueTask is equivalent to calling **clEnqueueNDRangeKernel** with *work_dim* = 1, *global_work_offset* = NULL, *global_work_size*[0] set to 1 and *local_work_size*[0] set to 1.

clEnqueueTask 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 *command_queue*.

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- CL_INVALID_KERNEL if *kernel* is not a valid kernel object.

- CL_INVALID_CONTEXT if context associated with *command_queue* and *kernel* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.

- CL_INVALID_KERNEL_ARGS if the kernel argument values have not been specified.

- CL_INVALID_WORK_GROUP_SIZE if a work-group size is specified for *kernel* using the `__attribute__((reqd_work_group_size(X, Y, Z)))` qualifier in program source and is not `(1, 1, 1)`.

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

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

- CL_OUT_OF_RESOURCES if there is a failure to queue the execution instance of *kernel* on the command-queue because of insufficient resources needed to execute the kernel.

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

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.

- 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 function

cl_int **clEnqueueNativeKernel** (cl_command_queue *command_queue*,
void (**user_func*)(void *)
void **args*,
size_t *cb_args*,
cl_uint *num_mem_objects*,
const cl_mem **mem_list*,
const void ***args_mem_loc*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

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

command_queue is a valid command-queue. 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 a pointer to the args list that *user_func* should be called with.

cb_args is the size in bytes of the args list that *args* points to.

The data pointed to by *args* and *cb_args* 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 **clEnqueueNativeKernel** returns, the memory region pointed to by *args* can be reused by the application.

num_mem_objects is the number of buffer objects that are passed in *args*.

mem_list is a list of valid buffer objects, if *num_mem_objects* > 0. The buffer object values specified in *mem_list* are memory object handles (`cl_mem` values) returned by **clCreateBuffer** or NULL.

args_mem_loc is a pointer to appropriate locations that *args* points to where memory object handles (`cl_mem` values) are stored. Before the user function is executed, the memory object handles are replaced by pointers to global memory.

event_wait_list, num_events_in_wait_list and event are as described in **clEnqueueNDRangeKernel**.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_VALUE if *user_func* is NULL.
- CL_INVALID_VALUE if *args* is a NULL value and *cb_args* > 0, or if *args* is a NULL value and *num_mem_objects* > 0.
- CL_INVALID_VALUE if *args* is not NULL and *cb_args* is 0.
- CL_INVALID_VALUE if *num_mem_objects* > 0 and *mem_list* or *args_mem_loc* are NULL.
- CL_INVALID_VALUE if *num_mem_objects* = 0 and *mem_list* or *args_mem_loc* are not NULL.
- CL_INVALID_OPERATION if the device associated with *command_queue* cannot execute the native kernel.
- CL_INVALID_MEM_OBJECT if one or more memory objects specified in *mem_list* are not valid or are not buffer objects.

- CL_OUT_OF_RESOURCES if there is a failure to queue the execution instance of *kernel* on the command-queue because of insufficient resources needed to execute the kernel.

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

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.

- 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.

5.9 Event Objects

Event objects can be used to refer to a kernel execution command (**clEnqueueNDRangeKernel, clEnqueueTask, clEnqueueNativeKernel**), read, write, map and copy commands on memory objects (**clEnqueue{Read|Write|Map}Buffer, clEnqueueUnmapMemObject**, **clEnqueue{Read|Write}BufferRect**, **clEnqueue{Read|Write|Map}Image**, **clEnqueueCopy{Buffer|Image}**, **clEnqueueCopyBufferRect**, **clEnqueueCopyBufferToImage**, **clEnqueueCopyImageToBuffer), clEnqueueMarker** (refer to *section 5.10*) or user events.

An event object can be used to track the execution status of a command. The API calls that enqueue commands to a command-queue create a new event object that is returned in the *event* argument. In case of an error enqueuing the command in the command-queue the event argument does not return an event object.

The execution status of an enqueued command at any given point in time can be CL_QUEUED (command has been enqueued in the command-queue), CL_SUBMITTED (enqueued command has been submitted by the host to the device associated with the command-queue), CL_RUNNING (device is currently executing this command), CL_COMPLETE (command has successfully completed) or the appropriate error code if the command was abnormally terminated (this may be caused by a bad memory access etc.). The error code returned by a terminated command is a negative integer value. A command is considered to be complete if its execution status is CL_COMPLETE or is a negative integer value.

If the execution of a command is terminated, the command-queue associated with this terminated command, and the associated context (and all other command-queues in this context) may no longer be available. The behavior of OpenCL API calls that use this context (and command-queues associated with this context) are now considered to be implementation-defined. The user registered callback function specified when context is created can be used to report appropriate error information.

The function

cl_event **clCreateUserEvent** (cl_context *context*, cl_int **errcode_ret*)

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.

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

clCreateUserEvent returns a valid non-zero event object and *errcode_ret* is set to CL_SUCCESS if the user event object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- 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 execution status of the user event object created is set to CL_SUBMITTED.

The function

cl_int **clSetUserEventStatus** (cl_event *event*, cl_int *execution_status*)

sets the execution status of a user event object.

event is a user event object created using **clCreateUserEvent**.

execution_status specifies the new execution status to be set and can be CL_COMPLETE or a negative integer value to indicate an error. A negative integer value causes all enqueued commands that wait on this user event to be terminated. **clSetUserEventStatus** can only be called once to change the execution status of *event*.

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

- CL_INVALID_EVENT if *event* is not a valid user event object.
- CL_INVALID_VALUE if the *execution_status* is not CL_COMPLETE or a negative integer value.
- CL_INVALID_OPERATION if the *execution_status* for *event* has already been changed by a previous call to **clSetUserEventStatus**.
- 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.

NOTE: Enqueued commands that specify user events in the *event_wait_list* argument of **clEnqueue***** commands must ensure that the status of these user events being waited on are set using **clSetUserEventStatus** before any OpenCL APIs that release OpenCL objects except for event objects are called; otherwise the behavior is undefined.

For example, the following code sequence will result in undefined behavior of **clReleaseMemObject**.

```
ev1 = clCreateUserEvent(ctx, NULL);
clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ...,
                                   1, &ev1, NULL);
clEnqueueWriteBuffer(cq, buf2, CL_FALSE,...);
clReleaseMemObject(buf2);
clSetUserEventStatus(ev1, CL_COMPLETE);
```

The following code sequence, however, works correctly.

```
ev1 = clCreateUserEvent(ctx, NULL);
clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ...,
                                   1, &ev1, NULL);
clEnqueueWriteBuffer(cq, buf2, CL_FALSE,...);
clSetUserEventStatus(ev1, CL_COMPLETE);
clReleaseMemObject(buf2);
```

The function

cl_int **clWaitForEvents** (cl_uint *num_events*, const cl_event **event_list*)

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

clWaitForEvents returns CL_SUCCESS if the execution status of all events in *event_list* is CL_COMPLETE. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *num_events* is zero or *event_list* is NULL.
- CL_INVALID_CONTEXT if events specified in *event_list* do not belong to the same context.
- CL_INVALID_EVENT if event objects specified in *event_list* are not valid event objects.
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the execution status of any of the events in *event_list* is a negative integer value.
- 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 function

cl_int **clGetEventInfo** (cl_event *event*,
cl_event_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

returns information about the event object.

event specifies the event object being queried.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetEventInfo** is described in *table 5.15*.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.15*.

param_value_size_ret returns the actual size in bytes of data copied to *param_value*. If *param_value_size_ret* is NULL, it is ignored.

cl_event_info	**Return Type**	**Info. returned in *param_value***
CL_EVENT_COMMAND_QUEUE	cl_command_queue	Return the command-queue associated with *event*. For user event objects, a NULL value is returned.
CL_EVENT_CONTEXT	cl_context	Return the context associated with *event*.
CL_EVENT_COMMAND_TYPE	cl_command_type	Return the command associated with *event*. Can be one of the following values: CL_COMMAND_NDRANGE_KERNEL CL_COMMAND_TASK CL_COMMAND_NATIVE_KERNEL CL_COMMAND_READ_BUFFER CL_COMMAND_WRITE_BUFFER CL_COMMAND_COPY_BUFFER CL_COMMAND_READ_IMAGE CL_COMMAND_WRITE_IMAGE CL_COMMAND_COPY_IMAGE CL_COMMAND_COPY_BUFFER_TO_IMAGE CL_COMMAND_COPY_IMAGE_TO_BUFFER CL_COMMAND_MAP_BUFFER CL_COMMAND_MAP_IMAGE CL_COMMAND_UNMAP_MEM_OBJECT CL_COMMAND_MARKER CL_COMMAND_ACQUIRE_GL_OBJECTS

		CL_COMMAND_RELEASE_GL_OBJECTS CL_COMMAND_READ_BUFFER_RECT CL_COMMAND_WRITE_BUFFER_RECT CL_COMMAND_COPY_BUFFER_RECT CL_COMMAND_USER
CL_EVENT_COMMAND_ EXECUTION_STATUS[18]	cl_int	Return the execution status of the command identified by *event*. Valid values are: CL_QUEUED (command has been enqueued in the command-queue), CL_SUBMITTED (enqueued command has been submitted by the host to the device associated with the command-queue), CL_RUNNING (device is currently executing this command), CL_COMPLETE (the command has completed), or Error code given by a negative integer value. (command was abnormally terminated – this may be caused by a bad memory access etc.). These error codes come from the same set of error codes that are returned from the platform or runtime API calls as return values or *errcode_ret* values.
CL_EVENT_REFERENCE_ COUNT[19]	cl_uint	Return the *event* reference count.

Table 5.15 *clGetEventInfo parameter queries.*

Using **clGetEventInfo** to determine if a command identified by *event* has finished execution (i.e. CL_EVENT_COMMAND_EXECUTION_STATUS returns CL_COMPLETE) is not a synchronization point. There are no guarantees that the memory objects being modified by command associated with *event* will be visible to other enqueued commands.

---

[18] The error code values are negative, and event state values are positive. The event state values are ordered from the largest value (CL_QUEUED) for the first or initial state to the smallest value (CL_COMPLETE or negative integer value) for the last or complete state. The value of CL_COMPLETE and CL_SUCCESS are the same.

[19] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.15* and *param_value* is not NULL.
- CL_INVALID_VALUE if information to query given in *param_name* cannot be queried for *event*.
- CL_INVALID_EVENT if *event* is a not a valid event 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 function

cl_int **clSetEventCallback** (cl_event *event*,
cl_int *command_exec_callback_type*,
void (CL_CALLBACK **pfn_event_notify*)(*cl_event event,*
cl_int event_command_exec_status,
*void *user_data*),
void **user_data*)

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*.

Each call to **clSetEventCallback** registers the specified user callback function on a callback stack associated with *event*. The order in which the registered user callback functions are called is undefined.

event is a valid event object.

command_exec_callback_type specifies the command execution status for which the callback is registered. The command execution callback values for which a callback can be registered are: CL_COMPLETE[20]. 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.

---

[20] The callback function registered for a *command_exec_callback_type* value of CL_COMPLETE will be called when the command has completed successfully or is abnormally terminated.

pfn_event_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.
- *event_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 *event_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.

All callbacks registered for an event object must be called. All enqueued callbacks shall be called before the event object is destroyed. Callbacks must return promptly. The behavior of calling expensive system routines, OpenCL API calls to create contexts or command-queues, or blocking OpenCL operations from the following list below, in a callback is undefined.

clFinish,
clWaitForEvents,
blocking calls to **clEnqueueReadBuffer**, **clEnqueueReadBufferRect**,
clEnqueueWriteBuffer, **clEnqueueWriteBufferRect,**
blocking calls to **clEnqueueReadImage** and **clEnqueueWriteImage,**
blocking calls to **clEnqueueMapBuffer** and **clEnqueueMapImage**,
blocking calls to **clBuildProgram**

If an application needs to wait for completion of a routine from the above list in a callback, please use the non-blocking form of the function, and assign a completion callback to it to do the remainder of your work. Note that when a callback (or other code) enqueues commands to a command-queue, the commands are not required to begin execution until the queue is flushed. In standard usage, blocking enqueue calls serve this role by implicitly flushing the queue. Since blocking calls are not permitted in callbacks, those callbacks that enqueue commands on a command queue should either call **clFlush** on the queue before returning or arrange for **clFlush** to be called later on another thread.

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

- CL_INVALID_EVENT if *event* is not a valid event object.

- CL_INVALID_VALUE if *pfn_event_notify* is NULL or if *command_exec_callback_type* is not CL_COMPLETE.

- 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 function

cl_int **clRetainEvent** (cl_event *event*)

increments the *event* reference count. The OpenCL commands that return an event perform an implicit retain.

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

- CL_INVALID_EVENT if *event* is not a valid event 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.

To release an event, use the following function

cl_int **clReleaseEvent** (cl_event *event*)

decrements the *event* reference count.

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

- CL_INVALID_EVENT if *event* is not a valid event 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 event object is deleted once the reference count becomes zero, the specific command identified by this event has completed (or terminated) and there are no commands in the command-queues of a context that require a wait for this event to complete.

5.10 Markers, Barriers and Waiting for Events

The function

cl_int **clEnqueueMarker** (cl_command_queue *command_queue*,
cl_event **event*)

enqueues a marker command to *command_queue*. The marker command is not completed until all commands enqueued before it have completed. The marker command returns an *event* which can be waited on, i.e. this event can be waited on to insure that all commands, which have been queued before the marker command, have been completed.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_VALUE if *event* is a NULL value.
- 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 function

cl_int **clEnqueueBarrier** (cl_command_queue *command_queue*)

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

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- 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 function

cl_int **clEnqueueWaitForEvents** (cl_command_queue *command_queue*,
cl_uint *num_events*,
const cl_event **event_list*)

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. *num_events* specifies the number of events given by *event_list*. Each event in *event_list* must be a valid event object returned by a previous call to **clEnqueueNDRangeKernel**, **clEnqueueTask**, **clEnqueueNativeKernel**, **clEnqueue{Read|Write|Map}{Buffer|Image}**, **clEnqueueUnmapMemObject**, **clEnqueue{Read|Write}BufferRect**, **clEnqueueCopy{Buffer|Image}**, **clEnqueueCopyBufferRect**, **clEnqueueCopyBufferToImage, clEnqueueCopyImageToBuffer** or **clEnqueueMarker**.

The events specified in *event_list* act as synchronization points. The context associated with events in *event_list* and *command_queue* must be the same.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue* and events in *event_list* are not the same.
- CL_INVALID_VALUE if *num_events* is zero or *event_list* is NULL.
- CL_INVALID_EVENT if event objects specified in *event_list* are not valid events.
- 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.

5.11 Out-of-order Execution of Kernels and Memory Object Commands

The OpenCL functions that are submitted to a command-queue are enqueued in the order the calls are made but can be configured to execute in-order or out-of-order. The *properties* argument in **clCreateCommandQueue** can be used to specify the execution order.

If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is not set, the commands enqueued to a command-queue execute in order. For example, if an application calls **clEnqueueNDRangeKernel** to execute kernel A followed by a **clEnqueueNDRangeKernel** to execute kernel B, the application can assume that kernel A finishes first and then kernel B is executed. If the memory objects output by kernel A are inputs to kernel B then kernel B will see the correct data in memory objects produced by execution of kernel A. If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is set, then there is no guarantee that kernel A will finish before kernel B starts execution.

Applications can configure the commands enqueued to a command-queue to execute out-of-order by setting the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of the command-queue. This can be specified when the command-queue is created. In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued. As there is no guarantee that kernels will be executed in order, i.e. based on when the **clEnqueueNDRangeKernel** calls are made within a command-queue, it is therefore possible that an earlier **clEnqueueNDRangeKernel** call to execute kernel A identified by event A may execute and/or finish later than a **clEnqueueNDRangeKernel** call to execute kernel B which was called by the application at a later point in time. To guarantee a specific order of execution of kernels, a wait on a particular event (in this case event A) can be used. The wait for event A can be specified in the *event_wait_list* argument to **clEnqueueNDRangeKernel** for kernel B.

In addition, a wait for events (**clEnqueueWaitForEvents**) or a barrier (**clEnqueueBarrier**) command can be enqueued to the command-queue. The wait for events command ensures that previously enqueued commands identified by the list of events to wait for have finished before the next batch of commands is executed. The barrier command ensures that all previously enqueued commands in a command-queue have finished execution before the next batch of commands is executed.

Similarly, commands to read, write, copy or map memory objects that are enqueued after **clEnqueueNDRangeKernel, clEnqueueTask** or **clEnqueueNativeKernel** commands are not guaranteed to wait for kernels scheduled for execution to have completed (if the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property is set). To ensure correct ordering of commands, the event object returned by **clEnqueueNDRangeKernel, clEnqueueTask** or **clEnqueueNativeKernel** can be used to enqueue a wait for event or a barrier command can be enqueued that must complete before reads or writes to the memory object(s) occur.

5.12 Profiling Operations on Memory Objects and Kernels

This section describes profiling of OpenCL functions that are enqueued as commands to a command-queue. The specific functions[21] being referred to are: **clEnqueue{Read|Write|Map}Buffer, clEnqueue{Read|Write}BufferRect, clEnqueue{Read|Write|Map}Image, clEnqueueUnmapMemObject**, **clEnqueueCopyBuffer, clEnqueueCopyBufferRect, clEnqueueCopyImage, clEnqueueCopyImageToBuffer, clEnqueueCopyBufferToImage, clEnqueueNDRangeKernel** , **clEnqueueTask** and **clEnqueueNativeKernel**. These enqueued commands are identified by unique event objects.

Event objects can be used to capture profiling information that measure execution time of a command. Profiling of OpenCL commands can be enabled either by using a command-queue created with CL_QUEUE_PROFILING_ENABLE flag set in *properties* argument to **clCreateCommandQueue**.

If profiling is enabled, the function

cl_int **clGetEventProfilingInfo** (cl_event *event*,
cl_profiling_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

returns profiling information for the command associated with event.

event specifies the event object.

param_name specifies the profiling data to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetEventProfilingInfo** is described in *table 5.16*.

param_value is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 5.16*.

param_value_size_ret returns the actual size in bytes of data copied to *param_value*. If *param_value_size_ret* is NULL, it is ignored.

---

[21] **clEnqueueAcquireGLObjects** and **clEnqueueReleaseGLObjects** defined in *section 9.7.6* are also included.

cl_profiling_info	**Return Type**	**Info. returned in *param_value***
CL_PROFILING_COMMAND_QUEUED	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* is enqueued in a command-queue by the host.
CL_PROFILING_COMMAND_SUBMIT	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* that has been enqueued is submitted by the host to the device associated with the command-queue.
CL_PROFILING_COMMAND_START	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* starts execution on the device.
CL_PROFILING_COMMAND_END	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* has finished execution on the device.

Table 5.16 *clGetEventProfilingInfo parameter queries.*

The unsigned 64-bit values returned can be used to measure the time in nano-seconds consumed by OpenCL commands.

OpenCL devices are required to correctly track time across changes in device frequency and power states. The CL_DEVICE_PROFILING_TIMER_RESOLUTION specifies the resolution of the timer i.e. the number of nanoseconds elapsed before the timer is incremented.

clGetEventProfilingInfo returns CL_SUCCESS if the function is executed successfully and the profiling information has been recorded. Otherwise, it returns one of the following errors:

- CL_PROFILING_INFO_NOT_AVAILABLE if the CL_QUEUE_PROFILING_ENABLE flag is not set for the command-queue, if the execution status of the command identified by *event* is not CL_COMPLETE or if *event* is a user event object.
- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.16* and *param_value* is not NULL.

- CL_INVALID_EVENT if *event* is a not a valid event 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.

5.13 Flush and Finish

The function

cl_int **clFlush** (cl_command_queue *command_queue*)

issues all previously queued OpenCL commands in *command_queue* to the device associated with *command_queue*. **clFlush** 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 **clFlush** returns.

clFlush returns CL_SUCCESS if the function call was executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- 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.

Any blocking commands queued in a command-queue and **clReleaseCommandQueue** perform an implicit flush of the command-queue. These blocking commands are **clEnqueueReadBuffer**, **clEnqueueReadBufferRect**, **clEnqueueReadImage**, with *blocking_read* set to CL_TRUE; **clEnqueueWriteBuffer**, **clEnqueueWriteBufferRect**, **clEnqueueWriteImage** with *blocking_write* set to CL_TRUE; **clEnqueueMapBuffer**, **clEnqueueMapImage** with *blocking_map* set to CL_TRUE; or **clWaitForEvents**.

To use event objects that refer to commands enqueued in a command-queue as event objects to wait on by commands enqueued in a different command-queue, the application must call a **clFlush** or any blocking commands that perform an implicit flush of the command-queue where the commands that refer to these event objects are enqueued.

The function

cl_int **clFinish** (cl_command_queue *command_queue*)

blocks until all previously queued OpenCL commands in *command_queue* are issued to the associated device and have completed. **clFinish** does not return until all queued commands in *command_queue* have been processed and completed. **clFinish** is also a synchronization point.

clFinish returns CL_SUCCESS if the function call was executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- 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.

6. The OpenCL C Programming Language

This section describes the OpenCL C programming language used to create kernels that are executed on OpenCL device(s). The OpenCL C programming language (also referred to as OpenCL C) is based on the ISO/IEC 9899:1999 C language specification (a.k.a. C99 specification) with specific extensions and restrictions. Please refer to the ISO/IEC 9899:1999 specification for a detailed description of the language grammar. This section describes modifications and restrictions to ISO/IEC 9899:1999 supported in OpenCL C.

6.1 Supported Data Types

The following data types are supported.

6.1.1 Built-in Scalar Data Types

Table 6.1 describes the list of built-in scalar data types.

Type	Description
bool[22]	A conditional data type which is either *true* or *false*. The value *true* expands to the integer constant 1 and the value *false* expands to the integer constant 0.
char	A signed two's complement 8-bit integer.
unsigned char, uchar	An unsigned 8-bit integer.
short	A signed two's complement 16-bit integer.
unsigned short, ushort	An unsigned 16-bit integer.
int	A signed two's complement 32-bit integer.
unsigned int, uint	An unsigned 32-bit integer.
long	A signed two's complement 64-bit integer.
unsigned long, ulong	An unsigned 64-bit integer.
float	A single precision float. The float data type must conform to the IEEE 754 single precision storage format.
half	A 16-bit float. The half data type must conform to the IEEE 754-2008 half precision storage format.
size_t	The unsigned integer type of the result of the **sizeof** operator. This is a 32-bit unsigned integer if CL_DEVICE_ADDRESS_BITS

---

[22] When any scalar value is converted to **bool**, the result is 0 if the value compares equal to 0; otherwise, the result is 1.

	defined in *table 4.3* is 32-bits and is a 64-bit unsigned integer if CL_DEVICE_ADDRESS_BITS is 64-bits.
ptrdiff_t	A signed integer type that is the result of subtracting two pointers. This is a 32-bit signed integer if CL_DEVICE_ADDRESS_BITS defined in *table 4.3* is 32-bits and is a 64-bit signed integer if CL_DEVICE_ADDRESS_BITS is 64-bits.
intptr_t	A signed integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer.
uintptr_t	An unsigned integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer.
void	The **void** type comprises an empty set of values; it is an incomplete type that cannot be completed.

Table 6.1 *Built-in Scalar Data Types*

Most built-in scalar data types are also declared as appropriate types in the OpenCL API (and header files) that can be used by an application. The following table describes the built-in scalar data type in the OpenCL C programming language and the corresponding data type available to the application:

Type in OpenCL Language	API type for application
bool	**n/a**
char	**cl_char**
unsigned char, **uchar**	**cl_uchar**
short	**cl_short**
unsigned short, **ushort**	**cl_ushort**
int	**cl_int**
unsigned int, **uint**	**cl_uint**
long	**cl_long**
unsigned long, **ulong**	**cl_ulong**
float	**cl_float**
half	**cl_half**
size_t	**n/a**
ptrdiff_t	**n/a**
intptr_t	**n/a**
uintptr_t	**n/a**
void	**void**

6.1.1.1 The half data type

The `half` data type must be IEEE 754-2008 compliant. `half` numbers have 1 sign bit, 5 exponent bits, and 10 mantissa bits. The interpretation of the sign, exponent and mantissa is analogous to IEEE 754 floating-point numbers. The exponent bias is 15. The `half` data type must represent finite and normal numbers, denormalized numbers, infinities and NaN. Denormalized numbers for the `half` data type which may be generated when converting a `float` to a `half` using vstore_half and converting a half to a float using vload_half cannot be flushed to zero. Conversions from `float` to `half` correctly round the mantissa to 11 bits of precision. Conversions from `half` to `float` are lossless; all `half` numbers are exactly representable as `float` values.

The `half` data type can only be used to declare a pointer to a buffer that contains half values. A few valid examples are given below:

```
void
bar (__global half *p)
{
    ….
}

__kernel void
foo (__global half *pg, __local half *pl)
{
    __global half *ptr;
    int    offset;

    ptr = pg + offset;
    bar(ptr);
}
```

Below are some examples that are not valid usage of the `half` type:

```
half  a;
half  a[100];

half  *p;
a = *p;    ← not allowed. must use vload_half function
```

Loads from a pointer to a `half` and stores to a pointer to a `half` can be performed using the **vload_half**, **vload_half*n*, vloada_halfn** and **vstore_half**, **vstore_half*n*, vstorea_halfn** functions respectively as described in *section 6.11.7*. The load functions read scalar or vector half values from memory and convert them to a scalar or vector float value. The store functions take a scalar or vector float value as input, convert it to a half scalar or vector value (with appropriate rounding mode) and write the half scalar or vector value to memory.

6.1.2 Built-in Vector Data Types[23]

The char, unsigned char, short, unsigned short, integer, unsigned integer, long, unsigned long, float vector data types are supported. The vector data type is defined with the type name i.e. char, uchar, short, ushort, int, uint, float, long, ulong followed by a literal value *n* that defines the number of elements in the vector. Supported values of *n* are 2, 3, 4, 8, and 16 for all vector data types.

Table 6.2 describes the list of built-in vector data types.

Type	Description
char*n*	A 8-bit signed two's complement integer vector.
uchar*n*	A 8-bit unsigned integer vector.
short*n*	A 16-bit signed two's complement integer vector.
ushort*n*	A 16-bit unsigned integer vector.
int*n*	A 32-bit signed two's complement integer vector.
uint*n*	A 32-bit unsigned integer vector.
long*n*	A 64-bit signed two's complement integer vector.
ulong*n*	A 64-bit unsigned integer vector.
float*n*	A float vector.

Table 6.2 *Built-in Vector Data Types*

The built-in vector data types are also declared as appropriate types in the OpenCL API (and header files) that can be used by an application. The following table describes the built-in vector data type in the OpenCL C programming language and the corresponding data type available to the application:

Type in OpenCL Language	API type for application
char*n*	**cl_char*n***
uchar*n*	**cl_uchar*n***
short*n*	**cl_short*n***
ushort*n*	**cl_ushort*n***
int*n*	**cl_int*n***
uint*n*	**cl_uint*n***
long*n*	**cl_long*n***
ulong*n*	**cl_ulong*n***
float*n*	**cl_float*n***

---

[23] *Built-in vector data types are supported by the OpenCL implementation even if the underlying compute device does not support any or all of the vector data types. These are to be converted by the device compiler to appropriate instructions that use underlying built-in types supported natively by the compute device. Refer to Appendix B for a description of the order of the components of a vector type in memory.*

6.1.3 Other Built-in Data Types

Table 6.3 describes the list of additional data types supported by OpenCL.

Type	Description
image2d_t	A 2D image. Refer to *section 6.11.13* for a detailed description of this type.
image3d_t	A 3D image. Refer to *section 6.11.13* for a detailed description of this type.
sampler_t	A sampler type. Refer to *section 6.11.13* for a detailed description of this type.
event_t	An event. This can be used to identify async copies from global to local memory and vice-versa. Refer to *section 6.11.10*.

Table 6.3 *Other Built-in Data Types*

6.1.4 Reserved Data Types

The data type names described in *table 6.4* are reserved and cannot be used by applications as user-defined type names. The vector data type names defined in *table 6.2*, but where *n* is any value other than 2, 3, 4, 8 and 16, are also reserved.

Type	Description
bool*n*	A boolean vector.
half*n*	A 16-bit float vector.
quad, quad*n*	A 128-bit floating-point number and vectors.
complex half, **complex half*n*** **imaginary half,** **imaginary half*n***	A complex 16-bit floating-point number, and complex and imaginary 16-bit floating-point vectors.
complex float, **complex float*n*** **imaginary float,** **imaginary float*n***	A complex single precision floating-point number, and complex and imaginary single precision floating-point vectors.
complex double, **complex double*n*,** **imaginary double,** **imaginary double*n***	A complex double precision floating-point number, and complex and imaginary double precision floating-point vectors.
complex quad, **complex quad*n*,**	A complex 128-bit floating-point number, and complex and imaginary 128-bit floating-point

imaginary quad, **imaginary quad*n***	vectors.
float*n*x*m*	An *n* x *m* matrix of single precision floating-point values stored in column-major order.
double*n*x*m*	An *n* x *m* matrix of double precision floating-point values stored in column-major order.
long double **long double*n***	A floating-point scalar and vector type with at least as much precision and range as a double and no more precision and range than a quad.
long long, long long*n*	A 128-bit signed integer scalar and vector.
unsigned long long, **ulong long, ulong long*n***	A 128-bit unsigned integer scalar and vector.

Table 6.4 *Reserverd Data Types*

The C99 derived types (arrays, structs, union, function, and pointers), constructed from the built-in data types described in *sections 6.1.1, 6.1.2 and 6.1.3*, are also supported.

The type qualifiers `const`, `restrict` and `volatile` as defined by the C99 specification are supported. These qualifiers cannot be used with `image2d_t` and `image3d_t` type. Types other than pointer types shall not use the `restrict` qualifier.

6.1.5 Alignment of Types

A data item declared to be a data type in memory is always aligned to the size of the data type in bytes. For example, a float4 variable will be aligned to a 16-byte boundary, a char2 variable will be aligned to a 2-byte boundary.

For 3-component vector data types, the size of the data type is 4 * `sizeof`(component). This means that a 3-component vector data type will be aligned to a 4 * `sizeof`(component) boundary. The **vload3** and **vstore3** built-in functions can be used to read and write, respectively, 3-component vector data types from an array of packed scalar data type.

A built-in data type that is not a power of two bytes in size must be aligned to the next larger power of two. This rule applies to built-in types only, not structs or unions.

The OpenCL compiler is responsible for aligning data items to the appropriate alignment as required by the data type. For arguments to a `__kernel` function declared to be a pointer to a data type, the OpenCL compiler can assume that the pointee is always appropriately aligned as required by the data type. The behavior of an unaligned load or store is undefined, except for the **vload*n***, **vload_half*n***, **vstore*n***, and **vstore_half*n*** functions defined in *section 6.11.7*. The vector load functions can read a vector from an address aligned to the element type of the vector. The vector store functions can write a vector to an address aligned to the element type of the vector.

6.1.6 Vector Literals

Vector literals can be used to create vectors from a set of scalars, or vectors. A vector literal is written as a parenthesized vector type followed by a parenthesized set of expressions. Vector literals may be used either in initialization statements or as constants in executable statements. The number of literal values specified must be one, i.e. referring to a scalar value, or must match the size of the vector type being created. If a scalar literal value is specified, the scalar literal value will be replicated to all the components of the vector type.

Examples:

```
float4  f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

uint4   u = (uint4)(1); ← u will be (1, 1, 1, 1).

float4  f = (float4)((float2)(1.0f, 2.0f),
                     (float2)(3.0f, 4.0f));

float4  f = (float4)(1.0f, 2.0f); ← error
```

6.1.7 Vector Components

The components of vector data types with 1 … 4 components can be addressed as `<vector_data_type>.xyzw`. Vector data types of type `char2, uchar2, short2, ushort2, int2, uint2, long2, ulong2,` and `float2` can access `.xy` elements. Vector data types of type `char3, uchar3, short3, ushort3, int3, uint3, long3, ulong3,` and `float3` can access `.xyz` elements. Vector data types of type `char4, uchar4, short4, ushort4, int4, uint4, long4, ulong4, float4` can access `.xyzw` elements.

Accessing components beyond those declared for the vector type is an error so, for example:

```
float2 pos;
pos.x = 1.0f;    // is legal
pos.z = 1.0f;    // is illegal

float3 pos;
pos.z = 1.0f;    // is legal
pos.w = 1.0f;    // is illegal
```

The component selection syntax allows multiple components to be selected by appending their names after the period (**.**).

```
float4 c;

c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;
c.xy = (float2)(3.0f, 4.0f);
c.xyz = (float3)(3.0f, 4.0f, 5.0f);
```

The component selection syntax also allows components to be permuted or replicated.

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

float4 swiz= pos.wzyx; // swiz = (4.0f, 3.0f, 2.0f, 1.0f)

float4 dup = pos.xxyy; // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

The component group notation can occur on the left hand side of an expression. To form an l-value, swizzling must be applied to an l-value of vector type, contain no duplicate components, and it results in an l-value of scalar or vector type, depending on number of components specified. Each component must be a supported scalar or vector type.

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

pos.xw = (float2)(5.0f, 6.0f);// pos = (5.0f, 2.0f, 3.0f, 6.0f)
pos.wx = (float2)(7.0f, 8.0f);// pos = (8.0f, 2.0f, 3.0f, 7.0f)
pos.xyz = (float3)(3.0f, 5.0f, 9.0f); // pos = (3.0f, 5.0f, 9.0f, 4.0f)
pos.xx = (float2)(3.0f, 4.0f);// illegal - 'x' used twice

// illegal - mismatch between float2 and float4
pos.xy = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

float4  a, b, c, d;
float16 x;
x = (float16)(a, b, c, d);
x = (float16)(a.xxxx, b.xyz, c.xyz, d.xyz, a.yzw);

// illegal – component a.xxxxxxx is not a valid vector type
x = (float16)(a.xxxxxxx, b.xyz, c.xyz, d.xyz);
```

Elements of vector data types can also be accessed using a numeric index to refer to the appropriate element in the vector. The numeric indices that can be used are given in the table below:

Vector Components	Numeric indices that can be used
2-component	0, 1
3-component	0, 1, 2
4-component	0, 1, 2, 3
8-component	0, 1, 2, 3, 4, 5, 6, 7
16-component	0, 1, 2, 3, 4, 5, 6, 7,

	`8, 9, a, A, b, B, c, C, d, D, e, E, f, F`

Table 6.5 *Numeric indices for built-in vector data types*

The numeric indices must be preceded by the letter `s` or `S`.

In the following example

```
float8          f;
```

`f.s0` refers to the 1st element of the `float8` variable `f` and `f.s7` refers to the 8th element of the `float8` variable `f`.

In the following example

```
float16         x;
```

`x.sa (or x.sA)` refers to the 11th element of the `float16` variable `x` and `x.sf (or x.sF)` refers to the 16th element of the `float16` variable `x`.

The numeric indices used to refer to an appropriate element in the vector cannot be intermixed with `.xyzw` notation used to access elements of a 1 .. 4 component vector.

For example

```
float4          f, a;

a = f.x12w;     // illegal use of numeric indices with .xyzw

a.xyzw = f.s0123;   // valid
```

Vector data types can use the `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes to get smaller vector types or to combine smaller vector types to a larger vector type. Multiple levels of `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes can be used until they refer to a scalar term.

The `.lo` suffix refers to the lower half of a given vector. The `.hi` suffix refers to the upper half of a given vector.

The `.odd` suffix refers to the odd elements of a vector. The `.even` suffix refers to the even elements of a vector.

Some examples to help illustrate this are given below:

```
float4     vf;

float2     low = vf.lo;     // returns vf.xy
float2     high = vf.hi;    // returns vf.zw

float2     even = vf.even;  // returns vf.xz
float2     odd = vf.odd;    // returns vf.yw
```

The suffixes `.lo` (or `.even`) and `.hi` (or `.odd`) for a 3-component vector type operate as if the 3-component vector type is a 4-component vector type with the value in the `w` component undefined.

Some examples are given below:

```
float8      vf;
float4      left = vf.odd;
float4      right = vf.even;
float2      high = vf.even.hi;
float2      low = vf.odd.lo;

// interleave L+R stereo stream
float4      left,  right;
float8      interleaved;
interleaved.even = left;
interleaved.odd = right;

// deinterleave
left = interleaved.even;
right = interleaved.odd;

// transpose a 4x4 matrix
void transpose( float4 m[4] )
{
      // read matrix into a float16 vector
      float16 x = (float16)( m[0], m[1], m[2], m[3] );
      float16 t;

      //transpose
      t.even = x.lo;
      t.odd = x.hi;
      x.even = t.lo;
      x.odd = t.hi;
      //write back
      m[0] = x.lo.lo;   // { m[0][0], m[1][0], m[2][0], m[3][0] }
      m[1] = x.lo.hi;   // { m[0][1], m[1][1], m[2][1], m[3][1] }
      m[2] = x.hi.lo;   // { m[0][2], m[1][2], m[2][2], m[3][2] }
      m[3] = x.hi.hi;   // { m[0][3], m[1][3], m[2][3], m[3][3] }
}

float3      vf = (float3)(1.0f, 2.0f, 3.0f);
float2      low = vf.lo;  // (1.0f, 2.0f);
float2      high = vf.hi; // (3.0f, undefined);
```

6.1.8 Keywords

The following names are reserved for use as keywords in OpenCL C and shall not be used otherwise.

- Names reserved as keywords by C99.
- OpenCL C data types defined in *tables 6.2*, *6.3* and *6.4*.
- Address space qualifiers: `__global, global, __local, local, __constant, constant, __private and private`.
- Function qualifiers: `__kernel` and `kernel`.
- Access qualifiers: `__read_only`, `read_only`, `__write_only`, `write_only`, `__read_write` and `read_write`.

6.2 Conversions and Type Casting

6.2.1 Implicit Conversions

Implicit conversions between scalar built-in types defined in *table 6.1* (except `void` and `half`[24]) are supported. When an implicit conversion is done, it is not just a re-interpretation of the expression's value but a conversion of that value to an equivalent value in the new type. For example, the integer value `5` will be converted to the floating-point value `5.0`.

Implicit conversions between built-in vector data types are disallowed. Implicit conversions for members of an array or structure are disallowed. For example, an array of `int` cannot be implicitly converted to an array of `float`.

Implicit conversions for pointer types follow the rules described in the C99 specification.

6.2.2 Explicit Casts

Standard typecasts for built-in scalar data types defined in *table 6.1* will perform appropriate conversion (except `void` and `half`[25]). In the example below:

```
float    f = 1.0f;
int      i = (int)f;
```

`f` stores `0x3F800000` and `i` stores `0x1` which is the floating-point value `1.0f` in `f` converted to an integer value.

Explicit casts between vector types are not legal. The examples below will generate a compilation error.

```
int4       i;
uint4      u = (uint4) i;  ← not allowed

float4     f;
int4       i = (int4) f;  ← not allowed

float4     f;
int8       i = (int8) f;  ← not allowed
```

Scalar to vector conversions may be performed by casting the scalar to the desired vector data type. Type casting will also perform appropriate arithmetic conversion. The round to zero rounding mode will be used for conversions to built-in integer vector types. The current

---

[24] Unless the **cl_khr_fp16** extension is supported.

[25] Unless the **cl_khr_fp16** extension is supported.

rounding mode will be used for conversions to floating-point vector types. When casting a `bool` to a vector integer data type, the vector components will be set to -1 (i.e. all bits set) if the `bool` value is *true* and 0 otherwise.

Below are some correct examples of explicit casts.

```
float  f  = 1.0f;
float4 va = (float4)f;

// va is a float4 vector with elements (f, f, f, f).

uchar  u = 0xFF;
float4 vb = (float4)u;

// vb is a float4 vector with elements((float)u, (float)u,
//                                     (float)u, (float)u).

float f = 2.0f;
int2  vc = (int2)f;

// vc is an int2 vector with elements ((int)f, (int)f).

uchar4     vtrue = (uchar4)true;

// vtrue is a uchar4 vector with elements (0xff, 0xff,
//                                         0xff, 0xff).
```

6.2.3 Explicit Conversions

Explicit conversions may be performed using the

```
convert_destType(sourceType)
```

suite of functions. These provide a full set of type conversions between supported types (see *section 6.1.1*) except for the following types: `bool`, `half`, `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t`, and `void`.

The number of elements in the source and destination vectors must match.

In the example below:

```
uchar4  u;
int4    c = convert_int4(u);
```

`convert_int4` converts a `uchar4` vector `u` to an `int4` vector `c`.

```
float   f;
```

```
int     i = convert_int(f);
```

`convert_int` converts a `float` scalar `f` to an `int` scalar `i`.

Explicit conversions from a type to the same type has no effect on the type or value of an expression.

The behavior of the conversion may be modified by one or two optional modifiers that specify saturation for out-of-range inputs and rounding behavior.

The full form of the scalar convert function is:

```
destType convert_destType<_sat><_roundingMode> (sourceType)
```

The full form of the vector convert function is:

```
destTypen convert_destTypen<_sat><_roundingMode> (sourceTypen)
```

6.2.3.1 Data Types

Conversions are available for the following scalar types: `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, and built-in vector types derived therefrom. The operand and result type must have the same number of elements. The operand and result type may be the same type.

6.2.3.2 Rounding Modes

Conversions to and from floating-point type shall conform to IEEE-754 rounding rules. Conversions involving a floating-point or integer source operand or destination type may have an optional rounding mode modifier. These are described in the table below:

Modifier	**Rounding Mode Description**
`_rte`	Round to nearest even
`_rtz`	Round toward zero
`_rtp`	Round toward positive infinity
`_rtn`	Round toward negative infinity
no modifier specified	Use the default rounding mode for this destination type, `_rtz` for conversion to integers or the current rounding mode for conversion to floating-point types.

Table 6.6 *Rounding Modes*

By default, conversions to integer type use the `_rtz` (round toward zero) rounding mode and conversions to floating-point type[26] use the current rounding mode. The only default floating-point rounding mode supported is round to nearest even i.e the current rounding mode will be `_rte` for floating-point types.

6.2.3.3 Out-of-Range Behavior and Saturated Conversions

When the conversion operand is either greater than the greatest representable destination value or less than the least representable destination value, it is said to be out-of-range. The result of out-of-range conversion is determined by the conversion rules specified by the C99 specification in *section 6.3*. When converting from a floating-point type to integer type, the behavior is implementation-defined.

Conversions to integer type may opt to convert using the optional saturated mode by appending the `_sat` modifier to the conversion function name. When in saturated mode, values that are outside the representable range shall clamp to the nearest representable value in the destination format. (NaN should be converted to 0).

Conversions to floating-point type shall conform to IEEE-754 rounding rules. The `_sat` modifier may not be used for conversions to floating-point formats.

6.2.3.4 Explicit Conversion Examples

Example 1:

```
short4  s;

// negative values clamped to 0
ushort4     u = convert_ushort4_sat( s );

// values > CHAR_MAX converted to CHAR_MAX
// values < CHAR_MIN converted to CHAR_MIN
char4  c = convert_char4_sat( s );
```

Example 2:

```
float4 f;

// values implementation defined for
// f > INT_MAX, f < INT_MIN or NaN
int4   i = convert_int4( f );
```

---

[26] For conversions to floating-point format, when a finite source value exceeds the maximum representable finite floating-point destination value, the rounding mode will affect whether the result is the maximum finite floating-point value or infinity of same sign as the source value, per IEEE-754 rules for rounding.

```
// values > INT_MAX clamp to INT_MAX, values < INT_MIN clamp
// to INT_MIN. NaN should produce 0.
// The _rtz rounding mode is used to produce the integer values.
int4   i2 = convert_int4_sat( f );

// similar to convert_int4, except that floating-point values
// are rounded to the nearest integer instead of truncated
int4   i3 = convert_int4_rte( f );

// similar to convert_int4_sat, except that floating-point values
// are rounded to the nearest integer instead of truncated
int4   i4 = convert_int4_sat_rte( f );
```

Example 3:

```
int4   i;

// convert ints to floats using the current rounding mode.
float4 f = convert_float4( i );

// convert ints to floats. integer values that cannot
// be exactly represented as floats should round up to the
// next representable float.
float4 f = convert_float4_rtp( i );
```

6.2.4 Reinterpreting Data As Another Type

It is frequently necessary to reinterpret bits in a data type as another data type in OpenCL. This is typically required when direct access to the bits in a floating-point type is needed, for example to mask off the sign bit or make use of the result of a vector relational operator (see *section 6.3.d*) on floating-point data[27]. Several methods to achieve this (non-) conversion are frequently practiced in C, including pointer aliasing, unions and memcpy. Of these, only memcpy is strictly correct in C99. Since OpenCL does not provide **memcpy**, other methods are needed.

6.2.4.1 Reinterpreting Types Using Unions

The OpenCL language extends the union to allow the program to access a member of a union object using a member of a different type. The relevant bytes of the representation of the object are treated as an object of the type used for the access. If the type used for access is larger than

---

[27] In addition, some other extensions to the C language designed to support particular vector ISA (e.g. AltiVec™, CELL Broadband Engine™ Architecture) use such conversions in conjunction with swizzle operators to achieve type unconversion. So as to support legacy code of this type, as_typen() allows conversions between vectors of the same size but different numbers of elements, even though the behavior of this sort of conversion is not likely to be portable except to other OpenCL implementations for the same hardware architecture. AltiVec™ is a trademark of Motorola Inc. Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.

the representation of the object, then the value of the additional bytes is undefined.

Examples:

```
union{ float f; uint u; double d[28];} u;

u.u = 1;          // u.f contains 2**-149.  u.d is undefined --
                  // depending on endianness the low or high half
                  // of d is unknown

u.f = 1.0f;       // u.u contains 0x3f800000, u.d contains an
                  // undefined value -- depending on endianness
                  // the low or high half of d is unknown

u.d = 1.0;        // u.u contains 0x3ff00000 (big endian) or 0
                  // (little endian). u.f contains either 0x1.ep0f
                  // (big endian) or 0.0f (little endian)
```

6.2.4.2 Reinterpreting Types Using as_type() and as_type*n*()

All data types described in tables 6.1 and 6.2 (except `bool`, `half`[29] and `void`) may be also reinterpreted as another data type of the same size using the **as_type()** operator for scalar data types and the **as_type*n*()** operator[30] for vector data types. When the operand and result type contain the same number of elements, the bits in the operand shall be returned directly without modification as the new type. The usual type promotion for function arguments shall not be performed.

For example, `as_float(0x3f800000)` returns `1.0f`, which is the value that the bit pattern `0x3f800000` has if viewed as an IEEE-754 single precision value.

When the operand and result type contain a different number of elements, the result shall be implementation-defined except if the operand is a 4-component vector and the result is a 3-component vector. In this case, the bits in the operand shall be returned directly without modification as the new type. That is, a conforming implementation shall explicitly define a

---

[28] Only if the **cl_khr_fp64** extension is supported.

[29] Unless the **cl_khr_fp16** extension is supported.

[30] While the union is intended to reflect the organization of data in memory, the **as_type**() and **as_type*n***() constructs are intended to reflect the organization of data in register. The **as_type**() and **as_type*n***() constructs are intended to compile to no instructions on devices that use a shared register file designed to operate on both the operand and result types. Note that while differences in memory organization are expected to largely be limited to those arising from endianness, the register based representation may also differ due to size of the element in register. (For example, an architecture may load a char into a 32-bit register, or a char vector into a SIMD vector register with fixed 32-bit element size.) If the element count does not match, then the implementation should pick a data representation that most closely matches what would happen if an appropriate result type operator was applied to a register containing data of the source type. If the number of elements matches, then the as_typen() should faithfully reproduce the behavior expected from a similar data type reinterpretation using memory/unions. So, for example if an implementation stores all single precision data as double in register, it should implement as_int(float) by first downconverting the double to single precision and then (if necessary) moving the single precision bits to a register suitable for operating on integer data. If data stored in different address spaces do not have the same endianness, then the "dominant endianness" of the device should prevail.

behavior, but two conforming implementations need not have the same behavior when the number of elements in the result and operand types does not match. The implementation may define the result to contain all, some or none of the original bits in whatever order it chooses. It is an error to use **as_type()** or **as_type*n*()** operator to reinterpret data to a type of a different number of bytes.

Examples:

```
float f = 1.0f;
uint u = as_uint(f);  // Legal. Contains:  0x3f800000

float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
// Legal. Contains:
// (int4)(0x3f800000, 0x40000000, 0x40400000, 0x40800000)
int4 i = as_int4(f);

float4 f, g;
int4  is_less = f < g;

// Legal.  f[i] = f[i] < g[i] ? f[i] : 0.0f
f = as_float4(as_int4(f) & is_less);

int i;
// Legal. Result is implementation-defined.
short2 j = as_short2(i);

int4 i;
// Legal. Result is implementation-defined.
short8 j = as_short8(i);

float4 f;
// Error.  Result and operand have different sizes
double4 g = as_double4[31](f);

float4 f;
// Legal. g.xyz will have same values as f.xyz.  g.w is undefined
float3 g = as_float3(f);

float3 f;
// Error.  Result and operand have different sizes
float4 g = as_float4(f);
```

6.2.5 Pointer Casting

Pointers to old and new types may be cast back and forth to each other. Casting a pointer to a new type represents an unchecked assertion that the address is correctly aligned. The developer

---

[31] Only if the **cl_khr_fp64** extension is supported.

will also need to know the endianness of the OpenCL device and the endianness of the data to determine how the scalar and vector data elements are stored in memory.

6.2.6 Usual Arithmetic Conversions

Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a common real type for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. For this purpose, all vector types shall be considered to have higher conversion rank than scalars. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the usual arithmetic conversions. If the operands are of more than one vector type, then an error shall occur. Implicit conversions between vector types are not permitted, per *section 6.2.1*.

Otherwise, if there is only a single vector type, and all other operands are scalar types, the scalar types are converted to the type of the vector element, then widened into a new vector containing the same number of elements as the vector, by duplication of the scalar value across the width of the new vector. An error shall occur if any scalar operand has greater rank than the type of the vector element. For this purpose, the rank order defined as follows:

1. The rank of a floating point type is greater than the rank of another floating-point type, if the floating point type can exactly represent all numeric values in the second floating point type. (For this purpose, the encoding of the floating-point value is used, rather than the subset of the encoding usable by the device.)
2. The rank of any floating point type is greater than the rank of any integer type.
3. The rank of a integer type is greater than the rank of a integer type with less precision.
4. The rank of a unsigned integer type is **greater than** the rank of a signed integer type with the same precision.[32]
5. bool shall have a rank less than any other type.
6. The rank of an enumerated type shall equal the rank of the compatible integer type.
7. For all types, T1, T2 and T3, if T1 has greater rank than T2, and T2 has greater rank than T3, then T1 has greater rank than T3.

Otherwise, if all operands are scalar, the usual arithmetic conversions apply, per *section 6.3.1.8* of the C99 standard.

NOTE: Both the standard orderings in *sections 6.3.1.8* and *6.3.1.1* of C99 were examined and rejected. Had we used integer conversion rank here, `int4 + 0U` would have been legal and had `int4` return type. Had we used standard C99 usual arithmetic conversion rules for scalars, then the standard integer promotion would have been performed on vector integer element types and `short8 + char` would either have return type of `int8` or be illegal.

---

[32] This is different from the standard integer conversion rank described in C99 TC2, *section 6.3.1.1*.

6.3 Operators

a. The arithmetic operators add (+), subtract (-), multiply (*) and divide (/) operate on built-in integer and floating-point scalar, and vector data types. The remainder (%) operates on built-in integer scalar and integer vector data types. All arithmetic operators result in the same fundamental type (integer or floating-point) as the operand they operate on, after operand type conversion. After conversion, the following cases are valid:

   - The two operands are scalars. In this case, the operation is applied, resulting in a scalar.

   - One operand is a scalar, and the other is a vector. In this case, the scalar maybe subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

   - The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.

   All other cases of implicit conversions are illegal. Division on integer types which results in a value that lies outside of the range bounded by the maximum and minimum representable values of the integer type will not cause an exception but will result in an unspecified value. A divide by zero with integer types does not cause an exception but will result in an unspecified value. Division by zero for floating-point types will result in ±infinity or NaN as prescribed by the IEEE-754 standard. Use the built-in functions **dot** and **cross** to get, respectively, the vector dot product and the vector cross product.

b. The arithmetic unary operators (+ or −), operates on built-in scalar and vector types.

c. The arithmetic post- and pre-increment and decrement (**--** and **++**) operate on built-in scalar and vector types except the built-in scalar and vector float types[33]. All unary operators work component-wise on their operands. These result with the same type they operated on. For post- and pre-increment and decrement, the expression must be one that could be assigned to (an l-value). Pre-increment and pre-decrement add or subtract `1` to the contents of the expression they operate on, and the value of the pre-increment or pre-decrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract `1` to the contents of the expression they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.

---

[33] The pre- and post- increment operators may have unexpected behavior on floating-point values and are therefore not supported for floating-point scalar and vector built-in types. For example, 0x1.0p25f++ returns 0x1.0p25f. Also, (a++)-- is not guaranteed to return a, if a has fractional value. In non-default rounding modes, (a++)-- may produce the same result as a++ or a-- for large a.

d. The relational operators greater than (**>**), less than (**<**), greater than or equal (**>=**), and less than or equal (**<=**) operate on scalar and vector types. All relational operators result in an integer type. After operand type conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in an `int` scalar.
- One operand is a scalar, and the other is a vector. In this case, the scalar maybe subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.

All other cases of implicit conversions are illegal.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `charn` and `ucharn` return a `charn` result; vector source operands of type `shortn` and `ushortn` return a `shortn` result; vector source operands of type `intn`, `uintn` and `floatn` return an `intn` result; vector source operands of type `longn` and `ulongn` return a `longn` result. For scalar types, the relational operators shall return `0` if the specified relation is *false* and `1` if the specified relation is *true*. For vector types, the relational operators shall return `0` if the specified relation is *false* and `-1` (i.e. all bits set) if the specified relation is *true*. The relational operators always return `0` if either argument is not a number (NaN).

e. The equality operators equal **(==)**, and not equal (**!=**) operate on built-in scalar and vector types. All equality operators result in an integer type. After operand type conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in a scalar.
- One operand is a scalar, and the other is a vector. In this case, the scalar maybe subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.

All other cases of implicit conversions are illegal.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `char`*n* and `uchar`*n* return a `char`*n* result; vector source operands of type `short`*n* and `ushort`*n* return a `short`*n* result; vector source operands of type `int`*n*, `uint`*n* and `float`*n* return an `int`*n* result; vector source operands of type `long`*n* and `ulong`*n* return a `long`*n* result.

For scalar types, the equality operators return `0` if the specified relation is *false* and return `1` if the specified relation is *true*. For vector types, the equality operators shall return `0` if the specified relation is *false* and `-1` (i.e. all bits set) if the specified relation is *true*. The equality operator equal (==) returns `0` if one or both arguments are not a number (NaN). The equality operator not equal (!=) returns `1` (for scalar source operands) or `-1` (for vector source operands) if one or both arguments are not a number (NaN).

f. The bitwise operators and (**&**), or (**|**), exclusive or (**^**), not (**~**) operate on all scalar and vector built-in types except the built-in scalar and vector float types. For vector built-in types, the operators are applied component-wise. If one operand is a scalar and the other is a vector, the scalar maybe subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

g. The logical operators and (**&&**), or (**||**) operate on all scalar and vector built-in types except the built-in scalar and vector float types. For scalar built-in types only, and (**&&**) will only evaluate the right hand operand if the left hand operand compares unequal to 0. For scalar built-in types only, or (**||**) will only evaluate the right hand operand if the left hand operand compares equal to 0. For built-in vector types, both operands are evaluated and the operators are applied component-wise. If one operand is a scalar and the other is a vector, the scalar maybe subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

The logical operator exclusive or (**^^**) is reserved.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `char`*n* and `uchar`*n* return a `char`*n* result; vector source operands of type `short`*n* and `ushort`*n* return a `short`*n* result; vector source operands of type `int`*n*, and `uint`*n* return an `int`*n* result; vector source operands of type `long`*n* and `ulong`*n* return a `long`*n* result.

For scalar types, the logical operators shall return `0` if the result of the operation is *false* and `1` if the result is *true*. For vector types, the logical operators shall return `0` if the result of the operation is *false* and `-1` (i.e. all bits set) if the result is *true*.

h. The logical unary operator not (**!**) operates on all scalar and vector built-in types except the built-in scalar and vector float types. For built-in vector types, the operators are applied component-wise.

   The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `char`*`n`* and `uchar`*`n`* return a `char`*`n`* result; vector source operands of type `short`*`n`* and `ushort`*`n`* return a `short`*`n`* result; vector source operands of type `int`*`n`*, and `uint`*`n`* return an `int`*`n`* result; vector source operands of type `long`*`n`* and `ulong`*`n`* return a `long`*`n`* result.

   For scalar types, the result of the logical unary operator is `0` if the value of its operand compares unequal to `0`, and `1` if the value of its operand compares equal to `0`. For vector types, the unary operator shall return a `0` if the value of its operand compares unequal to `0`, and `-1` (i.e. all bits set) if the value of its operand compares equal to `0`.

i. The ternary selection operator (**?:**) operates on three expressions (*exp1* **?** *exp2* **:** *exp3*). This operator evaluates the first expression *exp1*, which can be a scalar or vector result except float. If the result is a scalar value then it selects to evaluate the second expression if the result compares unequal to 0, otherwise it selects to evaluate the third expression. If the result is a vector value, then this is equivalent to calling **select**(*exp3*, *exp2*, *exp1*). The **select** function is described in *table 6.14*. The second and third expressions can be any type, as long their types match, or there is a conversion in *section 6.2.1* **Implicit Conversions** that can be applied to one of the expressions to make their types match, or one is a vector and the other is a scalar and the scalar maybe subject to the usual arithmetic conversion to the element type used by the vector operand and widened to the same type as the vector type. This resulting matching type is the type of the entire expression.

j. The operators right-shift (**>>**), left-shift (**<<**) operate on all scalar and vector built-in types except the built-in scalar and vector float types. For built-in vector types, the operators are applied component-wise. For the right-shift (**>>**), left-shift (**<<**) operators, the rightmost operand must be a scalar if the first operand is a scalar, and the rightmost operand can be a vector or scalar if the first operand is a vector.

   The result of `E1 << E2` is `E1` left-shifted by $\log_2(N)$ least significant bits in `E2` viewed as an unsigned integer value, where `N` is the number of bits used to represent the data type of `E1` after integer promotion[34], if `E1` is a scalar, or the number of bits used to represent the type of `E1` elements, if `E1` is a vector. The vacated bits are filled with zeros.

   The result of `E1 >> E2` is `E1` right-shifted by $\log_2(N)$ least significant bits in `E2` viewed as an unsigned integer value, where `N` is the number of bits used to represent the data type of

---

[34] Integer promotion is described in ISO/IEC 9899:1999 in *section 6.3.1.1*.

`E1` after integer promotion, if `E1` is a scalar, or the number of bits used to represent the type of `E1` elements, if `E1` is a vector. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the vacated bits are filled with zeros. If `E1` has a signed type and a negative value, the vacated bits are filled with ones.

k. The `sizeof` operator yields the size (in bytes) of its operand, including any padding bytes (refer to *section 6.1.5*) needed for alignment, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array[35] type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

   When applied to an operand that has type `char`, `uchar`, the result is `1`. When applied to an operand that has type `short`, `ushort`, or `half` the result is `2`. When applied to an operand that has type `int`, `uint` or `float`, the result is `4`. When applied to an operand that has type `long`, `ulong` or `double`, the result is `8`. When applied to an operand that is a vector type, the result[36] is number of components * size of each scalar component. When applied to an operand that has array type, the result is the total number of bytes in the array. When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding. The `sizeof` operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member.

l. The comma (,) operator operates on expressions by returning the type and value of the right-most expression in a comma separated list of expressions. All expressions are evaluated, in order, from left to right.

m. The unary **(*)** operator denotes indirection. If the operand points to an object, the result is an lvalue designating the object. If the operand has type ‘‘pointer to *type*’’, the result has type ‘‘*type*’’. If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined[37].

n. The unary **(&)** operator returns the address of its operand. If the operand has type ‘‘*type*’’, the result has type ‘‘pointer to *type*’’. If the operand is the result of a unary * operator, neither that operator nor the **&** operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. Similarly, if the operand is the result of a **[]** operator, neither the **&** operator nor the unary * that is implied by the **[]** is evaluated and the result is as if the **&** operator were removed and

---

[35] Variable length arrays are not supported in OpenCL 1.1. Refer to *section 6.8.d*.

[36] Except for 3-component vectors whose size is defined as 4 * size of each scalar component.

[37] Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime. If ***P** is an lvalue and **T** is the name of an object pointer type, ***(T)P** is an lvalue that has a type compatible with that to which **T** points.

the **[]** operator were changed to a **+** operator. Otherwise, the result is a pointer to the object designated by its operand[38].

o. Assignments of values to variable names are done with the assignment operator (**=**), like

```
lvalue = expression
```

The assignment operator stores the value of *expression* into *lvalue*. The *expression* and *lvalue* must have the same type, or the expression must have a type in *table 6.1*, in which case an implicit conversion will be done on the expression before the assignment is done.

If *expression* is a scalar type and *lvalue* is a vector type, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

Any other desired type-conversions must be specified explicitly. L-values must be writable. Variables that are built-in types, entire structures or arrays, structure fields, l-values with the field selector (**.**) applied to select components or swizzles without repeated fields, l-values within parentheses, and l-values dereferenced with the array subscript operator (**[ ]**) are all l-values. Other binary or unary expressions, function names, swizzles with repeated fields, and constants cannot be l-values. The ternary operator (**?:**) is also not allowed as an l-value.

The order of evaluation of the operands is unspecified. If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined. Other assignment operators are the assignments add into (+=), subtract from (-=), multiply into (*=), divide into (/=), modulus into (**%**=), left shift by (<<=), right shift by (>>=), and into (&=), inclusive or into (|=), and exclusive or into (^=).

The expression

```
lvalue op= expression
```

is equivalent to

```
lvalue = lvalue op expression
```

and the l-value and expression must satisfy the requirements for both operator *op* and assignment (=).

Note: Except for the **`sizeof`** operator, the **`half`** data type cannot be used with any of the operators described in this section.

---

[38] Thus, **&*E** is equivalent to **E** (even if **E** is a null pointer), and **&(E1[E2])** to **((E1)+(E2))**. It is always true that if **E** is an lvalue that is a valid operand of the unary **&** operator, ***&E** is an lvalue equal to **E**.

6.4 Vector Operations

Vector operations are component-wise. Usually, when an operator operates on a vector, it is operating independently on each component of the vector, in a component-wise fashion.

For example,

```
float4     v, u;
float      f;

v = u + f;
```

will be equivalent to

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
v.w = u.w + f;
```

And

```
float4     v, u, w;

w = v + u;
```

will be equivalent to

```
w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;
w.w = v.w + u.w;
```

and likewise for most operators and all integer and floating-point vector types.

6.5 Address Space Qualifiers

OpenCL implements the following disjoint address spaces: **`__global, __local, __constant`** and **`__private`**. The address space qualifier may be used in variable declarations to specify the region of memory that is used to allocate the object. The C syntax for type qualifiers is extended in OpenCL to include an address space name as a valid type qualifier. If the type of an object is qualified by an address space name, the object is allocated in the specified address name; otherwise, the object is allocated in the generic address space.

The address space names without the __ prefix i.e. **`global`**, **`local`**, **`constant`** and **`private`** may be substituted for the corresponding address space names with the __prefix.

The generic address space name for arguments to a function in a program, or local variables of a function is `__private`. All function arguments shall be in the `__private` address space.

`__kernel` function arguments declared to be a pointer of a type can point to one of the following address spaces only: `__global`, `__local` or `__constant`. A pointer to address space A can only be assigned to a pointer to the same address space A. Casting a pointer to address space A to a pointer to address space B is illegal.

The `__kernel` function arguments declared to be of type `image2d_t`, or `image3d_t` always point to the `__global` address space.

There is no generic address space name for program scope variables. All program scope variables must be declared in the `__constant` address space.

Examples:

```
// declares a pointer p in the __private address space that
// points to an int object in address space __global
__global int *p;

// declares an array of 4 floats in the __private address space.
float x[4];
```

6.5.1 __global (or global)

The **`__global`** or **`global`** address space name is used to refer to memory objects (buffer or image objects) allocated from the global memory pool.

A buffer memory object can be declared as a pointer to a scalar, vector or user-defined struct. This allows the kernel to read and/or write any location in the buffer.

The actual size of the array memory object is determined when the memory object is allocated via appropriate API calls in the host code.

Some examples are:

```
__global float4   *color;   //  An array of float4 elements
typedef struct {
     float a[3];
     int   b[2];
} foo_t;
__global foo_t    *my_info; // An array of foo_t elements.
__global image2d_t texture; // A 2D texture image
```

If an image object is attached to an argument declared with this qualifier, the argument must be declared as type `image2d_t` for a 2D image object or as type `image3d_t` for a 3D image object. The elements of an image object cannot be directly accessed. Built-in functions to read from and write to an image object are provided.

The **`const`** qualifier can also be used with the **`__global`** qualifier to specify a read-only buffer memory object.

6.5.2 __local (or local)

The **`__local`** or **`local`** address space name is used to describe variables that need to be allocated in local memory and are shared by all work-items of a work-group. Pointers to the **`__local`** address space are allowed as arguments to functions (including `__kernel` functions). Variables declared in the **`__local`** address space inside a `__kernel` function must occur at `__kernel` function scope.

Some examples of variables allocated in the **`__local`** address space inside a `__kernel` function are:

```
__kernel void my_func(…)
{
     __local float   a;    // A single float allocated
                           // in local address space

     __local float   b[10];// An array of 10 floats
                            // allocated in local address space.

     if (…)
     {
         // example of variable in __local address space but not
         // declared at __kernel function scope.
         __local float     c;    ← not allowed.
     }
}
```

Variables allocated in the **`__local`** address space inside a `__kernel` function cannot be initialized.

```
__kernel void my_func(…)
{
        __local float    a = 1;      ← not allowed

        __local float    b;
        b = 1;                       ← allowed
}
```

NOTE: Variables allocated in the **`__local`** address space inside a `__kernel` function are allocated for each work-group executing the kernel and exist only for the lifetime of the work-group executing the kernel.

6.5.3 __constant (or constant)

The **`__constant`** or **`constant`** address space name is used to describe variables allocated in global memory and which are accessed inside a kernel(s) as read-only variables. These read-only variables can be accessed by all (global) work-items of the kernel during its execution. Pointers to the **`__constant`** address space are allowed as arguments to functions (including `__kernel` functions) and for variables declared inside functions. Variables allocated in the **`__constant`** address space can only be defined as program scope variables and are required to be initialized.

Writes to variables declared with the `__constant` address space qualifier in the OpenCL program source should be a compile-time error.

All string literal storage shall be in the `__constant` address space.

NOTE: Any argument to a kernel that is declared with the **__constant** address space qualifier counts as a separate constant argument. *Table 4.3* defines the relevant device capabilities CL_DEVICE_MAX_CONSTANT_ARGS and CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE.

Variables inside a function or in program scope can also be declared with the **__constant** address qualifier. Implementations are not required to aggregate these declarations into the fewest number of constant arguments. This behavior is implementation defined.

Thus portable code must conservatively assume that each variable declared inside a function or in program scope with the **__constant** qualifier counts as a separate constant argument.

6.5.4 __private (or private)

Variables inside a `__kernel` function not declared with an address space qualifier, all variables inside non-kernel functions, and all function arguments are in the **`__private`** or **`private`** address space. Variables declared as pointers are considered to point to the **`__private`** address space if an address space qualifier is not specified except for arguments declared to be of type `image2d_t`, and `image3d_t` implicitly point to the `__global` address space.

The `__global, __constant, __local, __private, global, constant, local and private` names are reserved for use as address space qualifiers and shall not be used otherwise.

6.6 Access Qualifiers

Image objects specified as arguments to a kernel can be declared to be read-only or write-only. A kernel cannot read from and write to the same image object. The **`__read_only (or read_only)`** and **`__write_only (or write_only)`** qualifiers must be used with image object arguments to declare if the image object is being read or written by a kernel. The default qualifier is **`__read_only`**.

In the following example

```
__kernel void
foo (read_only image2d_t imageA,
     write_only image2d_t imageB)
{
     ….
}
```

`imageA` is a read-only 2D image object, and `imageB` is a write-only 2D image object.

The `__read_only, __write_only, __read_write, read_only, write_only and read_write` names are reserved for use as access qualifiers and shall not be used otherwise.

6.7 Function Qualifiers

6.7.1 __kernel (or kernel)

The **`__kernel (or kernel)`** qualifier declares a function to be a kernel that can be executed by an application on an OpenCL device(s). The following rules apply to functions that are declared with this qualifier:

- It can be executed on the device only
- It can be called by the host
- It is just a regular function call if a `__kernel` function is called by another kernel function.

NOTE:

Kernel functions with variables declared inside the function with the `__local` or `local` qualifier can be called by the host using appropriate APIs such as **clEnqueueNDRangeKernel**, and **clEnqueueTask**.

The behavior of calling kernel functions with variables declared inside the function with the `__local` or `local` qualifier from other kernel functions is implementation-defined.

The `__kernel and kernel` names are reserved for use as functions qualifiers and shall not be used otherwise.

6.7.2 Optional Attribute Qualifiers

The `__kernel` qualifier can be used with the keyword `__attribute__` to declare additional information about the kernel function as described below.

The optional `__attribute__((vec_type_hint(<typen>)))`[39] is a hint to the compiler and is intended to be a representation of the computational *width* of the `__kernel`, and should serve as the basis for calculating processor bandwidth utilization when the compiler is looking to autovectorize the code. `vec_type_hint (<typen>)` shall be one of the built-in scalar or vector data type described in *tables 6.1 and 6.2*. If `vec_type_hint (<typen>)` is not specified, the default value is `int`.

---

[39] Implicit in autovectorization is the assumption that any libraries called from the __kernel must be recompilable at run time to handle cases where the compiler decides to merge or separate workitems. This probably means that such libraries can never be hard coded binaries or that hard coded binaries must be accompanied either by source or some retargetable intermediate representation. This may be a code security question for some.

The `__attribute__((vec_type_hint(int)))` is the default type.

For example, where the developer specified a width of `float4`, the compiler should assume that the computation usually uses up 4 lanes of a float vector, and would decide to merge work-items or possibly even separate one work-item into many threads to better match the hardware capabilities. A conforming implementation is not required to autovectorize code, but shall support the hint. A compiler may autovectorize, even if no hint is provided. If an implementation merges `N` work-items into one thread, it is responsible for correctly handling cases where the number of global or local work-items in any dimension modulo `N` is not zero.

Examples:

```
// autovectorize assuming float4 as the
// basic computation width
__kernel __attribute__((vec_type_hint(float4)))
void foo( __global float4 *p ) {  ....

// autovectorize assuming double as the
// basic computation width
__kernel __attribute__((vec_type_hint(double)))
void foo( __global float4 *p ){ ....

// autovectorize assuming int (default)
// as the basic computation width
__kernel
void foo( __global float4 *p ){ ....
```

If for example, a `__kernel` is declared with `__attribute__(( vec_type_hint (float4)))` (meaning that most operations in the `__kernel` are explicitly vectorized using `float4`) and the kernel is running using Intel® Advanced Vector Instructions (Intel® AVX) which implements a 8-float-wide vector unit, the autovectorizer might choose to merge two work-items to one thread, running a second work-item in the high half of the 256-bit AVX register.

As another example, a Power4 machine has two scalar double precision floating-point units with an 6-cycle deep pipe. An autovectorizer for the Power4 machine might choose to interleave six `__attribute__(( vec_type_hint (double2))) __kernels` into one hardware thread, to ensure that there is always 12-way parallelism available to saturate the FPUs. It might also choose to merge 4 or 8 work-items (or some other number) if it concludes that these are better choices, due to resource utilization concerns or some preference for divisibility by 2.

The optional `__attribute__((work_group_size_hint(X, Y, Z)))` is a hint to the compiler and is intended to specify the work-group size that may be used i.e. value most likely to be specified by the *local_work_size* argument to **clEnqueueNDRangeKernel**. For example the `__attribute__((work_group_size_hint(1, 1, 1)))` is a hint to the compiler that the kernel will most likely be executed with a work-group size of 1.

The optional `__attribute__((reqd_work_group_size(X, Y, Z)))` is the work-group size that must be used as the *local_work_size* argument to **clEnqueueNDRangeKernel**. This allows the compiler to optimize the generated code appropriately for this kernel. The optional `__attribute__((reqd_work_group_size(X, Y, Z)))`, if specified, must be (1, 1, 1) if the kernel is executed via **clEnqueueTask**.

If `Z` is one, the *work_dim* argument to **clEnqueueNDRangeKernel** can be 2 or 3. If `Y` and `Z` are one, the *work_dim* argument to **clEnqueueNDRangeKernel** can be 1, 2 or 3.

6.8 Restrictions[40]

a. The use of pointers is somewhat restricted. The following rules apply:

   - Arguments to `__kernel` functions declared in a program that are pointers must be declared with the `__global`, `__constant` or `__local` qualifier.

   - A pointer declared with the `__constant`, `__local` or `__global` qualifier can only be assigned to a pointer declared with the `__constant`, `__local` or `__global` qualifier respectively.

   - Pointers to functions are not allowed.

   - Arguments to `__kernel` functions in a program cannot be declared as a pointer to a pointer(s). Variables inside a function or arguments to non `__kernel` functions in a program can be declared as a pointer to a pointer(s).

b. Variables that are declared to be of type `image2d_t` or `image3d_t` refer to image memory objects. These can only be specified as arguments to a function. Elements of an image cannot be directly accessed. Specific built-in functions are provided to read from and write to any location in the image. Refer to *section 6.11.13* for a list of image read and write functions. The `image2d_t` or `image3d_t` data types cannot be declared in a struct. The `image2d_t` or `image3d_t` data types cannot be used to declare local variables or as the return type of a function. A `image2d_t`, or `image3d_t` argument to a function cannot be modified.

   Samplers cannot be declared as arrays, pointers, or be used as the type for local variables inside a function or as the return value of a function defined in a program. Samplers cannot be passed as arguments to functions called by a `__kernel` function. A sampler argument to a `__kernel` function cannot be modified.

c. Bit-fields are currently not supported.

d. Variable length arrays and structures with flexible (or unsized) arrays are not supported.

e. Variadic macros and functions are not supported.

f. The library functions defined in the C99 standard headers `assert.h`, `ctype.h`, `complex.h`, `errno.h`, `fenv.h`, `float.h`, `inttypes.h`, `limits.h`, `locale.h`, `setjmp.h`, `signal.h`, `stdarg.h`, `stdio.h`, `stdlib.h`, `string.h`, `tgmath.h`, `time.h`, `wchar.h` and `wctype.h` are not available and cannot be included by a program.

---

[40] Items struckthrough are restrictions in OpenCL 1.0 that are removed in OpenCL 1.1.

g. The `extern`, `static`, `auto` and `register` storage-class specifiers are not supported.

h. Predefined identifiers are not supported.

i. Recursion is not supported.

j. The function using the `__kernel` qualifier can only have return type `void` in the source code.

k. Arguments to `__kernel` functions in a program cannot be declared with the built-in scalar types `bool`, `half`, `size_t`, `ptrdiff_t`, `intptr_t`, and `uintptr_t`. The size in bytes of these types except `half` are implementation-defined and in addition can also be different for the OpenCL device and the host processor making it difficult to allocate buffer objects to be passed as arguments to a kernel declared as pointer to these types. `half` is not supported as `half` can be used as a storage format[41] only and is not a data type on which floating-point arithmetic can be performed.

l. Whether or not irreducible control flow is illegal is implementation defined.

~~m. Built-in types that are less than 32-bits in size i.e. `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort`, and `half` have the following restriction:~~

- ~~Writes to a pointer (or arrays) of type `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort`, and `half` or to elements of a struct that are of type `char`, `uchar`, `char2`, `uchar2`, `short` and `ushort` are not supported. Refer to *section 9.9* for additional information.~~

~~The kernel example below shows what memory operations are not supported on built-in types less than 32-bits in size.~~

```
~~kernel void
do_proc (__global char *pA, short b,
                     __global short *pB)
{
    char            x[100];
    __private char *px = x;
    int             id = (int)get_global_id(0);
    short           f;

    f = pB[id] + b; ← is allowed

    px[1] = pA[1];   ← error.  px cannot be written.~~
```

---

[41] Unless the **cl_khr_fp16** extension is supported.

~~`pB[id] = b; ← error. pB cannot be written`~~

~~`}`~~

n. Arguments to `__kernel` functions in a program cannot be declared to be of type `event_t`.

o. Elements of a struct or union must belong to the same address space. Declaring a struct or union whose elements are in different address spaces is illegal.

p. Arguments to `__kernel` functions that are declared to be a struct do not allow OpenCL objects to be passed as elements of the struct.

6.9 Preprocessor Directives and Macros

The preprocessing directives defined by the C99 specification are supported.

The # **pragma** directive is described as:

pragma *pp-tokens*$_{opt}$ *new-line*

A # **pragma** directive where the preprocessing token **OPENCL** (used instead of **STDC**) does not immediately follow **pragma** in the directive (prior to any macro replacement) causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such **pragma** that is not recognized by the implementation is ignored. If the preprocessing token **OPENCL** does immediately follow **pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms whose meanings are described elsewhere:

```
#pragma OPENCL FP_CONTRACT on-off-switch
     on-off-switch: one of ON OFF DEFAULT

#pragma OPENCL EXTENSION extensionname : behavior
#pragma OPENCL EXTENSION all : behavior
```

The following predefined macro names are available.

`__FILE__` The presumed name of the current source file (a character string literal).

`__LINE__` The presumed line number (within the current source file) of the current source line (an integer constant).

`__OPENCL_VERSION__` substitutes an integer reflecting the version number of the OpenCL supported by the OpenCL device. The version of OpenCL described in this document will have `__OPENCL_VERSION__` substitute the integer 110.

`CL_VERSION_1_0` substitutes the integer 100 reflecting the OpenCL 1.0 version.

`CL_VERSION_1_1` substitutes the integer 110 reflecting the OpenCL 1.1 version.

`__ENDIAN_LITTLE__` is used to determine if the OpenCL device is a little endian architecture or a big endian architecture (an integer constant of 1 if device is little endian and is undefined otherwise). Also refer to CL_DEVICE_ENDIAN_LITTLE specified in *table 4.3*.

`__kernel_exec(X, typen) (and kernel_exec(X, typen))` is defined as

```
__kernel __attribute__((work_group_size_hint(X, 1, 1))) \
         __attribute__((vec_type_hint(typen)))
```

`__IMAGE_SUPPORT__` is used to determine if the OpenCL device supports images. This is an integer constant of 1 if images are supported and is undefined otherwise. Also refer to CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3*.

`__FAST_RELAXED_MATH__` is used to determine if the –cl-fast-relaxed-math optimization option is specified in build options given to **clBuildProgram**. This is an integer constant of 1 if the –cl-fast-relaxed-math build option is specified and is undefined otherwise.

The macro names defined by the C99 specification but not currently supported by OpenCL are reserved for future use.

6.10 Attribute Qualifiers

This section describes the syntax with which `__attribute__` may be used, and the constructs to which attribute specifiers bind.

An attribute specifier is of the form `__attribute__ ((`*`attribute-list`*`)).`

An attribute list is defined as:

```
attribute-list:
     attribute_opt
     attribute-list , attribute_opt

attribute:
     attribute-token attribute-argument-clause_opt

attribute-token:
     identifier

attribute-argument-clause:
     ( attribute-argument-list )

attribute-argument-list:
     attribute-argument
     attribute-argument-list, attribute-argument

attribute-argument:
     assignment-expression
```

This syntax is taken directly from GCC but unlike GCC, which allows attributes to be applied only to functions, types, and variables, OpenCL attributes can be associated with:

- types;
- functions;
- variables;
- blocks; and
- control-flow statements.

In general, the rules for how an attribute binds, for a given context, are non-trivial and the reader is pointed to GCC's documentation and Maurer and Wong's paper [See 16. and 17. in *section 11* – **References**] for the details.

6.10.1 Specifying Attributes of Types

The keyword `__attribute__` allows you to specify special attributes of `enum`, `struct` and `union` types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Two attributes are currently defined for types: aligned, and packed.

You may specify type attributes in an `enum`, `struct` or `union` type declaration or definition, or for other types in a `typedef` declaration.

For an `enum`, `struct` or `union` type, you may specify attributes either between the `enum`, `struct` or `union` tag and the name of the type, or just past the closing curly brace of the *definition*. The former syntax is preferred.

`aligned (`*`alignment`*`)`

> This attribute specifies a minimum alignment (in bytes) for variables of the specified type. For example, the declarations:
>
> ```
> struct S { short f[3]; } __attribute__ ((aligned (8)));
> typedef int more_aligned_int __attribute__ ((aligned (8)));
> ```
>
> force the compiler to insure (as far as it can) that each variable whose type is `struct S` or `more_aligned_int` will be allocated and aligned *at least* on a 8-byte boundary.
>
> Note that the alignment of any given `struct` or `union` type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the `struct` or `union` in question and must also be a power of two. This means that you *can* effectively adjust the alignment of a `struct` or `union` type by attaching an aligned attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire `struct` or `union` type.
>
> As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given `struct` or `union` type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:
>
> ```
> struct S { short f[3]; } __attribute__ ((aligned));
> ```
>
> Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. In the example

above, the size of each `short` is 2 bytes, and therefore the size of the entire `struct S` type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire `struct S` type to 8 bytes.

Note that the effectiveness of aligned attributes may be limited by inherent limitations of the OpenCL device and compiler. For some devices, the OpenCL compiler may only be able to arrange for variables to be aligned up to a certain maximum alignment. If the OpenCL compiler is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your platform-specific documentation for further information.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

`packed`

This attribute, attached to `struct` or `union` type definition, specifies that each member of the structure or union is placed to minimize the memory required. When attached to an `enum` definition, it indicates that the smallest integral type should be used.

Specifying this attribute for `struct` and `union` types is equivalent to specifying the `packed` attribute on each of the structure or union members.

In the following example `struct my_packed_struct`'s members are packed closely together, but the internal layout of its `s` member is not packed. To do that, `struct my_unpacked_struct` would need to be packed, too.

```
struct my_unpacked_struct
{
   char c;
   int i;
};

struct __attribute__ ((packed)) my_packed_struct
{
   char c;
   int  i;
   struct my_unpacked_struct s;
};
```

You may only specify this attribute on the definition of a `enum`, `struct` or `union`, not on a `typedef` which does not also define the enumerated type, structure or union.

6.10.2 Specifying Attributes of Functions

Refer to *section 6.7* for the function attribute qualifiers currently supported.

6.10.3 Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. The following attribute qualifiers are currently defined:

`aligned (`*`alignment`*`)`

> This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:
>
> ```
> int x __attribute__ ((aligned (16))) = 0;
> ```
>
> causes the compiler to allocate the global variable `x` on a 16-byte boundary. The alignment value specified must be a power of two.
>
> You can also specify the alignment of structure fields. For example, to create double-word aligned `int` pair, you could write:
>
> ```
> struct foo { int x[2] __attribute__ ((aligned (8))); };
> ```
>
> This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.
>
> As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:
>
> ```
>     short array[3] __attribute__ ((aligned));
> ```
>
> Whenever you leave out the alignment factor in an `aligned` attribute specification, the OpenCL compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target device you are compiling for.
>
> When used on a `struct`, or struct member, the `aligned` attribute can only increase

the alignment; in order to decrease it, the `packed` attribute must be specified as well. When used as part of a `typedef`, the `aligned` attribute can both increase and decrease alignment, and specifying the `packed` attribute will generate a warning.

Note that the effectiveness of aligned attributes may be limited by inherent limitations of the OpenCL device and compiler. For some devices, the OpenCL compiler may only be able to arrange for variables to be aligned up to a certain maximum alignment. If the OpenCL compiler is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your platform-specific documentation for further information.

`packed`

The `packed` attribute specifies that a variable or structure field should have the smallest possible alignment—one byte for a variable, unless you specify a larger value with the `aligned` attribute.

Here is a structure in which the field `x` is packed, so that it immediately follows `a`:

```
struct foo
{
  char a;
  int x[2] __attribute__ ((packed));
};
```

An attribute list placed at the beginning of a user-defined type applies to the variable of that type and not the type, while attributes following the type body apply to the type.
For example:

```
/* a has alignment of 128 */
__attribute__((aligned(128))) struct A {int i;} a;

/* b has alignment of 16 */
__attribute__((aligned(16))) struct B {double d;}
                   __attribute__((aligned(32))) b ;

struct A a1; /* a1 has alignment of 4 */

struct B b1; /* b1 has alignment of 32 */
```

`endian (`*`endiantype`*`)`

The `endian` attribute determines the byte ordering of a variable. *`endiantype`* can be set to `host` indicating the variable uses the endianness of the host processor or can be set to `device` indicating the variable uses the endianness of the device on which the kernel will be executed. The default is `device`.

For example:

```
float4 *p  __attribute__ ((endian(host)));
```

specifies that data stored in memory pointed to by `p` will be in the host endian format.

6.10.4 Specifying Attributes of Blocks and Control-Flow-Statements

For basic blocks and control-flow-statements the attribute is placed before the structure in question, for example:

```
__attribute__((attr1)) {…}

for __attribute__((attr2)) (…) __attribute__((attr3)) {…}
```

Here `attr1` applies to the block in braces and `attr2` and `attr3` apply to the loop's control construct and body, respectively.

No attribute qualifiers for blocks and control-flow-statements are currently defined.

6.10.5 Extending Attribute Qualifiers

The attribute syntax can be extended for standard language extensions and vendor specific extensions. Any extensions should follow the naming conventions outlined in the introduction to *section 9*.

Attributes are intended as useful hints to the compiler. It is our intention that a particular implementation of OpenCL be free to ignore all attributes and the resulting executable binary will produce the same result. This does not preclude an implementation from making use of the additional information provided by attributes and performing optimizations or other transformations as it sees fit. In this case it is the programmer's responsibility to guarantee that the information provided is in some sense correct.

6.11 Built-in Functions

The OpenCL C programming language provides a rich set of built-in functions for scalar and vector operations. Many of these functions are similar to the function names provided in common C libraries but they support scalar and vector argument types. Applications should use the built-in functions wherever possible instead of writing their own version.

User defined OpenCL functions, behave per C standard rules for functions (C99, TC2, Section 6.9.1). On entry to the function, the size of each variably modified parameter is evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment rules described in *section 6.3.o*, including implicit scalar widening as necessary. Built-in functions described in this section behave similarly, except that in order to avoid ambiguity between multiple forms of the same built-in function, implicit scalar widening shall not occur. Note that some built-in functions described in this section do have forms that operate on mixed scalar and vector types, however.

6.11.1 Work-Item Functions

Table 6.7 describes the list of built-in work-item functions that can be used to query the number of dimensions, the global and local work size specified to **clEnqueueNDRangeKernel**, and the global and local identifier of each work-item when this kernel is being executed on a device. The number of dimensions, the global and local work size when executing a kernel using the function **clEnqueueTask** is one.

Function	**Description**
uint **get_work_dim** ()	Returns the number of dimensions in use. This is the value given to the *work_dim* argument specified in **clEnqueueNDRangeKernel**. For **clEnqueueTask,** this returns 1.
size_t **get_global_size** (uint *dimindx*)	Returns the number of global work-items specified for dimension identified by *dimindx*. This value is given by the *global_work_size* argument to **clEnqueueNDRangeKernel**. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values of *dimindx*, **get_global_size()** returns 1. For **clEnqueueTask**, this always returns 1.
size_t **get_global_id** (uint *dimindx*)	Returns the unique global work-item ID value for dimension identified by *dimindx*. The global work-item ID specifies the work-item ID based on the number of global work-items specified to execute the kernel. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For

	other values of *dimindx*, **get_global_id()** returns 0. For **clEnqueueTask**, this returns 0.
size_t **get_local_size** (uint *dimindx*)	Returns the number of local work-items specified in dimension identified by *dimindx*. This value is given by the *local_work_size* argument to **clEnqueueNDRangeKernel** if *local_work_size* is not NULL; otherwise the OpenCL implementation chooses an appropriate *local_work_size* value which is returned by this function. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values of *dimindx*, **get_local_size()** returns 1. For **clEnqueueTask**, this always returns 1.
size_t **get_local_id** (uint *dimindx*)	Returns the unique local work-item ID i.e. a work-item within a specific work-group for dimension identified by *dimindx*. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values of *dimindx*, **get_local_id()** returns 0. For **clEnqueueTask**, this returns 0.
size_t **get_num_groups** (uint *dimindx*)	Returns the number of work-groups that will execute a kernel for dimension identified by *dimindx*. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values of *dimindx*, **get_num_groups ()** returns 1. For **clEnqueueTask**, this always returns 1.
size_t **get_group_id** (uint *dimindx*)	**get_group_id** returns the work-group ID which is a number from 0 .. **get_num_groups**(*dimindx*) – 1. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values, **get_group_id()** returns 0. For **clEnqueueTask**, this returns 0.
size_t **get_global_offset** (uint *dimindx*)	**get_global_offset** returns the offset values specified in *global_work_offset* argument to **clEnqueueNDRangeKernel**. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values, **get_global_offset()** returns 0. For **clEnqueueTask**, this returns 0.

Table 6.7 *Work-Item Functions Table*

6.11.2 Math Functions

The list of built-in math functions is described in *table 6.8*. The built-in math functions are categorized into the following:

- A list of built-in functions that have scalar or vector argument versions, and,
- A list of built-in functions that only take scalar float arguments.

The vector versions of the math functions operate component-wise. The description is per-component.

The built-in math functions are not affected by the prevailing rounding mode in the calling environment, and always return the same value as they would if called with the round to nearest even rounding mode.

Table 6.8 describes the list of built-in math functions that can take scalar or vector arguments. We use the generic type name `gentype` to indicate that the function can take `float, float2, float3, float4, float8` or `float16` as the type for the arguments. For any specific use of a function, the actual type has to be the same for all arguments and the return type, unless otherwise specified.

Function	**Description**
gentype **acos** (gentype)	Arc cosine function.
gentype **acosh** (gentype)	Inverse hyperbolic cosine.
gentype **acospi** (gentype *x*)	Compute **acos** (*x*) / π.
gentype **asin** (gentype)	Arc sine function.
gentype **asinh** (gentype)	Inverse hyperbolic sine.
gentype **asinpi** (gentype *x*)	Compute **asin** (*x*) / π.
gentype **atan** (gentype *y_over_x*)	Arc tangent function.
gentype **atan2** (gentype *y*, gentype *x*)	Arc tangent of *y* / *x*.
gentype **atanh** (gentype)	Hyperbolic arc tangent.
gentype **atanpi** (gentype *x*)	Compute **atan** (*x*) / π.
gentype **atan2pi** (gentype *y*, gentype *x*)	Compute **atan2** (*y*, *x*) / π.
gentype **cbrt** (gentype)	Compute cube-root.
gentype **ceil** (gentype)	Round to integral value using the round to +ve infinity rounding mode.
gentype **copysign** (gentype *x*, gentype *y*)	Returns *x* with its sign changed to match the sign of *y*.
gentype **cos** (gentype)	Compute cosine.
gentype **cosh** (gentype)	Compute hyperbolic consine.
gentype **cospi** (gentype *x*)	Compute **cos** (π *x*).
gentype **erfc** (gentype)	Complementary error function.
gentype **erf** (gentype)	Error function encountered in integrating the

	normal distribution.
gentype **exp** (gentype x)	Compute the base- e exponential of x.
gentype **exp2** (gentype)	Exponential base 2 function.
gentype **exp10** (gentype)	Exponential base 10 function.
gentype **expm1** (gentype x)	Compute e^x- 1.0.
gentype **fabs** (gentype)	Compute absolute value of a floating-point number.
gentype **fdim** (gentype x, gentype y)	x - y if $x > y$, +0 if x is less than or equal to y.
gentype **floor** (gentype)	Round to integral value using the round to –ve infinity rounding mode.
gentype **fma** (gentype a, gentype b, gentype c)	Returns the correctly rounded floating-point representation of the sum of c with the infinitely precise product of a and b. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.
gentype **fmax** (gentype x, gentype y) gentype **fmax** (gentype x, float y)	Returns y if $x < y$, otherwise it returns x. If one argument is a NaN, **fmax()** returns the other argument. If both arguments are NaNs, **fmax()** returns a NaN.
gentype **fmin**[42] (gentype x, gentype y) gentype **fmin** (gentype x, float y)	Returns y if $y < x$, otherwise it returns x. If one argument is a NaN, **fmin()** returns the other argument. If both arguments are NaNs, **fmin()** returns a NaN.
gentype **fmod** (gentype x, gentype y)	Modulus. Returns $x - y$ * **trunc** (x/y).
gentype **fract** (gentype x, __global gentype *$iptr$)[43] gentype **fract** (gentype x, __local gentype *$iptr$) gentype **fract** (gentype x, __private gentype *$iptr$)	Returns **fmin**(x – **floor** (x), 0x1.fffffep-1f). **floor**(x) is returned in *iptr*.
gentype **frexp** (gentype x, __global int*n* *exp) gentype **frexp** (gentype x, __local int*n* *exp) gentype **frexp** (gentype x, __private int*n* *exp)	Extract mantissa and exponent from x. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of x equals mantissa returned * 2^{exp}.
gentype **hypot** (gentype x, gentype y)	Compute the value of the square root of $x^2 + y^2$ without undue overflow or underflow.
int*n* **ilogb** (gentype x)	Return the exponent as an integer value.
gentype **ldexp** (gentype x, int*n* n) gentype **ldexp** (gentype x, int n)	Multiply x by 2 to the power n.

---

[42] **fmin** and **fmax** behave as defined by C99 and may not match the IEEE 754-2008 definition for **minNum** and **maxNum** with regard to signaling NaNs. Specifically, signaling NaNs may behave as quiet NaNs.

[43] The min() operator is there to prevent **fract**(-small) from returning 1.0. It returns the largest positive floating-point number less than 1.0.

gentype **lgamma** (gentype *x*) gentype **lgamma_r** (gentype *x*, __global int*n* *signp) gentype **lgamma_r** (gentype *x*, __local int*n* *signp) gentype **lgamma_r** (gentype *x*, __private int*n* *signp)	Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the *signp* argument of **lgamma_r**.
gentype **log** (gentype)	Compute natural logarithm.
gentype **log2** (gentype)	Compute a base 2 logarithm.
gentype **log10** (gentype)	Compute a base 10 logarithm.
gentype **log1p** (gentype *x*)	Compute $\log_e(1.0 + x)$.
gentype **logb** (gentype *x*)	Compute the exponent of x, which is the integral part of $\log_r \mid x \mid$.
gentype **mad** (gentype *a*, gentype *b*, gentype *c*)	**mad** approximates $a * b + c$. Whether or how the product of $a * b$ is rounded and how supernormal or subnormal intermediate products are handled is not defined. **mad** is intended to be used where speed is preferred over accuracy[44].
gentype **maxmag** (gentype *x*, gentype *y*)	Returns x if $\mid x \mid > \mid y \mid$, y if $\mid y \mid > \mid x \mid$, otherwise **fmax**(*x*, *y*).
gentype **minmag** (gentype *x*, gentype *y*)	Returns x if $\mid x \mid < \mid y \mid$, y if $\mid y \mid < \mid x \mid$, otherwise **fmin**(*x*, *y*).
gentype **modf** (gentype *x*, __global gentype *iptr) gentype **modf** (gentype *x*, __local gentype *iptr) gentype **modf** (gentype *x*, __private gentype *iptr)	Decompose a floating-point number. The **modf** function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by *iptr*.
float*n* **nan** (uint*n* *nancode*)	Returns a quiet NaN. The *nancode* may be placed in the significand of the resulting NaN.
gentype **nextafter** (gentype *x*, gentype *y*)	Computes the next representable single-precision floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, **nextafter**() returns the largest representable floating-point number less than *x*.
gentype **pow** (gentype *x*, gentype *y*)	Compute *x* to the power *y*.
gentype **pown** (gentype *x*, int*n* *y*)	Compute *x* to the power *y*, where *y* is an integer.
gentype **powr** (gentype *x*, gentype *y*)	Compute *x* to the power *y*, where x is >= 0.
gentype **remainder** (gentype *x*, gentype *y*)	Compute the value *r* such that $r = x - n*y$, where *n* is the integer nearest the exact value of x/y. If there are two integers closest to x/y, *n* shall be the even one. If *r* is zero, it is given the same sign as *x*.
gentype **remquo** (gentype *x*,	The **remquo** function computes the value r such

---

[44] The user is cautioned that for some usages, e.g. **mad**(a, b, -a*b), the definition of **mad**() is loose enough that almost any result is allowed from **mad**() for some values of a and b.

gentype *y*, __global int*n* **quo*) gentype **remquo** (gentype *x*, gentype *y*, __local int*n* **quo*) gentype **remquo** (gentype *x*, gentype *y*, __private int*n* **quo*)	that *r* = *x* - *n***y*, where n is the integer nearest the exact value of *x*/*y*. If there are two integers closest to *x*/*y*, *n* shall be the even one. If *r* is zero, it is given the same sign as *x*. This is the same value that is returned by the **remainder** function. **remquo** also calculates the lower seven bits of the integral quotient *x*/*y*, and gives that value the same sign as *x*/*y*. It stores this signed value in the object pointed to by *quo*.
gentype **rint** (gentype)	Round to integral value (using round to nearest even rounding mode) in floating-point format. Refer to section 7.1 for description of rounding modes.
gentype **rootn** (gentype *x*, int*n y*)	Compute *x* to the power 1/*y*.
gentype **round** (gentype *x*)	Return the integral value nearest to *x* rounding halfway cases away from zero, regardless of the current rounding direction.
gentype **rsqrt** (gentype)	Compute inverse square root.
gentype **sin** (gentype)	Compute sine.
gentype **sincos** (gentype *x*, __global gentype **cosval*) gentype **sincos** (gentype *x*, __local gentype **cosval*) gentype **sincos** (gentype *x*, __private gentype **cosval*)	Compute sine and cosine of x. The computed sine is the return value and computed cosine is returned in *cosval*.
gentype **sinh** (gentype)	Compute hyperbolic sine.
gentype **sinpi** (gentype *x*)	Compute **sin** (π *x*).
gentype **sqrt** (gentype)	Compute square root.
gentype **tan** (gentype)	Compute tangent.
gentype **tanh** (gentype)	Compute hyperbolic tangent.
gentype **tanpi** (gentype *x*)	Compute **tan** (π *x*).
gentype **tgamma** (gentype)	Compute the gamma function.
gentype **trunc** (gentype)	Round to integral value using the round to zero rounding mode.

Table 6.8 *Scalar and Vector Argument Built-in Math Function Table*

Table 6.9 describes the following functions:

- A subset of functions from *table 6.8* that are defined with the `half_` prefix . These functions are implemented with a minimum of 10-bits of accuracy i.e. an ULP value <= 8192 ulp.

- A subset of functions from *table 6.8* that are defined with the `native_` prefix. These functions may map to one or more native device instructions and will typically have better performance compared to the corresponding functions (without the `native__` prefix) described in *table 6.8*. The accuracy (and in some cases the input range(s)) of these functions is implementation-defined.

- `half_` and `native_` functions for following basic operations: divide and reciprocal.

We use the generic type name `gentype` to indicate that the functions in *table 6.9* can take `float, float2, float3, float4, float8` or `float16` as the type for the arguments.

Function	**Description**
gentype **half_cos** (gentype *x*)	Compute cosine. *x* must be in the range -2^{16} … $+2^{16}$.
gentype **half_divide** (gentype *x*, gentype *y*)	Compute *x* / *y*.
gentype **half_exp** (gentype *x*)	Compute the base- e exponential of *x*.
gentype **half_exp2** (gentype *x*)	Compute the base- 2 exponential of *x*.
gentype **half_exp10** (gentype *x*)	Compute the base- 10 exponential of *x*.
gentype **half_log** (gentype *x*)	Compute natural logarithm.
gentype **half_log2** (gentype *x*)	Compute a base 2 logarithm.
gentype **half_log10** (gentype *x*)	Compute a base 10 logarithm.
gentype **half_powr** (gentype *x*, gentype *y*)	Compute *x* to the power *y*, where *x* is >= 0.
gentype **half_recip** (gentype *x*)	Compute reciprocal.
gentype **half_rsqrt** (gentype *x*)	Compute inverse square root.
gentype **half_sin** (gentype *x*)	Compute sine. *x* must be in the range -2^{16} … $+2^{16}$.
gentype **half_sqrt** (gentype *x*)	Compute square root.
gentype **half_tan** (gentype *x*)	Compute tangent. *x* must be in the range -2^{16} … $+2^{16}$.
gentype **native_cos** (gentype *x*)	Compute cosine over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_divide** (gentype *x*, gentype *y*)	Compute *x* / *y* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_exp** (gentype *x*)	Compute the base- e exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_exp2** (gentype *x*)	Compute the base- 2 exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_exp10** (gentype *x*)	Compute the base- 10 exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_log** (gentype *x*)	Compute natural logarithm over an implementation-defined range. The maximum error is implementation-

	defined.
gentype **native_log2** (gentype *x*)	Compute a base 2 logarithm over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_log10** (gentype *x*)	Compute a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_powr** (gentype *x*, gentype *y*)	Compute *x* to the power *y*, where *x* is >= 0. The range of *x* and *y* are implementation-defined. The maximum error is implementation-defined.
gentype **native_recip** (gentype *x*)	Compute reciprocal over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_rsqrt** (gentype *x*)	Compute inverse square root over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_sin** (gentype *x*)	Compute sine over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_sqrt** (gentype *x*)	Compute square root over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_tan** (gentype *x*)	Compute tangent over an implementation-defined range. The maximum error is implementation-defined.

Table 6.9 *Scalar and Vector Argument Built-in half__ and native__ Math Functions*

Support for denormal values is optional for **half_** functions. The **half_** functions may return any result allowed by *section 7.5.3*, even when -cl-denorms-are-zero (see *section 5.6.3.2*) is not in force. Support for denormal values is implementation-defined for **native_** functions.

The following symbolic constants are available. Their values are of type `float` and are accurate within the precision of a single precision floating-point number.

Constant Name	Description
MAXFLOAT	Value of maximum non-infinite single-precision floating-point number.
HUGE_VALF	A positive float constant expression. **HUGE_VALF** evaluates to +infinity. Used as an error value returned by the built-in math functions.
INFINITY	A constant expression of type float representing positive or unsigned infinity.
NAN	A constant expression of type float representing a quiet NaN.

6.11.2.1 Floating-point macros and pragmas for math.h

The **FP_CONTRACT** pragma can be used to allow (if the state is `on`) or disallow (if the state is `off`) the implementation to contract expressions. Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **FP_CONTRACT** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **FP_CONTRACT** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined.

The pragma definition to set **FP_CONTRACT** is:

```
#pragma OPENCL FP_CONTRACT on-off-switch
```

`on-off-switch` is one of:
ON, **OFF** or **DEFAULT**.
The **DEFAULT** value is **ON**.

The **FP_FAST_FMAF** macro indicates whether the **fma** function is fast compared with direct code for single precision floating-point. If defined, the **FP_FAST_FMAF** macro shall indicate that the **fma** function generally executes about as fast as, or faster than, a multiply and an add of **float** operands.

The macro names given in the following list must use the values specified. These constant expressions are suitable for use in #if preprocessing directives.

```
#define FLT_DIG          6
#define FLT_MANT_DIG     24
#define FLT_MAX_10_EXP   +38
#define FLT_MAX_EXP      +128
#define FLT_MIN_10_EXP   -37
#define FLT_MIN_EXP      -125
#define FLT_RADIX        2
#define FLT_MAX          0x1.fffffep127f
#define FLT_MIN          0x1.0p-126f
#define FLT_EPSILON      0x1.0p-23f
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	Macro for application
FLT_DIG	**CL_FLT_DIG**
FLT_MANT_DIG	**CL_FLT_MANT_DIG**
FLT_MAX_10_EXP	**CL_FLT_MAX_10_EXP**
FLT_MAX_EXP	**CL_FLT_MAX_EXP**
FLT_MIN_10_EXP	**CL_FLT_MIN_10_EXP**

FLT_MIN_EXP	**CL_FLT_MIN_EXP**
FLT_RADIX	**CL_FLT_RADIX**
FLT_MAX	**CL_FLT_MAX**
FLT_MIN	**CL_FLT_MIN**
FLT_EPSILSON	**CL_FLT_EPSILON**

The following macros shall expand to integer constant expressions whose values are returned by **ilogb**(*x*) if *x* is zero or NaN, respectively. The value of **FP_ILOGB0** shall be either {`INT_MIN`} or – {`INT_MAX`}. The value of **FP_ILOGBNAN** shall be either {`INT_MAX`} or {`INT_MIN`}.

The following constants are also available. They are of type `float` and are accurate within the precision of the `float` type.

Constant	Description
M_E_F	Value of e
M_LOG2E_F	Value of $\log_2 e$
M_LOG10E_F	Value of $\log_{10} e$
M_LN2_F	Value of $\log_e 2$
M_LN10_F	Value of $\log_e 10$
M_PI_F	Value of π
M_PI_2_F	Value of $\pi / 2$
M_PI_4_F	Value of $\pi / 4$
M_1_PI_F	Value of $1 / \pi$
M_2_PI_F	Value of $2 / \pi$
M_2_SQRTPI_F	Value of $2 / \sqrt{\pi}$
M_SQRT2_F	Value of $\sqrt{2}$
M_SQRT1_2_F	Value of $1 / \sqrt{2}$

6.11.3 Integer Functions

Table 6.10 describes the built-in integer functions that take scalar or vector arguments. The vector versions of the integer functions operate component-wise. The description is per-component.

We use the generic type name `gentype` to indicate that the function can take `char`, `char{2|3|4|8|16}, uchar, uchar{2|3|4|8|16}, short, short{2|3|4|8|16}, ushort, ushort{2|3|4|8|16}, int, int{2|3|4|8|16}, uint, uint{2|3|4|8|16}, long, long{2|3|4|8|16} ulong,` or `ulong{2|3|4|8|16}` as the type for the arguments. We use the generic type name `ugentype` to refer to unsigned versions of `gentype`. For example, if `gentype` is `char4, ugentype` is `uchar4`. We also use the generic type name `sgentype` to indicate that the function can take a scalar data type i.e. `char, uchar, short, ushort, int, uint, long,` or `ulong` as the type for the arguments. For built-in integer functions that take `gentype` and `sgentype` arguments, the `gentype` argument must be a vector or scalar version of the `sgentype` argument. For example, if `sgentype` is uchar, `gentype` must be `uchar` or `uchar{2|3|4|8|16}`.

For any specific use of a function, the actual type has to be the same for all arguments and the return type unless otherwise specified.

Function	Description
ugentype **abs** (gentype *x*)	Returns | x |.
ugentype **abs_diff** (gentype *x*, gentype *y*)	Returns | x – y | without modulo overflow.
gentype **add_sat** (gentype *x*, gentype *y*)	Returns $x + y$ and saturates the result.
gentype **hadd** (gentype *x*, gentype *y*)	Returns $(x + y) >> 1$. The intermediate sum does not modulo overflow.
gentype **rhadd** (gentype *x*, gentype *y*)[45]	Returns $(x + y + 1) >> 1$. The intermediate sum does not modulo overflow.
gentype **clamp** (gentype *x*, gentype *minval*, gentype *maxval*) gentype **clamp** (gentype *x*, sgentype *minval*, sgentype *maxval*)	Returns **min**(**max**(*x*, *minval*), *maxval*). Results are undefined if *minval* > *maxval*.
gentype **clz** (gentype *x*)	Returns the number of leading 0-bits in *x*, starting at the most significant bit position.
gentype **mad_hi** (gentype *a*, gentype *b*, gentype *c*)	Returns **mul_hi**(*a*, *b*) + *c*.

---

[45] Frequently vector operations need n + 1 bits temporarily to calculate a result. The **rhadd** instruction gives you an extra bit without needing to upsample and downsample. This can be a profound performance win.

gentype **mad_sat** (gentype *a*, gentype *b*, gentype *c*)	Returns *a* * *b* + *c* and saturates the result.
gentype **max** (gentype *x*, gentype *y*) gentype **max** (gentype *x*, sgentype *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*.
gentype **min** (gentype *x*, gentype *y*) gentype **min** (gentype *x*, sgentype *y*)	Returns *y* if *y* < *x*, otherwise it returns *x*.
gentype **mul_hi** (gentype *x*, gentype *y*)	Computes *x* * *y* and returns the high half of the product of *x* and *y*.
gentype **rotate** (gentype *v*, gentype *i*)	For each element in *v*, the bits are shifted left by the number of bits given by the corresponding element in *i* (subject to usual shift modulo rules described in *section 6.3*). Bits shifted off the left side of the element are shifted back in from the right.
gentype **sub_sat** (gentype *x*, gentype *y*)	Returns *x* - *y* and saturates the result.
short **upsample** (char *hi*, uchar *lo*) ushort **upsample** (uchar *hi*, uchar *lo*) short*n* **upsample** (char*n* *hi*, uchar*n* *lo*) ushort*n* **upsample** (uchar*n* *hi*, uchar*n* *lo*) int **upsample** (short *hi*, ushort *lo*) uint **upsample** (ushort *hi*, ushort *lo*) int*n* **upsample** (short*n* *hi*, ushort*n* *lo*) uint*n* **upsample** (ushort*n* *hi*, ushort*n* *lo*) long **upsample** (int *hi*, uint *lo*) ulong **upsample** (uint *hi*, uint *lo*) long*n* **upsample** (int*n* *hi*, uint*n* *lo*) ulong*n* **upsample** (uint*n* *hi*, uint*n* *lo*)	*result*[i] = ((short)*hi*[i] << 8) | *lo*[i] *result*[i] = ((ushort)*hi*[i] << 8) | *lo*[i] *result*[i] = ((int)*hi*[i] << 16) | *lo*[i] *result*[i] = ((uint)*hi*[i] << 16) | *lo*[i] *result*[i] = ((long)*hi*[i] << 32) | *lo*[i] *result*[i] = ((ulong)*hi*[i] << 32) | *lo*[i]

Table 6.10 *Scalar and Vector Integer Argument Built-in Functions*

Table 6.11 describes fast integer functions that can be used for optimizing performance of kernels. We use the generic type name `gentype` to indicate that the function can take `int, int2, int3, int4, int8, int16, uint, uint2, uint4, uint8` or `uint16` as the type for the arguments.

Function	**Description**
gentype **mad24** (gentype *x*, gentype *y*, gentype z)	Multipy two 24-bit integer values *x* and *y* and add the 32-bit integer result to the 32-bit integer *z*. Refer to definition of **mul24** to see how the 24-bit integer multiplication is performed.
gentype **mul24** (gentype *x*, gentype *y*)	Multiply two 24-bit integer values *x* and *y*. *x* and *y* are 32-bit integers but only the low 24-bits are used

	to perform the multiplication. **mul24** should only be used when values in *x* and *y* are in the range [-2^{23}, $2^{23}-1$] if *x* and *y* are signed integers and in the range [0, $2^{24}-1$] if *x* and *y* are unsigned integers. If *x* and *y* are not in this range, the multiplication result is implementation-defined.

Table 6.11 *Fast Integer Built-in Functions*

The macro names given in the following list must use the values specified. The values shall all be constant expressions suitable for use in `#if` preprocessing directives.

```
#define CHAR_BIT    8
#define CHAR_MAX    SCHAR_MAX
#define CHAR_MIN    SCHAR_MIN
#define INT_MAX     2147483647
#define INT_MIN     (-2147483647 – 1)
#define LONG_MAX    0x7fffffffffffffffL
#define LONG_MIN    (-0x7fffffffffffffffL – 1)
#define SCHAR_MAX   127
#define SCHAR_MIN   (-127 – 1)
#define SHRT_MAX    32767
#define SHRT_MIN    (-32767 – 1)
#define UCHAR_MAX   255
#define USHRT_MAX   65535
#define UINT_MAX    0xffffffff
#define ULONG_MAX   0xffffffffffffffffUL
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	Macro for application
CHAR_BIT	**CL_CHAR_BIT**
CHAR_MAX	**CL_CHAR_MAX**
CHAR_MIN	**CL_CHAR_MIN**
INT_MAX	**CL_INT_MAX**
INT_MIN	**CL_INT_MIN**
LONG_MAX	**CL_LONG_MAX**
LONG_MIN	**CL_LONG_MIN**
SCHAR_MAX	**CL_SCHAR_MAX**
SCHAR_MIN	**CL_SCHAR_MIN**
SHRT_MAX	**CL_SHRT_MAX**
SHRT_MIN	**CL_SHRT_MIN**
UCHAR_MAX	**CL_UCHAR_MAX**
USHRT_MAX	**CL_USHRT_MAX**
UINT_MAX	**CL_UINT_MAX**
ULONG_MAX	**CL_ULONG_MAX**

6.11.4 Common Functions[46]

Table 6.12 describes the list of built-in common functions. These all operate component-wise. The description is per-component. We use the generic type name `gentype` to indicate that the function can take `float, float2, float3, float4, float8,` or `float16` as the type for the arguments.

The built-in common functions are implemented using the round to nearest even rounding mode.

Function	Description
gentype **clamp** (gentype x, gentype *minval*, gentype *maxval*) gentype **clamp** (gentype x, float *minval*, float *maxval*)	Returns **fmin**(**fmax**(x, *minval*), *maxval*). Results are undefined if *minval* > *maxval*.
gentype **degrees** (gentype *radians*)	Converts *radians* to degrees, i.e. (180 / π) * *radians*.
gentype **max** (gentype x, gentype y) gentype **max** (gentype x, float y)	Returns y if $x < y$, otherwise it returns x. If x and y are infinite or NaN, the return values are undefined.
gentype **min** (gentype x, gentype y) gentype **min** (gentype x, float y)	Returns y if $y < x$, otherwise it returns x. If x and y are infinite or NaN, the return values are undefined.
gentype **mix** (gentype x, gentype y, gentype a) gentype **mix** (gentype x, gentype y, float a)	Returns the linear blend of x & y implemented as: $x + (y - x) * a$ a must be a value in the range 0.0 … 1.0. If a is not in the range 0.0 … 1.0, the return values are undefined.
gentype **radians** (gentype *degrees*)	Converts *degrees* to radians, i.e. (π / 180) * *degrees*.
gentype **step** (gentype *edge*, gentype x) gentype **step** (float *edge*, gentype x)	Returns 0.0 if $x <$ *edge*, otherwise it returns 1.0.
genType **smoothstep** (genType *edge0*,	Returns 0.0 if $x <=$ *edge0* and 1.0 if $x >=$ *edge1* and

---

[46] The **mix** and **smoothstep** functions can be implemented using contractions such as **mad** or **fma**.

genType *edge1*, genType *x*) genType **smoothstep** (float *edge0*, float *edge1*, genType *x*)	performs smooth Hermite interpolation between 0 and 1when *edge0* < *x* < *edge1*. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to: gentype t; t = clamp ((x – edge0) / (edge1 – edge0), 0, 1); return t * t * (3 – 2 * t); Results are undefined if *edge0* >= *edge1* or if *x, edge0* or *edge1* is a NaN.
gentype **sign** (gentype *x*)	Returns 1.0 if *x* > 0, -0.0 if *x* = -0.0, +0.0 if *x* = +0.0, or –1.0 if *x* < 0. Returns 0.0 if *x* is a NaN.

Table 6.12 *Scalar and Vector Argument Built-in Common Function Table*

6.11.5 Geometric Functions[47]

Table 6.13 describes the list of built-in geometric functions. These all operate component-wise. The description is per-component. `floatn` is `float`, `float2`, `float3`, or `float4`. The built-in geometric functions are implemented using the round to nearest even rounding mode.

Function	Description
float4 **cross** (float4 *p0*, float4 *p1*) float3 **cross** (float3 *p0*, float3 *p1*)	Returns the cross product of *p0.xyz* and *p1.xyz*. The *w* component of float4 result returned will be 0.0.
float **dot** (float*n* *p0*, float*n* *p1*)	Compute dot product.
float **distance** (float*n* *p0*, float*n* *p1*)	Returns the distance between *p0* and *p1*. This is calculated as **length**(*p0* – *p1*).
float **length** (float*n* *p*)	Return the length of vector *p*, i.e., $\sqrt{p.x^2 + p.y^2 + \ldots}$
float*n* **normalize** (float*n* *p*)	Returns a vector in the same direction as *p* but with a length of 1.
float **fast_distance** (float*n* *p0*, float*n* *p1*)	Returns **fast_length**(*p0* – *p1*).
float **fast_length** (float*n* *p*)	Returns the length of vector *p* computed as: **half_sqrt**($p.x^2 + p.y^2 + \ldots.$)

float*n* **fast_normalize** (float*n* *p*)	Returns a vector in the same direction as *p* but with a length of 1. **fast_normalize** is computed as:
	p * **half_rsqrt** ($p.x^2 + p.y^2 + \ldots$)
	The result shall be within 8192 ulps error from the infinitely precise result of
	if (**all**(*p* == 0.0f))
	result = *p*;
	else
	result = *p* / **sqrt** ($p.x^2 + p.y^2 + \ldots$);
	with the following exceptions:
	1) If the sum of squares is greater than FLT_MAX then the value of the floating-point values in the result vector are undefined.

---

[47] The geometric functions can be implemented using contractions such as **mad** or **fma**.

	2) If the sum of squares is less than FLT_MIN then the implementation may return back *p*. 3) If the device is in "denorms are flushed to zero" mode, individual operand elements with magnitude less than **sqrt**(FLT_MIN) may be flushed to zero before proceeding with the calculation.

Table 6.13 *Scalar and Vector Argument Built-in Geometric Function Table*

6.11.6 Relational Functions

The relational and equality operators (<, <=, >, >=, !=, ==) can be used with scalar and vector built-in types and produce a scalar or vector signed integer result respectively as described in *section 6.3*.

The functions[48] described in *table 6.14* can be used with built-in scalar or vector types as arguments and return a scalar or vector integer result. The argument type `gentype` refers to the following built-in types: `char, charn, uchar, ucharn, short, shortn, ushort, ushortn, int, intn, uint, uintn, long, longn, ulong, ulongn, float,` and `floatn`. The argument type `igentype` refers to the built-in signed integer types i.e. `char`, `charn`, `short`, `shortn`, `int`, `intn`, `long` and `longn`. The argument type `ugentype` refers to the built-in unsigned integer types i.e. `uchar`, `ucharn`, `ushort`, `ushortn`, `uint`, `uintn`, `ulong` and `ulongn`. *n* is 2, 3, 4, 8, or 16.

The functions **isequal**, **isnotequal**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, **isfinite**, **isinf**, **isnan**, **isnormal**, **isordered**, **isunordered** and **signbit** described in *table 6.14* shall return a 0 if the specified relation is *false* and a 1 if the specified relation is true for scalar argument types. These functions shall return a 0 if the specified relation is *false* and a –1 (i.e. all bits set) if the specified relation is *true* for vector argument types.

The relational functions **isequal**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, and **islessgreater** always return 0 if either argument is not a number (NaN). **isnotequal** returns 1 if one or both arguments are not a number (NaN) and the argument type is a scalar and returns -1 if one or both arguments are not a number (NaN) and the argument type is a vector.

Function	**Description**
int **isequal** (float *x*, float *y*) int*n* **isequal** (float*n x*, float*n y*)	Returns the component-wise compare of *x* == *y*.
int **isnotequal** (float *x*, float *y*) int*n* **isnotequal** (float*n x*, float*n y*)	Returns the component-wise compare of *x* != *y*.
int **isgreater** (float *x*, float *y*) int*n* **isgreater** (float*n x*, float*n y*)	Returns the component-wise compare of *x* > *y*.
int **isgreaterequal** (float *x*, float *y*) int*n* **isgreaterequal** (float*n x*, float*n y*)	Returns the component-wise compare of *x* >= *y*.
int **isless** (float *x*, float *y*) int*n* **isless** (float*n x*, float*n y*)	Returns the component-wise compare of *x* < *y*.
int **islessequal** (float *x*, float *y*) int*n* **islessequal** (float*n x*, float*n y*)	Returns the component-wise compare of *x* <= *y*.

---

[48] If an implementation extends this specification to support IEEE-754 flags or exceptions, then all builtin functions defined in *table 6.14* shall proceed without raising the *invalid* floating-point exception when one or more of the operands are NaNs.

int **islessgreater** (float *x*, float *y*) int*n* **islessgreater** (float*n* *x*, float*n* *y*)	Returns the component-wise compare of (*x* < *y*) || (*x* > *y*) .
int **isfinite** (float) int*n* **isfinite** (float*n*)	Test for finite value.
int **isinf** (float) int*n* **isinf** (float*n*)	Test for infinity value (+ve or –ve) .
int **isnan** (float) int*n* **isnan** (float*n*)	Test for a NaN.
int **isnormal** (float) int*n* **isnormal** (float*n*)	Test for a normal value.
int **isordered** (float *x*, float *y*) int*n* **isordered** (float*n* *x*, float*n* *y*)	Test if arguments are ordered. **isordered**() takes arguments *x* and *y*, and returns the result **isequal**(*x*, *x*) && **isequal**(*y*, *y*).
int **isunordered** (float *x*, float *y*) int*n* **isunordered** (float*n* *x*, float*n* *y*)	Test if arguments are unordered. **isunordered**() takes arguments *x* and *y*, returning non-zero if *x* or *y* is NaN, and zero otherwise.
int **signbit** (float) int*n* **signbit** (float*n*)	Test for sign bit. The scalar version of the function returns a 1 if the sign bit in the float is set else returns 0. The vector version of the function returns the following for each component in float*n*: a -1 if the sign bit in the float is set else returns 0.
int **any** (igentype *x*)	Returns 1 if the most significant bit in any component of *x* is set; otherwise returns 0.
int **all** (igentype *x*)	Returns 1 if the most significant bit in all components of *x* is set; otherwise returns 0.
gentype **bitselect** (gentype *a*, gentype *b*, gentype *c*)	Each bit of the result is the corresponding bit of *a* if the corresponding bit of *c* is 0. Otherwise it is the corresponding bit of *b*.
gentype **select** (gentype *a*, gentype *b*, igentype *c*) gentype **select** (gentype *a*, gentype *b*, ugentype *c*)	For each component of a vector type, *result[i]* = if MSB of *c[i]* is set ? *b[i]* : *a[i]*. For a scalar type, *result* = *c* ? *b* : *a*.

Table 6.14 *Scalar and Vector Relational Functions*

6.11.7 Vector Data Load and Store Functions

Table 6.15 describes the list of supported functions that allow you to read and write vector types from a pointer to memory. We use the generic type `gentype` to indicate the built-in data types `char, uchar, short, ushort, int, uint, long, ulong,` or `float`. We use the generic type name `gentype`*n* to represent n-element vectors of `gentype` elements. We use the type name `half`*n* to represent n-element vectors of `half` elements[49]. The suffix *n* is also used in the function names (i.e. **vload*n***, **vstore*n*** etc.), where *n* = 2, 3, 4, 8 or 16.

Function	Description
gentype*n* **vload*n*** (size_t *offset*, const __global gentype **p*) gentype*n* **vload*n*** (size_t *offset*, const __local gentype **p*) gentype*n* **vload*n*** (size_t *offset*, const __constant gentype **p*) gentype*n* **vload*n*** (size_t *offset*, const __private gentype **p*)	Return sizeof (gentype*n*) bytes of data read from address (*p* + (*offset* * *n*)). The address computed as (*p* + (*offset* * *n*)) must be 8-bit aligned if gentype is char, uchar; 16-bit aligned if gentype is short, ushort; 32-bit aligned if gentype is int, uint, float; 64-bit aligned if gentype is long, ulong.
void **vstore*n*** (gentype*n* *data*, size_t *offset*, __global gentype **p*) void **vstore*n*** (gentype*n* *data*, size_t *offset*, __local gentype **p*) void **vstore*n*** (gentype*n* *data*, size_t *offset*, __private gentype **p*)	Write sizeof (gentype*n*) bytes given by *data* to address (*p* + (*offset* * *n*)). The address computed as (*p* + (*offset* * *n*)) must be 8-bit aligned if gentype is char, uchar; 16-bit aligned if gentype is short, ushort; 32-bit aligned if gentype is int, uint, float; 64-bit aligned if gentype is long, ulong.
float **vload_half** (size_t *offset*, const __global half **p*) float **vload_half** (size_t *offset*, const __local half **p*) float **vload_half** (size_t *offset*, const __constant half **p*) float **vload_half** (size_t *offset*,	Read sizeof (half) bytes of data from address (*p* + *offset*). The data read is interpreted as a half value. The half value is converted to a float value and the float value is returned. The read address computed as (*p* + *offset*) must be 16-bit aligned.

---

[49] The half*n* type is only defined by the **cl_khr_fp16** extension described in *section 9.6*.

const __private half **p*)	
float*n* **vload_half*n*** (size_t *offset*, const __global half **p*) float*n* **vload_half*n*** (size_t *offset*, const __local half **p*) float*n* **vload_half*n*** (size_t *offset*, const __constant half **p*) float*n* **vload_half*n*** (size_t *offset*, const __private half **p*)	Read sizeof (half*n*) bytes of data from address (*p* + (*offset* * *n*)). The data read is interpreted as a half*n* value. The half*n* value read is converted to a float*n* value and the float*n* value is returned. The read address computed as (p + (*offset* * *n*)) must be 16-bit aligned.
void **vstore_half** (float *data*, size_t *offset*, __global half **p*) void **vstore_half_*rte*** (float *data*, size_t *offset*, __global half **p*) void **vstore_half_*rtz*** (float *data*, size_t *offset*, __global half **p*) void **vstore_half_*rtp*** (float *data*, size_t *offset*, __global half **p*) void **vstore_half_*rtn*** (float *data*, size_t *offset*, __global half **p*) void **vstore_half** (float *data*, size_t *offset*, __local half **p*) void **vstore_half_*rte*** (float *data*, size_t *offset*, __local half **p*) void **vstore_half_*rtz*** (float *data*, size_t *offset*, __local half **p*) void **vstore_half_*rtp*** (float *data*, size_t *offset*, __local half **p*) void **vstore_half_*rtn*** (float *data*, size_t *offset*, __local half **p*) void **vstore_half** (float *data*, size_t *offset*, __private half **p*) void **vstore_half_*rte*** (float *data*, size_t *offset*, __private half **p*) void **vstore_half_*rtz*** (float *data*, size_t *offset*, __private half **p*) void **vstore_half_*rtp*** (float *data*, size_t *offset*, __private half **p*) void **vstore_half_*rtn*** (float *data*, size_t *offset*, __private half **p*)	The float value given by *data* is first converted to a half value using the appropriate rounding mode. The half value is then written to address computed as (*p* + *offset*). The address computed as (*p* + *offset*) must be 16-bit aligned. **vstore_half** use the current rounding mode. The default current rounding mode is round to nearest even.

void **vstore_half*n*** (float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_*rte*** (float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_*rtz*** (float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_*rtp*** (float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_*rtn*** (float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*** (float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_*rte*** (float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_*rtz*** (float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_*rtp*** (float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_*rtn*** (float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*** (float*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_*rte*** (float*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_*rtz*** (float*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_*rtp*** (float*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_*rtn*** (float*n* *data*, size_t *offset*, __private half *p)	The float*n* value given by *data* is converted to a half*n* value using the appropriate rounding mode. The half*n* value is then written to address computed as (*p* + (*offset* * *n*)). The address computed as (*p* + (*offset* * *n*)) must be 16-bit aligned. **vstore_half*n*** uses the current rounding mode. The default current rounding mode is round to nearest even.
float*n* **vloada_half*n*** (size_t *offset*, const __global half *p) float*n* **vloada_half*n*** (size_t *offset*, const __local half *p) float*n* **vloada_half*n*** (size_t *offset*, const __constant half *p) float*n* **vloada_half*n*** (size_t *offset*, const __private half *p)	For n = 1, 2, 4, 8 and 16 read sizeof (half*n*) bytes of data from address (*p* + (*offset* * *n*)). The data read is interpreted as a half*n* value. The half*n* value read is converted to a float*n* value and the float*n* value is returned. The address computed as (*p* + (*offset* * *n*)) must be aligned to sizeof (half*n*) bytes. For n = 3, **vloada_half3** reads a half3 from address (*p* + (*offset* * *4*)) and returns a float3.

	The address computed as (*p* + (*offset* * *4*)) must be aligned to sizeof (half) * 4 bytes.
void **vstorea_half*n*** (float*n* *data*, size_t *offset*, __global half **p*) void **vstorea_half*n*_*rte*** (float*n* *data*, size_t *offset*, __global half **p*) void **vstorea_half*n*_*rtz*** (float*n* *data*, size_t *offset*, __global half **p*) void **vstorea_half*n*_*rtp*** (float*n* *data*, size_t *offset*, __global half **p*) void **vstorea_half*n*_*rtn*** (float*n* *data*, size_t *offset*, __global half **p*) void **vstorea_half*n*** (float*n* *data*, size_t *offset*, __local half **p*) void **vstorea_half*n*_*rte*** (float*n* *data*, size_t *offset*, __local half **p*) void **vstorea_half*n*_*rtz*** (float*n* *data*, size_t *offset*, __local half **p*) void **vstorea_half*n*_*rtp*** (float*n* *data*, size_t *offset*, __local half **p*) void **vstorea_half*n*_*rtn*** (float*n* *data*, size_t *offset*, __local half **p*) void **vstorea_half*n*** (float*n* *data*, size_t *offset*, __private half **p*) void **vstorea_half*n*_*rte*** (float*n* *data*, size_t *offset*, __private half **p*) void **vstorea_half*n*_*rtz*** (float*n* *data*, size_t *offset*, __private half **p*) void **vstorea_half*n*_*rtp*** (float*n* *data*, size_t *offset*, __private half **p*) void **vstorea_half*n*_*rtn*** (float*n* *data*, size_t *offset*, __private half **p*)	The float*n* value given by *data* is converted to a half*n* value using the appropriate rounding mode. For n = 1, 2, 4, 8 and 16, the half*n* value is written to the address computed as (*p* + (*offset* * *n*)). The address computed as (*p* + (*offset* * *n*)) must be aligned to sizeof (half*n*) bytes. For n = 3, the half*3* value is written to the address computed as (*p* + (*offset* * *4*)). The address computed as (*p* + (*offset* * *4*)) must be aligned to sizeof (half) * 4 bytes. **vstorea_half*n*** uses the current rounding mode. The default current rounding mode is round to nearest even.

Table 6.15 *Vector Data Load and Store Functions*

The results of vector data load and store functions are undefined if the address being read from or written to is not correctly aligned as described in *table 6.15*. The pointer argument p can be a pointer to `__global`, `__local` or `__private` memory for store functions described in *table 6.15*. The pointer argument p can be a pointer to `__global`, `__local`, `__constant` or `__private` memory for load functions described in *table 6.15*.

NOTE: **vload*3***, and **vload_half*3*** read *x*, *y*, *z* components from address (*p* + (*offset* * 3)) into a 3-

component vector. **vstore*3***, and **vstore_half*3*** write *x*, *y*, *z* components from a 3-component vector to address (*p* + (*offset* * 3)).

In addition **vloada_half*3*** reads *x*, *y*, *z* components from address (*p* + (*offset* * 4)) into a 3-component vector and **vstorea_half*3*** writes *x*, *y*, *z* components from a 3-component vector to address (*p* + (*offset* * 4)).

6.11.8 Synchronization Functions

The OpenCL C programming language implements the following synchronization functions.

Function	**Description**
void **barrier** (cl_mem_fence_flags *flags*)	All work-items in a work-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 work-group executing the kernel. If **barrier** is inside a conditional statement, then all work-items must enter the conditional if any work-item enters the conditional statement and executes the **barrier**. If **barrer** is inside a loop, all work-items must execute the **barrier** for each iteration of the loop before any are allowed to continue execution beyond the **barrier**. The **barrier** function also queues a memory fence (reads and writes) to ensure correct ordering of memory operations to local or global memory. The *flags* argument specifies the memory address space and can be set to a combination of the following literal values. CLK_LOCAL_MEM_FENCE - The **barrier** function will either flush any variables stored in local memory or queue a memory fence to ensure correct ordering of memory operations to local memory. CLK_GLOBAL_MEM_FENCE – The **barrier** function will queue a memory fence to ensure correct ordering of memory operations to global memory. This can be useful when work-items, for example, write to buffer or image objects and then want to read the updated data.

Table 6.16 *Built-in Synchronization Functions*

6.11.9 Explicit Memory Fence Functions

The OpenCL C programming language implements the following explicit memory fence functions to provide ordering between memory operations of a work-item.

Function	Description
void **mem_fence** (cl_mem_fence_flags *flags*)	Orders loads and stores of a work-item executing a kernel. This means that loads and stores preceding the **mem_fence** will be committed to memory before any loads and stores following the **mem_fence**. The *flags* argument specifies the memory address space and can be set to a combination of the following literal values: CLK_LOCAL_MEM_FENCE CLK_GLOBAL_MEM_FENCE.
void **read_mem_fence** (cl_mem_fence_flags *flags*)	Read memory barrier that orders only loads. The *flags* argument specifies the memory address space and can be set to to a combination of the following literal values: CLK_LOCAL_MEM_FENCE CLK_GLOBAL_MEM_FENCE.
void **write_mem_fence** (cl_mem_fence_flags *flags*)	Write memory barrier that orders only stores. The *flags* argument specifies the memory address space and can be set to to a combination of the following literal values: CLK_LOCAL_MEM_FENCE CLK_GLOBAL_MEM_FENCE.

Table 6.17 *Built-in Explicit Memory Fence Functions*

6.11.10 Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch

The OpenCL C programming language implements the following functions that provide asynchronous copies between global and local memory and a prefetch from global memory.

We use the generic type name `gentype` to indicate the built-in data types `char`, `char{2|4|8|16}`, `uchar`, `uchar{2|4|8|16}`, `short`, `short{2|4|8|16}`, `ushort`, `ushort{2|4|8|16}`, `int`, `int{2|4|8|16}`, `uint`, `uint{2|4|8|16}`, `long`, `long{2|4|8|16}`, `ulong`, `ulong{2|4|8|16}`, or `float`, `float{2|4|8|16}` as the type for the arguments unless otherwise stated.

Function	**Description**
event_t **async_work_group_copy** (__local gentype **dst*, const __global gentype **src*, size_t *num_gentypes*, event_t *event*) event_t **async_work_group_copy** (__global gentype **dst*, const __local gentype **src*, size_t *num_gentypes*, event_t *event*)	Perform an async copy of *num_gentypes* `gentype` elements from *src* to *dst*. The async copy is performed by all work-items in a work-group and this built-in function must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined. Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. If *event* argument is non-zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit synchronization of source data such as using a **barrier** before performing the copy.
event_t **async_work_group_strided_copy** (__local gentype **dst*, const __global gentype **src*, size_t *num_gentypes*, size_t *src_stride*, event_t *event*)	Perform an async gather of *num_gentypes* `gentype` elements from *src* to *dst*. The *src_stride* is the stride in elements for each gentype element read from *src*. The *dst_stride* is the stride in elements for each gentype element written to *dst*. The async gather is performed by all work-items in a work-group.

event_t **async_work_group_strided_copy** (__global gentype **dst*, const __local gentype **src*, size_t *num_gentypes*, size_t *dst_stride*, event_t *event*)	This built-in function must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined. Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_strided_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. If *event* argument is non-zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit synchronization of source data such as using a **barrier** before performing the copy. The behavior of **async_work_group_strided_copy** is undefined if *src_stride* or *dst_stride* is 0, or if the *src_stride* or *dst_stride* values cause the *src* or *dst* pointers to exceed the upper bounds of the address space during the copy.
void **wait_group_events** (int *num_events*, event_t **event_list*)	Wait for events that identify the **async_work_group_copy** operations to complete. The event objects specified in *event_list* will be released after the wait is performed. This function must be encountered by all work-items in a work-group executing the kernel with the same *num_events* and event objects specified in *event_list*; otherwise the results are undefined.
void **prefetch** (const __global gentype **p*, size_t *num_gentypes*)	Prefetch *num_gentypes* * sizeof(`gentype`) bytes into the global cache. The prefetch instruction is applied to a work-item in a work-group and does not affect the functional behavior of the kernel.

Table 6.18 *Built-in Async Copy and Prefetch Functions*

6.11.11 Atomic Functions

The OpenCL C programming language implements the following functions that provide atomic operations on 32-bit signed, unsigned integers and single precision floating-point[50] to locations in `__global` or `__local` memory.

Function	**Description**
int **atomic_add** (volatile __global int **p*, int *val*) unsigned int **atomic_add** (volatile __global unsigned int **p*, unsigned int *val*) int **atomic_add** (volatile __local int **p*, int *val*) unsigned int **atomic_add** (volatile __local unsigned int **p*, unsigned int *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* + *val*) and store result at location pointed by *p*. The function returns *old*.
int **atomic_sub** (volatile __global int **p*, int *val*) unsigned int **atomic_sub** (volatile __global unsigned int **p*, unsigned int *val*) int **atomic_sub** (volatile __local int **p*, int *val*) unsigned int **atomic_sub** (volatile __local unsigned int **p*, unsigned int *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* - *val*) and store result at location pointed by *p*. The function returns *old*.
int **atomic_xchg** (volatile __global int **p*, int *val*) unsigned int **atomic_xchg** (volatile __global unsigned int **p*, unsigned int *val*) float **atomic_xchg** (volatile __global float **p*, float *val*) int **atomic_xchg** (volatile __local int **p*, int *val*) unsigned int **atomic_xchg** (volatile __local unsigned int **p*, unsigned int *val*) float **atomic_xchg** (volatile __local float **p*, float *val*)	Swaps the *old* value stored at location *p* with new value given by *val*. Returns *old* value.
int **atomic_inc** (volatile __global int **p*)	Read the 32-bit value (referred to as *old*)

---

[50] Only the **atomic_xchg** operation is supported for single precision floating-point data type.

unsigned int **atomic_inc** (volatile __global unsigned int *p) int **atomic_inc** (volatile __local int *p) unsigned int **atomic_inc** (volatile __local unsigned int *p)	stored at location pointed by *p*. Compute (*old* + *1*) and store result at location pointed by *p*. The function returns *old*.
int **atomic_dec** (volatile __global int *p) unsigned int **atomic_dec** (volatile __global unsigned int *p) int **atomic_dec** (volatile __local int *p) unsigned int **atomic_dec** (volatile __local unsigned int *p)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* - *1*) and store result at location pointed by *p*. The function returns *old*.
int **atomic_cmpxchg** (volatile __global int *p, int *cmp*, int *val*) unsigned int **atomic_cmpxchg** (volatile __global unsigned int *p, unsigned int *cmp*. unsigned int *val*) int **atomic_cmpxchg** (volatile __local int *p, int *cmp*, int *val*) unsigned int **atomic_cmpxchg** (volatile __local unsigned int *p, unsigned int *cmp*. unsigned int *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* == *cmp*) ? *val* : *old* and store result at location pointed by *p*. The function returns *old*.
int **atomic_min** (volatile __global int *p, int *val*) unsigned int **atomic_min** (volatile __global unsigned int *p, unsigned int *val*) int **atomic_min** (volatile __local int *p, int *val*) unsigned int **atomic_min** (volatile __local unsigned int *p, unsigned int *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute **min**(*old*, *val*) and store minimum value at location pointed by *p*. The function returns *old*.
int **atomic_max** (volatile __global int *p, int *val*) unsigned int **atomic_max** (volatile __global unsigned int *p, unsigned int *val*) int **atomic_max** (volatile __local int *p, int *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute **max**(*old*, *val*) and store maximum value at location pointed by *p*. The function returns *old*.

unsigned int **atomic_max** (volatile __local unsigned int **p*, unsigned int *val*)	
int **atomic_and** (volatile __global int **p*, int val) unsigned int **atomic_and** (volatile __global unsigned int **p*, unsigned int *val*) int **atomic_and** (volatile __local int **p*, int val) unsigned int **atomic_and** (volatile __local unsigned int **p*, unsigned int *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* & val) and store result at location pointed by *p*. The function returns *old*.
int **atomic_or** (volatile __global int **p*, int *val*) unsigned int **atomic_or** (volatile __global unsigned int **p*, unsigned int *val*) int **atomic_or** (volatile __local int **p*, int *val*) unsigned int **atomic_or** (volatile __local unsigned int **p*, unsigned int *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* | val) and store result at location pointed by *p*. The function returns *old*.
int **atomic_xor** (volatile __global int **p*, int val) unsigned int **atomic_xor** (volatile __global unsigned int **p*, unsigned int *val*) int **atomic_xor** (volatile __local int **p*, int val) unsigned int **atomic_xor** (volatile __local unsigned int **p*, unsigned int *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* ^ val) and store result at location pointed by *p*. The function returns *old*.

Table 6.19 *Built-in Atomic Functions*

NOTE: The atomic built-in functions that use the **atom_** prefix and are described by the following extensions

- **cl_khr_global_int32_base_atomics**
- **cl_khr_global_int32_extended_atomics**
- **cl_khr_local_int32_base_atomics**
- **cl_khr_local_int32_extended_atomics**

in *sections 9.5 and 9.6* of the OpenCL 1.0 specification are also supported.

6.11.12 Miscellaneous Vector Functions

The OpenCL C programming language implements the following additional built-in vector functions. We use the generic type name `gentypen` (or `gentypem`) to indicate the built-in data types `char{2|4|8|16}, uchar{2|4|8|16}, short{2|4|8|16}, ushort{2|4|8|16}, half{2|4|8|16}`[51]`, int{2|4|8|16}, uint{2|4|8|16}, long{2|4|8|16}, ulong{2|4|8|16}, float{2|4|8|16}` or `double{2|4|8|16}`[52] as the type for the arguments unless otherwise stated. We use the generic name `ugentypen` to indicate the built-in unsigned integer data types.

Function	Description
int **vec_step** (gentype*n a*) int **vec_step** (char3 *a*) int **vec_step** (uchar3 *a*) int **vec_step** (short3 *a*) int **vec_step** (ushort3 *a*) int **vec_step** (half3 *a*) int **vec_step** (int3 *a*) int **vec_step** (uint3 *a*) int **vec_step** (long3 *a*) int **vec_step** (ulong3 *a*) int **vec_step** (float3 *a*) int **vec_step**(double3 *a*) int **vec_step**(*type*)	The **vec_step** built-in function takes a built-in scalar or vector data type argument and returns an integer value representing the number of elements in the scalar or vector. For all scalar types, **vec_step** returns 1. The **vec_step** built-in functions that take a 3-component vector return 4. **vec_step** may also take a pure type as an argument, e.g. **vec_step**(`float2`)
gentype*n* **shuffle** (gentype*m x*, ugentype*n mask*) gentype*n* **shuffle2** (gentype*m x*, gentype*m y*, ugentype*n mask*)	The **shuffle** and **shuffle2** built-in functions construct a permutation of elements from one or two input vectors respectively that are of the same type, returning a vector with the same element type as the input and length that is the same as the shuffle mask. The size of each element in the *mask* must match the size of each element in the result. For **shuffle**, only the **ilogb**(2*m*-1) least significant bits of each *mask* element are considered. For **shuffle2**, only the **ilogb**(2*m*-1)+1 least significant bits of each *mask* element are considered. Other bits in the mask shall be ignored. The elements of the input vectors are numbered from left to right across one or both of the vectors. For this

---

[51] Only if the **cl_khr_fp16** extension is supported.
[52] Only if the **cl_khr_fp64** extension is supported.

purpose, the number of elements in a vector is given by **vec_step**(gentype*m*). The shuffle *mask* operand specifies, for each element of the result vector, which element of the one or two input vectors the result element gets.

Examples:

```
uint4   mask = (uint4)(3, 2,
                       1, 0);
float4 a;
float4 r = shuffle(a, mask);
// r.s0123 = a.wzyx

uint8  mask = (uint8)(0, 1, 2, 3,
                      4, 5, 6, 7);
float4 a, b;
float8 r = shuffle2(a, b, mask);
// r.s0123 = a.xyzw
// r.s4567 = b.xyzw


uint4 mask;
float8 a;
float4 b;
b = shuffle(a, mask);
```

Examples that are not valid are:

```
uint8   mask;
short16 a;
short8  b;
b = shuffle(a, mask); ← not valid
```

Table 6.20 *Built-in Miscellaneous Vector Functions*

6.11.13 Image Read and Write Functions

The built-in functions defined in this section can only be used with image memory objects created with **clCreateImage2D**, or **clCreateImage3D**. An image memory object can be accessed by specific function calls that read from and/or write to specific locations in the image.

Image memory objects that are being read by a kernel should be declared with the `__read_only` qualifier. **write_image** calls to image memory objects declared with the `__read_only` qualifier will generate a compilation error. Image memory objects that are being written to by a kernel should be declared with the `__write_only` qualifier. **read_image** calls to image memory objects declared with the `__write_only` qualifier will generate a compilation error. **read_image** and **write_image** calls to the same image memory object in a kernel are not supported.

The **read_image** calls returns a four component floating-point, integer or unsigned integer color value. The color values returned by **read_image** are identified as `x`, `y`, `z`, `w` where `x` refers to the red component, `y` refers to the green component, `z` refers to the blue component and `w` refers to the alpha component.

6.11.13.1 Samplers

The image read functions take a sampler argument. The sampler can be passed as an argument to the kernel using **clSetKernelArg**, or it can be a constant variable of type `sampler_t` declared in the program source.

Sampler variables in a program are declared to be of type `sampler_t`. The `sampler_t` type is a 32-bit unsigned int constant and is interpreted as a bit-field that specifies the following properties:

- Addressing Mode
- Filter Mode
- Normalized Coordinates

These properties control how elements of an image object are read by **read_image{f|i|ui}**.

Samplers can also be declared as global constants in the program source using the following syntax.

const sampler_t <sampler name> = <value>

The sampler fields are described in *table 6.21*.

Sampler State	Description
`<normalized coords>`	Specifies whether the *x*, *y* and *z* coordinates are passed in as normalized or unnormalized values. This must be a literal value and can be one of the following predefined enums: CLK_NORMALIZED_COORDS_TRUE or CLK_NORMALIZED_COORDS_FALSE. The samplers used with an image in multiple calls to **read_image{f|i|ui}** declared in a kernel must use the same value for `<normalized coords>`.
`<address mode>`	Specifies the image addressing-mode i.e. how out-of-range image coordinates are handled. This must be a literal value and can be one of the following predefined enums: CLK_ADDRESS_MIRRORED_REPEAT - Flip the image coordinate at every integer junction. This `address mode` can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined. CLK_ADDRESS_REPEAT – out-of-range image coordinates are wrapped to the valid range. This `address mode` can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined. CLK_ADDRESS_CLAMP_TO_EDGE – out-of-range image coordinates are clamped to the extent. CLK_ADDRESS_CLAMP[53] – out-of-range image coordinates will return a border color. CLK_ADDRESS_NONE – for this address mode the programmer guarantees that the image coordinates used to sample elements of the image refer to a location inside the image; otherwise the results are undefined.
`<filter mode>`	Specifies the filtering mode to use. This must be a literal

---

[53] This is similar to the GL_ADDRESS_CLAMP_TO_BORDER addressing mode.

	value and can be one of the following predefined enums: CLK_FILTER_NEAREST or CLK_FILTER_LINEAR. Refer to *section 8.2* for a description of these filtering modes.

Table 6.21 *Sampler Descriptor*

Samplers cannot be declared as arrays, pointers, or be used as the type for local variables inside a function or as the return value of a function defined in a program. Samplers cannot be passed as arguments to functions called by a `__kernel` function. A sampler argument to a `__kernel` function cannot be modified.

Examples:

```
const sampler_t  samplerA = CLK_NORMALIZED_COORDS_TRUE |
                            CLK_ADDRESS_REPEAT         |
                            CLK_FILTER_NEAREST;
```

samplerA specifies a sampler that uses normalized coordinates, the repeat addressing mode and a nearest filter.

The maximum number of samplers that can be declared in a kernel can be queried using the CL_DEVICE_MAX_SAMPLERS token in **clGetDeviceInfo**.

6.11.13.1.1 Determining the border color

If <`addressing mode`> in sampler is CLK_ADDRESS_CLAMP, then out-of-range image coordinates return the border color. The border color selected depends on the image channel order and can be one of the following values:

- If the image channel order is CL_A, CL_INTENSITY, CL_Rx, CL_RA, CL_RGx, CL_RGBx, CL_ARGB, CL_BGRA, or CL_RGBA, the border color is (`0.0f, 0.0f, 0.0f, 0.0f`).

- If the image channel order is CL_R, CL_RG, CL_RGB, or CL_LUMINANCE, the border color is (`0.0f, 0.0f, 0.0f, 1.0f`)

6.11.13.2 Built-in Image Functions

The following built-in function calls to read and write images are supported.

Function	Description
float4 **read_imagef** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) float4 **read_imagef** (image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate *(x, y)* to do an element lookup in the 2D image object specified by *image*. **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. **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. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to CLK_FILTER_NEAREST, normalized coordinates set to CLK_NORMALIZED_COORDS_FALSE and addressing mode set to CLK_ADDRESS_CLAMP_TO_EDGE, CLK_ADDRESS_CLAMP or CLK_ADDRESS_NONE; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.
int4 **read_imagei** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) int4 **read_imagei** (image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate *(x, y)* to do an element lookup in the 2D image object specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects

uint4 **read_imageui** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) uint4 **read_imageui** (image2d_t *image*, sampler_t *sampler*, float2 *coord*)	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_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** 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_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to CLK_FILTER_NEAREST; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to CLK_NORMALIZED_COORDS_FALSE and addressing mode set to CLK_ADDRESS_CLAMP_TO_EDGE, CLK_ADDRESS_CLAMP or CLK_ADDRESS_NONE; otherwise the values returned are undefined.
void **write_imagef** (image2d_t *image*, int2 *coord*, float4 *color*) void **write_imagei** (image2d_t *image*, int2 *coord*, int4 *color*) void **write_imageui** (image2d_t *image*, int2 *coord*, uint4 *color*)	Write *color* value to location specified by coordinate *(x, y)* in the 2D image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *x* & *y* are considered to be unnormalized coordinates and must be in the range 0 ... image width – 1, and 0 … image height – 1. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to CL_SNORM_INT8, CL_UNORM_INT8, CL_SNORM_INT16, CL_UNORM_INT16,

	CL_HALF_FLOAT or CL_FLOAT. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** 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. **write_imageui** 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. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*) coordinate values that are not in the range (0 … image width – 1, 0 … image height – 1), respectively, is undefined.
float4 **read_imagef** (image3d_t *image*, sampler_t *sampler*, int4 *coord*) float4 **read_imagef** (image3d_t *image*, sampler_t *sampler*, float4 *coord*)	Use the coordinate *(coord.x*, *coord.y*, *coord.z)* to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **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. **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. The **read_imagef** calls that take integer coordinates

	must use a sampler with filter mode set to CLK_FILTER_NEAREST, normalized coordinates set to CLK_NORMALIZED_COORDS_FALSE and addressing mode set to CLK_ADDRESS_CLAMP_TO_EDGE, CLK_ADDRESS_CLAMP or CLK_ADDRESS_NONE; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description are undefined.
int4 **read_imagei** (image3d_t *image*, sampler_t *sampler*, int4 *coord*) int4 **read_imagei** (image3d_t *image*, sampler_t *sampler*, float4 *coord*) uint4 **read_imageui** (image3d_t *image*, sampler_t *sampler*, int4 *coord*) uint4 **read_imageui** (image3d_t *image*, sampler_t *sampler*, float4 *coord*)	Use the coordinate *(coord.x*, *coord.y*, *coord.z)* to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** 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_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** 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_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to CLK_FILTER_NEAREST; otherwise the values returned are undefined.

	Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to CLK_NORMALIZED_COORDS_FALSE and addressing mode set to CLK_ADDRESS_CLAMP_TO_EDGE, CLK_ADDRESS_CLAMP or CLK_ADDRESS_NONE; otherwise the values returned are undefined.
int **get_image_width** (image2d_t *image*) int **get_image_width** (image3d_t *image*)	Return the image width in pixels.
int **get_image_height** (image2d_t *image*) int **get_image_height** (image3d_t *image*)	Return the image height in pixels.
int **get_image_depth** (image3d_t *image*)	Return the image depth in pixels.
int **get_image_channel_data_type** (image2d_t *image*) int **get_image_channel_data_type** (image3d_t *image*)	Return the channel data type. Valid values are: CLK_SNORM_INT8 CLK_SNORM_INT16 CLK_UNORM_INT8 CLK_UNORM_INT16 CLK_UNORM_SHORT_565 CLK_UNORM_SHORT_555 CLK_UNORM_SHORT_101010 CLK_SIGNED_INT8 CLK_SIGNED_INT16 CLK_SIGNED_INT32 CLK_UNSIGNED_INT8 CLK_UNSIGNED_INT16 CLK_UNSIGNED_INT32 CLK_HALF_FLOAT CLK_FLOAT
int **get_image_channel_order** (image2d_t *image*) int **get_image_channel_order** (image3d_t *image*)	Return the image channel order. Valid values are: CLK_A CLK_R CLK_Rx CLK_RG CLK_RGx CLK_RA CLK_RGB CLK_RGBx CLK_RGBA CLK_ARGB CLK_BGRA CLK_INTENSITY CLK_LUMINANCE

int2 **get_image_dim** (image2d_t *image*)	Return the 2D image width and height as an `int2` type. The width is returned in the *x* component, and the height in the *y* component.
int4 **get_image_dim** (image3d_t *image*)	Return the 3D image width, height, and depth as an `int4` type. The width is returned in the *x* component, height in the *y* component, depth in the *z* component and the *w* component is 0.

Table 6.22 *Built-in Image Read and Write Functions*

The values returned by **get_image_channel_data_type** and **get_image_channel_order** as specified in *table 6.22* with the CLK_ prefixes correspond to the CL_ prefixes used to describe the image channel order and data type in *tables 5.4* and *5.5*. For example, both CL_UNORM_INT8 and CLK_UNORM_INT8 refer to an image channel data type that is an unnormalized unsigned 8-bit integer.

The following table describes the mapping of the number of channels of an image element to the appropriate components in the float4, int4 or uint4 vector data type for the color values returned by **read_image{f|i|ui}** or supplied to **write_image{f|i|ui}**. The unmapped components will be set to `0.0` for red, green and blue channels and will be set to `1.0` for the alpha channel.

Channel Order	**float4, int4 or uint4 components of channel data**
`CL_R, CL_Rx`	`(r, 0.0, 0.0, 1.0)`
`CL_A`	`(0.0, 0.0, 0.0, a)`
`CL_RG, CL_RGx`	`(r, g, 0.0, 1.0)`
`CL_RA`	`(r, 0.0, 0.0, a)`
`CL_RGB, CL_RGBx`	`(r, g, b, 1.0)`
`CL_RGBA, CL_BGRA, CL_ARGB`	`(r, g, b, a)`
`CL_INTENSITY`	`(I, I, I, I)`
`CL_LUMINANCE`	`(L, L, L, 1.0)`

NOTE: A kernel that uses a sampler with the CL_ADDRESS_CLAMP addressing mode with multiple images may result in additional samplers being used internally by an implementation. If the same sampler is used with multiple images called via **read_image{f | i | ui}**, then it is possible that an implementation may need to allocate an additional sampler to handle the different border color values that may be needed depending on the image formats being used. These implementation allocated samplers will count against the maximum sampler values supported by the device and given by CL_DEVICE_MAX_SAMPLERS. Enqueuing a kernel that requires more samplers than the implementation can support will result in a CL_OUT_OF_RESOURCES error being returned.

7. OpenCL Numerical Compliance

This section describes features of the C99 and IEEE 754 standards that must be supported by all OpenCL compliant devices.

This section describes the functionality that must be supported by all OpenCL devices for single precision floating-point numbers. Currently, only single precision floating-point is a requirement. Double precision floating-point is an optional extension.

7.1 Rounding Modes

Floating-point calculations may be carried out internally with extra precision and then rounded to fit into the destination type. IEEE 754 defines four possible rounding modes:

- Round to nearest even
- Round toward + ∞
- Round toward - ∞
- Round toward zero

Round to nearest even is currently the only rounding mode required by the OpenCL specification and is therefore the default rounding mode. In addition, only static selection of rounding mode is supported. Dynamically reconfiguring the rounding modes as specified by the IEEE 754 spec is unsupported.

7.2 INF, NaN and Denormalized Numbers

INF and NaNs must be supported. Support for signaling NaNs is not required.

Support for denormalized numbers with single precision floating-point is optional. Denormalized single precision floating-point numbers passed as input or produced as the output of single precision floating-point operations such as add, sub, mul, divide, and the functions defined in *sections 6.14.2* (math functions), *6.14.4* (common functions) and *6.14.5* (geometric functions) may be flushed to zero.

7.3 Floating-Point Exceptions

Floating-point exceptions are disabled in OpenCL. The result of a floating-point exception must match the IEEE 754 spec for the exceptions not enabled case. Whether and when the implementation sets floating-point flags or raises floating-point exceptions is implementation-defined. This standard provides no method for querying, clearing or setting floating-point flags or trapping raised exceptions. Due to non-performance, non-portability of trap mechanisms and the impracticality of servicing precise exceptions in a vector context (especially on heterogeneous hardware), such features are discouraged.

Implementations that nevertheless support such operations through an extension to the standard shall initialize with all exception flags cleared and the exception masks set so that exceptions raised by arithmetic operations do not trigger a trap to be taken. If the underlying work is reused by the implementation, the implementation is however not responsible for reclearing the flags or resetting exception masks to default values before entering the kernel. That is to say that kernels that do not inspect flags or enable traps are licensed to expect that their arithmetic will not trigger a trap. Those kernels that do examine flags or enable traps are responsible for clearing flag state and disabling all traps before returning control to the implementation. Whether or when the underlying work-item (and accompanying global floating-point state if any) is reused is implementation-defined.

The expressions **math_errorhandling** and **MATH_ERREXCEPT** are reserved for use by this standard, but not defined. Implementations that extend this specification with support for floating-point exceptions shall define **math_errorhandling** and **MATH_ERREXCEPT** per ISO / IEC 9899 : TC2.

7.4 Relative Error as ULPs

In this section we discuss the maximum relative error defined as `ulp` (units in the last place). Addition, subtraction, multiplication, fused multiply-add and conversion between integer and a single precision floating-point format are IEEE 754 compliant and are therefore correctly rounded. Conversion between floating-point formats and explicit conversions specified in *section 6.2.3* must be correctly rounded.

The ULP is defined as follows:

> `If x is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then ulp(x) = |b - a|, otherwise ulp(x) is the distance between the two non-equal finite floating-point numbers nearest x. Moreover, ulp(NaN) is NaN.`

Attribution: This definition was taken with consent from Jean-Michel Muller with slight clarification for behavior at zero. Refer to ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5504.pdf.

Table 7.1[54] describes the minimum accuracy of single precision floating-point arithmetic operations given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

Function	Min Accuracy - ULP values[55]
$x + y$	Correctly rounded
$x - y$	Correctly rounded
$x * y$	Correctly rounded
1.0 $/ x$	<= 2.5 ulp
x / y	<= 2.5 ulp
acos	<= 4 ulp
acospi	<= 5 ulp
asin	<= 4 ulp
asinpi	<= 5 ulp
atan	<= 5 ulp
atan2	<= 6 ulp
atanpi	<= 5 ulp
atan2pi	<= 6 ulp
acosh	<= 4 ulp
asinh	<= 4 ulp
atanh	<= 5 ulp
cbrt	<= 2 ulp
ceil	Correctly rounded
copysign	0 ulp
cos	<= 4 ulp
cosh	<= 4 ulp
cospi	<= 4 ulp
erfc	<= 16 ulp
erf	<= 16 ulp
exp	<= 3 ulp
exp2	<= 3 ulp
exp10	<= 3 ulp
expm1	<= 3 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded

---

[54] The ULP values for built-in math functions **lgamma** and **lgamma_r** is currently undefined.

[55] 0 ulp is used for math functions that do not require rounding.

fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp
hypot	<= 4 ulp
ilogb	0 ulp
ldexp	Correctly rounded
log	<= 3 ulp
log2	<= 3 ulp
log10	<= 3 ulp
log1p	<= 2 ulp
logb	0 ulp
mad	Any value allowed (infinite ulp)
maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	<= 16 ulp
pown(x, y)	<= 16 ulp
powr(x, y)	<= 16 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	<= 16 ulp
round	Correctly rounded
rsqrt	<= 2 ulp
sin	<= 4 ulp
sincos	<= 4 ulp for sine and cosine values
sinh	<= 4 ulp
sinpi	<= 4 ulp
sqrt	<= 3 ulp
tan	<= 5 ulp
tanh	<= 5 ulp
tanpi	<= 6 ulp
tgamma	<= 16 ulp
trunc	Correctly rounded
half_cos	<= 8192 ulp
half_divide	<= 8192 ulp
half_exp	<= 8192 ulp
half_exp2	<= 8192 ulp
half_exp10	<= 8192 ulp

half_log	<= 8192 ulp
half_log2	<= 8192 ulp
half_log10	<= 8192 ulp
half_powr	<= 8192 ulp
half_recip	<= 8192 ulp
half_rsqrt	<= 8192 ulp
half_sin	<= 8192 ulp
half_sqrt	<= 8192 ulp
half_tan	<= 8192 ulp
native_cos	Implementation-defined
native_divide	Implementation-defined
native_exp	Implementation-defined
native_exp2	Implementation-defined
native_exp10	Implementation-defined
native_log	Implementation-defined
native_log2	Implementation-defined
native_log10	Implementation-defined
native_powr	Implementation-defined
native_recip	Implementation-defined
native_rsqrt	Implementation-defined
native_sin	Implementation-defined
native_sqrt	Implementation-defined
native_tan	Implementation-defined

Table 7.1 *ULP values for single precision built-in math functions*

7.5 Edge Case Behavior

The edge case behavior of the math functions (*section 6.11.2*) shall conform to sections F.9 and G.6 of ISO/IEC 9899:TC 2 (commonly known as C99, TC2), except where noted below in *section 7.5.1*.

7.5.1 Additional Requirements Beyond C99 TC2

Functions that return a NaN with more than one NaN operand shall return one of the NaN operands. Functions that return a NaN operand may silence the NaN if it is a signaling NaN. A non-signaling NaN shall be converted to a non-signaling NaN. A signaling NaN shall be converted to a NaN, and should be converted to a non-signaling NaN. How the rest of the NaN payload bits or the sign of NaN is converted is undefined.

half_<funcname> functions behave identically to the function of the same name without the **half_** prefix. They must conform to the same edge case requirements (see sections F.9 and G.6 of C99, TC2). For other cases, except where otherwise noted, these single precision functions are permitted to have up to 8192 ulps of error (as measured in the single precision result), although better accuracy is encouraged.

The usual allowances for rounding error (*section 7.4*) or flushing behavior (*section 7.5.3*) shall not apply for those values for which *section F.9* of C99, TC2, or *sections 7.5.1* and *7.5.3* below (and similar sections for other floating-point precisions) prescribe a result (e.g. **`ceil`** `( -1 < x < 0 )` returns `-0`). Those values shall produce exactly the prescribed answers, and no other. Where the ± symbol is used, the sign shall be preserved. For example, `sin(±0) = ±0` shall be interpreted to mean `sin(+0)` is `+0` and `sin(-0)` is `-0`.

acospi (1) = +0.
acospi (x) returns a NaN for $| x | > 1$.

asinpi (±0) = ±0.
asinpi (x) returns a NaN for $| x | > 1$.

atanpi (±0) = ±0.
atanpi (±∞) = ±0.5.

atan2pi (±0, -0) = ±1.
atan2pi (±0, +0) = ± 0.
atan2pi (±0, x) returns ± 1 for $x < 0$.
atan2pi (±0, x) returns ± 0 for $x > 0$.
atan2pi (y, ±0) returns -0.5 for $y < 0$.
atan2pi (y, ±0) returns 0.5 for $y > 0$.
atan2pi (±y, -∞) returns ± 1 for finite $y > 0$.
atan2pi (±y, +∞) returns ± 0 for finite $y > 0$.
atan2pi (±∞, x) returns ± 0.5 for finite x.
atan2pi (±∞, -∞) returns ±0.75.
atan2pi (±∞, +∞) returns ±0.25.

ceil ($-1 < x < 0$) returns -0.

cospi (±0) returns 1
cospi ($n + 0.5$) is +0 for any integer n where $n + 0.5$ is representable.
cospi (±∞) returns a NaN.

exp10 (±0) returns 1.
exp10 (-∞) returns +0.
exp10 (+∞) returns +∞.

distance (x, y) calculates the distance from x to y without overflow or extraordinary precision loss due to underflow.

fdim (any, NaN) returns NaN.
fdim (NaN, any) returns NaN.

fmod (±0, NaN) returns NaN.

frexp (±∞, *exp*) returns ±∞ and stores 0 in *exp*.
frexp (NaN, *exp*) returns the NaN and stores 0 in *exp*.

fract (x, *iptr*) shall not return a value greater than or equal to 1.0, and shall not return a value less than 0.
fract (+0, *iptr*) returns +0 and +0 in iptr.
fract (-0, *iptr*) returns -0 and -0 in iptr.
fract (+inf, *iptr*) returns +0 and +inf in *iptr*.
fract (-inf, *iptr*) returns -0 and -inf in *iptr*.
fract (NaN, iptr) returns the NaN and NaN in *iptr*.

length calculates the length of a vector without overflow or extraordinary precision loss due to underflow.

nextafter (-0, $y > 0$) returns smallest positive denormal value.
nextafter (+0, $y < 0$) returns smallest negative denormal value.

normalize shall reduce the vector to unit length, pointing in the same direction without overflow or extraordinary precision loss due to underflow.
normalize (v) returns v if all elements of v are zero.
normalize (v) returns a vector full of NaNs if any element is a NaN.

normalize (v) for which any element in v is infinite shall proceed as if the elements in v were replaced as follows:

```
for( i = 0; i < sizeof(v) / sizeof(v[0] ); i++ )
        v[i] = isinf(v[i] )  ?  copysign(1.0, v[i]) : 0.0 * v [i];
```

pow (±0, -∞) returns +∞

pown (x, 0) is 1 for any x, even zero, NaN or infinity.
pown (±0, n) is ±∞ for odd $n < 0$.
pown (±0, n) is +∞ for even $n < 0$.
pown (±0, n) is +0 for even $n > 0$.
pown (±0, n) is ±0 for odd $n > 0$.

powr (x, ±0) is 1 for finite $x > 0$.
powr (±0, y) is +∞ for finite $y < 0$.
powr (±0, -∞) is +∞.
powr (±0, y) is +0 for $y > 0$.

powr ($+1, y$) is 1 for finite y.
powr (x, y) returns NaN for $x < 0$.
powr ($\pm0, \pm0$) returns NaN.
powr ($+\infty, \pm0$) returns NaN.
powr ($+1, \pm\infty$) returns NaN.
powr (x, NaN) returns the NaN for $x >= 0$.
powr (NaN, y) returns the NaN.

rint ($-0.5 <= x < 0$) returns -0.

remquo (x, y, &*quo*) returns a NaN and 0 in *quo* if x is $\pm\infty$, or if y is 0 and the other argument is non-NaN or if either argument is a NaN.

rootn (±0, n) is $\pm\infty$ for odd $n < 0$.
rootn (±0, n) is $+\infty$ for even $n < 0$.
rootn (±0, n) is +0 for even $n > 0$.
rootn (±0, n) is ±0 for odd $n > 0$.
rootn (x, n) returns a NaN for $x < 0$ and n is even.
rootn (x, 0) returns a NaN.

round ($-0.5 < x < 0$) returns -0.

sinpi (±0) returns ±0.
sinpi ($+n$) returns +0 for positive integers n.
sinpi ($-n$) returns -0 for negative integers n.
sinpi ($\pm\infty$) returns a NaN.

tanpi (±0) returns ±0.
tanpi ($\pm\infty$) returns a NaN.
tanpi (n) is **copysign**(0.0, n) for even integers n.
tanpi (n) is **copysign**(0.0, - n) for odd integers n.
tanpi ($n + 0.5$) for even integer n is $+\infty$ where $n + 0.5$ is representable.
tanpi ($n + 0.5$) for odd integer n is $-\infty$ where $n + 0.5$ is representable.

trunc ($-1 < x < 0$) returns -0.

7.5.2 Changes to C99 TC2 Behavior

modf behaves as though implemented by:

```
gentype modf ( gentype value, gentype *iptr )
{
        *iptr = trunc( value );
        return copysign( isinf( value ) ? 0.0 : value – *iptr, value );
}
```

rint always rounds according to round to nearest even rounding mode even if the caller is in some other rounding mode.

7.5.3 Edge Case Behavior in Flush To Zero Mode

If denormals are flushed to zero, then a function may return one of four results:

1. Any conforming result for non-flush-to-zero mode
2. If the result given by 1. is a sub-normal before rounding, it may be flushed to zero
3. Any non-flushed conforming result for the function if one or more of its sub-normal operands are flushed to zero.
4. If the result of 3. is a sub-normal before rounding, the result may be flushed to zero.

In each of the above cases, if an operand or result is flushed to zero, the sign of the zero is undefined.

If subnormals are flushed to zero, a device may choose to conform to the following edge cases for **nextafter** instead of those listed in *section 7.5.1*:

nextafter (+smallest normal, *y* < +smallest normal) = +0.
nextafter (-smallest normal, *y* > -smallest normal) = -0.
nextafter (-0, *y* > 0) returns smallest positive normal value.
nextafter (+0, *y* < 0) returns smallest negative normal value.

For clarity, subnormals or denormals are defined to be the set of representable numbers in the range `0 < x < TYPE_MIN` and `-TYPE_MIN < x < -0`. They do not include `±0`. A non-zero number is said to be sub-normal before rounding if after normalization, its radix-2 exponent is less than (`TYPE_MIN_EXP - 1`).[56]

---

[56] Here TYPE_MIN and TYPE_MIN_EXP should be substituted by constants appropriate to the floating-point type under consideration, such as FLT_MIN and FLT_MIN_EXP for float.

8. Image Addressing and Filtering

Let w_t, h_t and d_t be the width, height and depth of the image in pixels. Let `coord.xy` also referred to as (`s, t`) or `coord.xyz` also referred to as (`s, t, r`) be the coordinates specified to **read_image{f|i|ui}**. The sampler specified in **read_image{f|i|ui}** is used to determine how to sample the image and return an appropriate color.

8.1 Normalized Coordinates

This affects the interpretation of image coordinates. If image coordinates specified to **read_image{f|i|ui}** are normalized (as specified in the sampler), the `s, t` and `r` coordinate values are multiplied by w_t, h_t and d_t respectively to generate the unnormalized coordinate value.

Let (`u, v, w`) represent the unnormalized image coordinate floating-point values.

8.2 Addressing Mode and Filtering

We first describe how the addressing and filtering modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is not CLK_ADDRESS_REPEAT nor CLK_ADDRESS_MIRRORED_REPEAT.

After generating the image coordinate (`u, v, w`) we apply the appropriate addressing and filtering mode to generate the appropriate sample locations to read from the image.

If values in (`u, v, w`) are INF or NaN, the behavior of **read_image{f|i|ui}**is undefined.

Filter Mode = CLK_FILTER_NEAREST

When filter mode is CLK_FILTER_NEAREST, the image element in the image that is nearest (in Manhattan distance) to that specified by (`u, v, w`) is obtained. This means the image element at location (`i, j, k`) becomes the image element value, where

```
i    =    address_mode((int)floor(u))
j    =    address_mode((int)floor(v))
k    =    address_mode((int)floor(w))
```

For a 3-dimensional image, the image element at location (`i`, `j`, `k`) becomes the color value. For a 2-dimensional image, the image element at location (`i`, `j`) becomes the color value.

Table 8.1 describes the `address_mode` function.

Addressing Mode	Result of address_mode(coord)
CLK_ADDRESS_CLAMP_TO_EDGE	`clamp (coord, 0, size - 1)`
CLK_ADDRESS_CLAMP	`clamp (coord, -1, size)`
CLK_ADDRESS_NONE	`coord`

Table 8.1 *Addressing modes to generate texel location.*

The `size` term in table 8.1 is w_t for `u`, h_t for `v` and d_t for `w`.

The `clamp` function used in *table 8.1* is defined as:

```
clamp(a, b, c) = return (a < b) ? b : ((a > c) ? c : a)
```

If the selected texel location (`i, j, k`) refers to a location outside the image, the border color is used as the color value for this texel.

Filter Mode = CLK_FILTER_LINEAR

When filter mode is CLK_FILTER_LINEAR, a `2 x 2` square of image elements for a 2D image or a `2 x 2 x 2` cube of image elements for a 3D texture is selected. This `2 x 2` square or `2 x 2 x 2` cube is obtained as follows.

Let

```
i0 =  address_mode((int)floor(u - 0.5))
j0 =  address_mode((int)floor(v - 0.5))
k0 =  address_mode((int)floor(w - 0.5))
i1 =  address_mode((int)floor(u - 0.5) + 1)
j1 =  address_mode((int)floor(v - 0.5) + 1)
k1 =  address_mode((int)floor(w - 0.5) + 1)
a  =  frac(u - 0.5)
b  =  frac(v - 0.5)
c  =  frac(w - 0.5)
```

where `frac(x)` denotes the fractional part of `x` and is computed as `x - floor(x)`.

For a three-dimensional image, the image element value is found as

$$\begin{aligned} T = {} & (1 - a) * (1 - b) * (1 - c) * T_{i0j0k0} \\ & + a * (1 - b) * (1 - c) * T_{i1j0k0} \\ & + (1 - a) * b * (1 - c) * T_{i0j1k0} \\ & + a * b * (1 - c) * T_{i1j1k0} \end{aligned}$$

$$
\begin{aligned}
&+ (1 - a) * (1 - b) * c * T_{i0j0k1} \\
&+ a * (1 - b) * c * T_{i1j0k1} \\
&+ (1 - a) * b * c * T_{i0j1k1} \\
&+ a * b * c * T_{i1j1k1}
\end{aligned}
$$

where T_{ijk} is the image element at location (`i, j, k`) in the three-dimensional image.

For a two-dimensional image, the image element value is found as

$$
\begin{aligned}
T = {} & (1 - a) * (1 - b) * T_{i0j0} \\
&+ a * (1 - b) * T_{i1j0} \\
&+ (1 - a) * b * T_{i0j1} \\
&+ a * b * T_{i1j1}
\end{aligned}
$$

where T_{ij} is the image element at location (`i, j`) in the two-dimensional image.

If any of the selected T_{ijk} or T_{ij} in the above equations refers to a location outside the image, the border color is used as the color value for T_{ijk} or T_{ij}.

We now discuss how the addressing and filtering modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is CLK_ADDRESS_REPEAT.

If values in (`s, t, r`) are INF or NaN, the behavior of the built-in image read functions is undefined.

Filter Mode = CLK_FILTER_NEAREST

When filter mode is CLK_FILTER_NEAREST, the image element at location (`i, j, k`) becomes the image element value, with `i, j and k` computed as

```
u = (s – floor(s)) * $w_t$
i = (int)floor(u)
if (i > $w_t$ – 1)
    i = i – $w_t$

v = (t – floor(t)) * $h_t$
j = (int)floor(v)
if (j > $h_t$ – 1)
    j = j – $h_t$

w = (r – floor(r)) * $d_t$
k = (int)floor(w)
if (k > $d_t$ – 1)
```

```
                k = k - $d_t$
```

For a 3-dimensional image, the image element at location (`i`, `j`, `k`) becomes the color value. For a 2-dimensional image, the image element at location (`i`, `j`) becomes the color value.

Filter Mode = CLK_FILTER_LINEAR

When filter mode is CLK_FILTER_LINEAR, a `2 x 2` square of image elements for a 2D image or a `2 x 2 x 2` cube of image elements for a 3D image is selected. This `2 x 2` square or `2 x 2 x 2` cube is obtained as follows.

Let

```
u = (s - floor(s)) * $w_t$
i0 = (int)floor(u - 0.5)
i1 = i0 + 1
if (i0 < 0)
    i0 = $w_t$ + i0
if (i1 > $w_t$ - 1)
    i1 = i1 - $w_t$

v = (t - floor(t)) * $h_t$
j0 = (int)floor(v - 0.5)
j1 = j0 + 1
if (j0 < 0)
    j0 = $h_t$ + j0
if (j1 > $h_t$ - 1)
    j1 = j1 - $h_t$

w = (r - floor(r)) * $d_t$
k0 = (int)floor(w - 0.5)
k1 = k0 + 1
if (k0 < 0)
    k0 = $d_t$ + k0
if (k1 > $d_t$ - 1)
    k1 = k1 - $d_t$

a  =  frac(u - 0.5)
b  =  frac(v - 0.5)
c  =  frac(w - 0.5)
```

where `frac(x)` denotes the fractional part of `x` and is computed as `x - floor(x)`.

For a three-dimensional image, the image element value is found as

```
T = (1 - a) * (1 - b) * (1 - c) * $T_{i0j0k0}$
```

```
         + a * (1 - b) * (1 - c) * $T_{i1j0k0}$
         + (1 - a) * b * (1 - c) * $T_{i0j1k0}$
         + a * b * (1 - c) * $T_{i1j1k0}$
         + (1 - a) * (1 - b) * c * $T_{i0j0k1}$
         + a * (1 - b) * c * $T_{i1j0k1}$
         + (1 - a) * b * c * $T_{i0j1k1}$
         + a * b * c * $T_{i1j1k1}$
```

where T_{ijk} is the image element at location (`i, j, k`) in the three-dimensional image.

For a two-dimensional image, the image element value is found as

```
T = (1 - a) * (1 - b) * $T_{i0j0}$
       + a * (1 - b) * $T_{i1j0}$
       + (1 - a) * b * $T_{i0j1}$
       + a * b * $T_{i1j1}$
```

where T_{ij} is the image element at location (`i, j`) in the two-dimensional image.

We now discuss how the addressing and filtering modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is CLK_ADDRESS_MIRRORED_REPEAT. The CLK_ADDRESS_MIRRORED_REPEAT addressing mode causes the image to be read as if it is tiled at every integer seam with the interpretation of the image data flipped at each integer crossing. For example, the (`s, t, r`) coordinates between 2 and 3 are addressed into the image as coordinates from 1 down to 0. If values in (`s, t, r`) are INF or NaN, the behavior of the built-in image read functions is undefined.

Filter Mode = CLK_FILTER_NEAREST

When filter mode is CLK_FILTER_NEAREST, the image element at location (`i, j, k`) becomes the image element value, with `i, j and k` computed as

```
s’ = 2.0f * rint(0.5f * s)
s’ = fabs(s - s’)
u = s’ * $w_t$
i = (int)floor(u)
i = min(i, $w_t$ - 1)

t’ = 2.0f * rint(0.5f * t)
t’ = fabs(t - t’)
v = t’ * $h_t$
j = (int)floor(v)
j = min(j, $h_t$ - 1)
```

```
r′ = 2.0f * rint(0.5f * r)
r′ = fabs(r - r′)
w = r′ * $d_t$
k = (int)floor(w)
k = min(k, $d_t$ - 1)
```

For a 3-dimensional image, the image element at location (`i`, `j`, `k`) becomes the color value. For a 2-dimensional image, the image element at location (`i`, `j`) becomes the color value.

Filter Mode = CLK_FILTER_LINEAR

When filter mode is CLK_FILTER_LINEAR, a `2 x 2` square of image elements for a 2D image or a `2 x 2 x 2` cube of image elements for a 3D image is selected. This `2 x 2` square or `2 x 2 x 2` cube is obtained as follows.

Let

```
s′ = 2.0f * rint(0.5f * s)
s′ = fabs(s - s′)
u = s′ * $w_t$
i0 = (int)floor(u - 0.5f)
i1 = i0 + 1
i0 = max(i0, 0)
i1 = min(i1, $w_t$ - 1)

t′ = 2.0f * rint(0.5f * t)
t′ = fabs(t - t′)
v = t′ * $h_t$
j0 = (int)floor(v - 0.5f)
j1 = j0 + 1
j0 = max(j0, 0)
j1 = min(j1, $h_t$ - 1)

r′ = 2.0f * rint(0.5f * r)
r′ = fabs(r - r′)
w = r′ * $d_t$
k0 = (int)floor(w - 0.5f)
k1 = k0 + 1
k0 = max(k0, 0)
k1 = min(k1, $d_t$ - 1)

a  =  frac(u - 0.5)
b  =  frac(v - 0.5)
c  =  frac(w - 0.5)
```

where `frac(x)` denotes the fractional part of `x` and is computed as `x - floor(x)`.

For a three-dimensional image, the image element value is found as

$$
\begin{aligned}
T = {} & (1 - a) * (1 - b) * (1 - c) * T_{i0j0k0} \\
& + a * (1 - b) * (1 - c) * T_{i1j0k0} \\
& + (1 - a) * b * (1 - c) * T_{i0j1k0} \\
& + a * b * (1 - c) * T_{i1j1k0} \\
& + (1 - a) * (1 - b) * c * T_{i0j0k1} \\
& + a * (1 - b) * c * T_{i1j0k1} \\
& + (1 - a) * b * c * T_{i0j1k1} \\
& + a * b * c * T_{i1j1k1}
\end{aligned}
$$

where T_{ijk} is the image element at location (`i, j, k`) in the three-dimensional image.

For a two-dimensional image, the image element value is found as

$$
\begin{aligned}
T = {} & (1 - a) * (1 - b) * T_{i0j0} \\
& + a * (1 - b) * T_{i1j0} \\
& + (1 - a) * b * T_{i0j1} \\
& + a * b * T_{i1j1}
\end{aligned}
$$

where T_{ij} is the image element at location (`i, j`) in the two-dimensional image.

NOTE

If the sampler is specified as using unnormalized coordinates (floating-point or integer coordinates), filtering mode set to CLK_FILTER_NEAREST *and addressing mode set to one of the following modes -* CLK_ADDRESS_NONE, CLK_ADDRESS_CLAMP_TO_EDGE *or* CLK_ADDRESS_CLAMP*, the location of the image element in the image given by (i, j, k) in section 8.2 will be computed without any loss of precision.*

For all other sampler combinations of normalized or unnormalized coordinates, filtering and addressing modes, the relative error or precision of the addressing mode calculations and the image filtering operation are not defined by this revision of the OpenCL specification. To ensure a minimum precision of image addressing and filtering calculations across any OpenCL device, for these sampler combinations, developers should unnormalize the image coordinate in the kernel and implement the linear filter in the kernel with appropriate calls to ***read_image{f|i|ui}*** *with a sampler that uses unnormalized coordinates, filtering mode set to* CLK_FILTER_NEAREST*, addressing mode set to* CLK_ADDRESS_NONE, CLK_ADDRESS_CLAMP_TO_EDGE *or* CLK_ADDRESS_CLAMP *and finally performing the interpolation of color values read from the image to generate the filtered color value.*

8.3 Conversion Rules

In this section we discuss conversion rules that are applied when reading and writing images in a kernel.

8.3.1 Conversion rules for normalized integer channel data types

In this section we discuss converting normalized integer channel data types to floating-point values and vice-versa.

8.3.1.1 Converting normalized integer channel data types to floating-point values

For images created with image channel data type of CL_UNORM_INT8 and CL_UNORM_INT16, **read_imagef** will convert the channel values from an 8-bit or 16-bit unsigned integer to normalized floating-point values in the range [`0.0f … 1.0`].

For images created with image channel data type of CL_SNORM_INT8 and CL_SNORM_INT16, **read_imagef** will convert the channel values from an 8-bit or 16-bit signed integer to normalized floating-point values in the range [`-1.0 … 1.0`].

These conversions are performed as follows:

CL_UNORM_INT8 (8-bit unsigned integer) → float

normalized float value = `(float)c / 255.0f`

CL_UNORM_INT_101010 (10-bit unsigned integer) → float

normalized float value = `(float)c / 1023.0f`

CL_UNORM_INT16 (16-bit unsigned integer) → float

normalized float value = `(float)c / 65535.0f`

CL_SNORM_INT8 (8-bit signed integer) → float

normalized float value = `max(-1.0f, (float)c / 127.0f)`

CL_SNORM_INT16 (16-bit signed integer) → float

normalized float value = `max(-1.0f, (float)c / 32767.0f)`

The precision of the above conversions is <= 1.5 ulp except for the following cases.

For CL_UNORM_INT8

```
0 must convert to 0.0f and
255 must convert to 1.0f
```

For CL_UNORM_INT_101010

```
0 must convert to 0.0f and
1023 must convert to 1.0f
```

For CL_UNORM_INT16

```
0 must convert to 0.0f and
65535 must convert to 1.0f
```

For CL_SNORM_INT8

```
-128 and -127 must convert to -1.0f,
0 must convert to 0.0f and
127 must convert to 1.0f
```

For CL_SNORM_INT16

```
-32768 and -32767 must convert to -1.0f,
0 must convert to 0.0f and
32767 must convert to 1.0f
```

8.3.1.2 Converting floating-point values to normalized integer channel data types

For images created with image channel data type of CL_UNORM_INT8 and CL_UNORM_INT16, **write_imagef** will convert the floating-point color value to an 8-bit or 16-bit unsigned integer.

For images created with image channel data type of CL_SNORM_INT8 and CL_SNORM_INT16, **write_imagef** will convert the floating-point color value to an 8-bit or 16-bit signed integer.

The preferred method for how conversions from floating-point values to normalized integer values are performed is as follows:

float → CL_UNORM_INT8 (8-bit unsigned integer)

```
convert_uchar_sat_rte(f * 255.0f)
```

float → CL_UNORM_INT_101010 (10-bit unsigned integer)

```
min(convert_ushort_sat_rte(f * 1023.0f), 0x3ff)
```

float → CL_UNORM_INT16 (16-bit unsigned integer)

```
convert_ushort_sat_rte(f * 65535.0f)
```

float → CL_SNORM_INT8 (8-bit signed integer)

```
convert_char_sat_rte(f * 127.0f)
```

float → CL_SNORM_INT16 (16-bit signed integer)

```
convert_short_sat_rte(f * 32767.0f)
```

Please refer to *section 6.2.3.3* for out-of-range behavior and saturated conversions rules.

OpenCL implementations may choose to approximate the rounding mode used in the conversions described above. If a rounding mode other than round to nearest even (`_rte`) is used, the absolute error of the implementation dependant rounding mode vs. the result produced by the round to nearest even rounding mode must be <= 0.6.

float → CL_UNORM_INT8 (8-bit unsigned integer)

Let $f_{preferred}$ = `convert_uchar_sat_rte(f * 255.0f)`
Let f_{approx} =
`convert_uchar_sat_<impl-rounding-mode>(f * 255.0f)`

fabs($f_{preferred}$ – f_{approx}) must be <= 0.6

float → CL_UNORM_INT_101010 (10-bit unsigned integer)

Let $f_{preferred}$ = `convert_ushort_sat_rte(f * 1023.0f)`
Let f_{approx} =
`convert_ushort_sat_<impl-rounding-mode>(f * 1023.0f)`

fabs($f_{preferred}$ – f_{approx}) must be <= 0.6

float → CL_UNORM_INT16 (16-bit unsigned integer)

Let $f_{preferred}$ = `convert_ushort_sat_rte(f * 65535.0f)`
Let f_{approx} =
`convert_ushort_sat_<impl-rounding-mode>(f * 65535.0f)`

```
fabs(f_preferred - f_approx) must be <= 0.6
```

float → CL_SNORM_INT8 (8-bit signed integer)

```
Let f_preferred = convert_char_sat_rte(f * 127.0f)
Let f_approx =
    convert_char_sat_<impl_rounding_mode>(f * 127.0f)

fabs(f_preferred - f_approx) must be <= 0.6
```

float → CL_SNORM_INT16 (16-bit signed integer)

```
Let f_preferred = convert_short_sat_rte(f * 32767.0f)
Let f_approx =
  convert_short_sat_<impl-rounding-mode>(f * 32767.0f)

fabs(f_preferred - f_approx) must be <= 0.6
```

8.3.2 Conversion rules for half floating-point channel data type

For images created with a channel data type of CL_HALF_FLOAT, the conversions from `half` to `float` are lossless (as described in *section 6.1.1.1*). Conversions from `float` to `half` round the mantissa using the round to nearest even or round to zero rounding mode. Denormalized numbers for the `half` data type which may be generated when converting a `float` to a `half` may be flushed to zero. A `float` NaN must be converted to an appropriate NaN in the `half` type. A `float` INF must be converted to an appropriate INF in the `half` type.

8.3.3 Conversion rules for floating-point channel data type

The following rules apply for reading and writing images created with channel data type of CL_FLOAT.

- NaNs may be converted to a NaN value(s) supported by the device.
- Denorms can be flushed to zero.
- All other values must be preserved.

8.3.4 Conversion rules for signed and unsigned 8-bit, 16-bit and 32-bit integer channel data types

Calls to **read_imagei** with channel data type values of CL_SIGNED_INT8, CL_SIGNED_INT16 and CL_SIGNED_INT32 return the unmodified integer values stored in the image at specified location.

Calls to **read_imageui** with channel data type values of CL_UNSIGNED_INT8, CL_UNSIGNED_INT16 and CL_UNSIGNED_INT32 return the unmodified integer values stored in the image at specified location.

Calls to **write_imagei** will perform one of the following conversions:

32 bit signed integer → 8-bit signed integer

```
convert_char_sat(i)
```

32 bit signed integer → 16-bit signed integer

```
convert_short_sat(i)
```

32 bit signed integer → 32-bit signed integer

```
no converion is performed
```

Calls to **write_imageui** will perform one of the following conversions:

32 bit unsigned integer → 8-bit unsigned integer

```
convert_uchar_sat(i)
```

32 bit unsigned integer → 16-bit unsigned integer

```
convert_ushort_sat(i)
```

32 bit unsigned integer → 32-bit unsigned integer

```
no converion is performed
```

The conversions described in this section must be correctly saturated.

9. Optional Extensions

This section describes the list of optional features supported by the OpenCL specification. Previous sections have discussed features that all implementations must support. The following are optional extensions that 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 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 should continue to export the name strings of promoted extensions in the CL_PLATFORM_EXTENSIONS or CL_DEVICE_EXTENSIONS string, and continue to 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 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*>**.

- 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:

```
#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.

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

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

```
#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 "**`cl_khr_fp64`**" should also add a preprocessor **`#define`** called **`cl_khr_fp64`**. A kernel can now use this preprocessor `#define` to do something like:

```
#ifdef cl_khr_fp64
     // do something using the extension
#else
     // do something else or #error!
#endif
```

9.2 Getting OpenCL API Extension Function Pointers

The function

void* **clGetExtensionFunctionAddress**[57] (const char ***funcname*)

returns the address of the extension function named by *funcname*. 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. A non-NULL return value for **clGetExtensionFunctionAddress** does not guarantee that an extension function is actually supported. 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.

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

---

[57] 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. The behaviorof calling a device extension function on a device not supporting that extension is undefined.

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

```
#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 *clextension_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:

```
#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 Double Precision Floating-Point

Support for double floating-point precision is a requirement for a class of scientific computing algorithms/applications. This class of applications can be enabled by adding support for double precision floating-point as an optional extension.

OpenCL 1.1 supports double precision floating-point as an optional extension. An application that wants to use `double` will need to include the **`#pragma OPENCL EXTENSION cl_khr_fp64 : enable`** directive before any double precision data type is declared in the kernel code.

The list of built-in scalar, and vector data types defined in *tables 6.1*, and *6.2* are extended to include the following:

Type	Description
double	A double precision float.
double2	A 2-component double vector.
double3	A 3-component double vector.
double4	A 4-component double vector.
double8	A 8-component double vector.
double16	A 16-component double vector.

The built-in scalar and vector data types for double are also declared as appropriate types in the OpenCL API (and header files) that can be used by an application. The following table describes the built-in scalar and vector data types for double as defined in the OpenCL C programming language and the corresponding data type available to the application:

Type in OpenCL Language	API type for application
double	**cl_double**
double2	**cl_double2**
double3	**cl_double3**
double4	**cl_double4**
double8	**cl_double8**
double16	**cl_double16**

The double data type must confirm to the IEEE-754 double precision storage format.

The following text is added to *section 6.1.1.1*.

> Conversions from `double` to `half` are correctly rounded. Conversions from `half` to `double` are lossless.

9.3.1 Conversions

The implicit conversion rules specified in *section 6.2.1* now include the `double` scalar and `double`*`n`* vector data types.

The explicit casts described in *section 6.2.2* are extended to take a `double` scalar data type and a `double`*`n`* vector data type.

The explicit conversion functions described in *section 6.2.3* are extended to take a `double` scalar data type and a `double`*`n`* vector data type.

The `as_typen()` function for re-interpreting types as described in *section 6.2.4.2* is extended to allow conversion-free casts between `long`*`n`*, `ulong`*`n`* and `double`*`n`* scalar and vector data types.

9.3.2 Math Functions

The built-in math functions defined in *table 6.8* (also listed below) are extended to include appropriate versions of functions that take `double,` and `double{2|3|4|8|16}` as arguments and return values. `gentype` now also includes `double, double2, double3, double4, double8` and `double16`. For any specific use of a function, the actual type has to be the same for all arguments and the return type.

Function	**Description**
gentype **acos** (gentype)	Arc cosine function.
gentype **acosh** (gentype)	Inverse hyperbolic cosine.
gentype **acospi** (gentype *x*)	Compute **acos** (*x*) / π.
gentype **asin** (gentype)	Arc sine function.
gentype **asinh** (gentype)	Inverse hyperbolic sine.
gentype **asinpi** (gentype *x*)	Compute **asin** (*x*) / π.
gentype **atan** (gentype *y_over_x*)	Arc tangent function.
gentype **atan2** (gentype *y*, gentype *x*)	Arc tangent of *y* / *x*.
gentype **atanh** (gentype)	Hyperbolic arc tangent.
gentype **atanpi** (gentype *x*)	Compute **atan** (*x*) / π.
gentype **atan2pi** (gentype *y*, gentype *x*)	Compute **atan2** (*y*, *x*) / π.
gentype **cbrt** (gentype)	Compute cube-root.
gentype **ceil** (gentype)	Round to integral value using the round to +ve infinity rounding mode.
gentype **copysign** (gentype *x*, gentype *y*)	Returns *x* with its sign changed to match the sign of *y*.
gentype **cos** (gentype)	Compute cosine.
gentype **cosh** (gentype)	Compute hyperbolic consine.

gentype **cospi** (gentype x)	Compute **cos** (πx).
gentype **erfc** (gentype)	Complementary error function.
gentype **erf** (gentype)	Error function encountered in integrating the normal distribution.
gentype **exp** (gentype x)	Compute the base- e exponential of x.
gentype **exp2** (gentype)	Exponential base 2 function.
gentype **exp10** (gentype)	Exponential base 10 function.
gentype **expm1** (gentype x)	Compute e^x- 1.0.
gentype **fabs** (gentype)	Compute absolute value of a floating-point number.
gentype **fdim** (gentype x, gentype y)	$x - y$ if $x > y$, +0 if x is less than or equal to y.
gentype **floor** (gentype)	Round to integral value using the round to –ve infinity rounding mode.
gentype **fma** (gentype a, gentype b, gentype c)	Returns the correctly rounded floating-point representation of the sum of c with the infinitely precise product of a and b. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.
gentype **fmax** (gentype x, gentype y) gentype **fmax** (gentype x, double y)	Returns y if $x < y$, otherwise it returns x. If one argument is a NaN, **fmax()** returns the other argument. If both arguments are NaNs, **fmax()** returns a NaN.
gentype **fmin** (gentype x, gentype y) gentype **fmin** (gentype x, double y)	Returns y if $y < x$, otherwise it returns x. If one argument is a NaN, **fmin()** returns the other argument. If both arguments are NaNs, **fmin()** returns a NaN.
gentype **fmod** (gentype x, gentype y)	Modulus. Returns $x – y$ * **trunc** (x/y) .
gentype **fract** (gentype x, __global gentype *$iptr$) gentype **fract** (gentype x, __local gentype *$iptr$) gentype **fract** (gentype x, __private gentype *$iptr$)	Returns **fmin**(x – **floor** (x), 0x1. fffffffffffffp-1). **floor**(x) is returned in $iptr$.
gentype **frexp** (gentype x, __global int$n$ *exp) gentype **frexp** (gentype x, __local int$n$ *exp) gentype **frexp** (gentype x, __private int$n$ *exp)	Extract mantissa and exponent from x. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of x equals mantissa returned * 2^{exp}.
gentype **hypot** (gentype x, gentype y)	Compute the value of the square root of $x^2+ y^2$ without undue overflow or underflow.
int$n$ **ilogb** (gentype x)	Return the exponent as an integer value.
gentype **ldexp** (gentype x, int$n$ n) gentype **ldexp** (gentype x, int n)	Multiply x by 2 to the power n.
gentype **lgamma** (gentype x) gentype **lgamma_r** (gentype x,	Log gamma function. Returns the natural logarithm of the absolute value of the gamma

__global int*n* **signp*) gentype **lgamma_r** (gentype *x*, __local int*n* **signp*) gentype **lgamma_r** (gentype *x*, __private int*n* **signp*)	function. The sign of the gamma function is returned in the *signp* argument of **lgamma_r**.
gentype **log** (gentype)	Compute natural logarithm.
gentype **log2** (gentype)	Compute a base 2 logarithm.
gentype **log10** (gentype)	Compute a base 10 logarithm.
gentype **log1p** (gentype *x*)	Compute $\log_e(1.0 + x)$.
gentype **logb** (gentype *x*)	Compute the exponent of x, which is the integral part of $\log_r \lvert x \rvert$.
gentype **mad** (gentype *a*, gentype *b*, gentype *c*)	**mad** approximates $a * b + c$. Whether or how the product of $a * b$ is rounded and how supernormal or subnormal intermediate products are handled is not defined. **mad** is intended to be used where speed is preferred over accuracy[58].
gentype **maxmag** (gentype *x*, gentype *y*)	Returns x if $\lvert x \rvert > \lvert y \rvert$, y if $\lvert y \rvert > \lvert x \rvert$, otherwise **fmax**(*x*, *y*).
gentype **minmag** (gentype *x*, gentype *y*)	Returns x if $\lvert x \rvert < \lvert y \rvert$, y if $\lvert y \rvert < \lvert x \rvert$, otherwise **fmin**(*x*, *y*).
gentype **modf** (gentype *x*, __global gentype **iptr*) gentype **modf** (gentype *x*, __local gentype **iptr*) gentype **modf** (gentype *x*, __private gentype **iptr*)	Decompose a floating-point number. The **modf** function breaks the argument x into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by *iptr*.
double*n* **nan** (ulong*n* *nancode*)	Returns a quiet NaN. The *nancode* may be placed in the significand of the resulting NaN.
gentype **nextafter** (gentype *x*, gentype *y*)	Computes the next representable single-precision floating-point value following x in the direction of y. Thus, if y is less than x, **nextafter**() returns the largest representable floating-point number less than x.
gentype **pow** (gentype *x*, gentype *y*)	Compute x to the power y.
gentype **pown** (gentype *x*, int*n* *y*)	Compute x to the power y, where y is an integer.
gentype **powr** (gentype *x*, gentype *y*)	Compute x to the power y, where x is >= 0.
gentype **remainder** (gentype *x*, gentype *y*)	Compute the value r such that $r = x - n*y$, where n is the integer nearest the exact value of x/y. If there are two integers closest to x/y, n shall be the even one. If r is zero, it is given the same sign as x.
gentype **remquo** (gentype *x*, gentype *y*, __global int*n* **quo*)	The **remquo** function computes the value r such that $r = x - n*y$, where n is the integer nearest the exact value of x/y. If there are two integers closest

---

[58] The user is cautioned that for some usages, e.g. **mad**(a, b, -a*b), the definition of **mad**() is loose enough that almost any result is allowed from **mad**() for some values of a and b.

gentype **remquo** (gentype *x*, gentype *y*, __local int*n* **quo*) gentype **remquo** (gentype *x*, gentype *y*, __private int*n* **quo*)	to *x*/*y*, *n* shall be the even one. If *r* is zero, it is given the same sign as *x*. This is the same value that is returned by the **remainder** function. **remquo** also calculates the lower seven bits of the integral quotient *x*/*y*, and gives that value the same sign as *x*/*y*. It stores this signed value in the object pointed to by *quo*.
gentype **rint** (gentype)	Round to integral value (using round to nearest even rounding mode) in floating-point format. Refer to section 7.1 for description of rounding modes.
gentype **rootn** (gentype *x*, int*n y*)	Compute *x* to the power 1/*y*.
gentype **round** (gentype *x*)	Return the integral value nearest to *x* rounding halfway cases away from zero, regardless of the current rounding direction.
gentype **rsqrt** (gentype)	Compute inverse square root.
gentype **sin** (gentype)	Compute sine.
gentype **sincos** (gentype *x*, __global gentype **cosval*) gentype **sincos** (gentype *x*, __local gentype **cosval*) gentype **sincos** (gentype *x*, __private gentype **cosval*)	Compute sine and cosine of x. The computed sine is the return value and computed cosine is returned in *cosval*.
gentype **sinh** (gentype)	Compute hyperbolic sine.
gentype **sinpi** (gentype *x*)	Compute **sin** (π *x*).
gentype **sqrt** (gentype)	Compute square root.
gentype **tan** (gentype)	Compute tangent.
gentype **tanh** (gentype)	Compute hyperbolic tangent.
gentype **tanpi** (gentype *x*)	Compute **tan** (π *x*).
gentype **tgamma** (gentype)	Compute the gamma function.
gentype **trunc** (gentype)	Round to integral value using the round to zero rounding mode.

Table 6.7 *Scalar and Vector Argument Built-in Math Function Table*

In addition, the following symbolic constant will also be available:

HUGE_VAL	- A positive double expression that evaluates to + infinity. Used as an error value returned by the built-in math functions.

The **FP_FAST_FMA** macro indicates whether the **fma()** family of functions are fast compared with direct code for double precision floating-point. If defined, the **FP_FAST_FMA** macro shall indicate that the **fma()** function generally executes about as fast as, or faster than, a multiply and an add of **double** operands

The macro names given in the following list must use the values specified. These constant expressions are suitable for use in #if preprocessing directives.

```
#define DBL_DIG          15
#define DBL_MANT_DIG     53
#define DBL_MAX_10_EXP   +308
#define DBL_MAX_EXP      +1024
#define DBL_MIN_10_EXP   -307
#define DBL_MIN_EXP      -1021
#define DBL_MAX          0x1.fffffffffffffp1023
#define DBL_MIN          0x1.0p-1022
#define DBL_EPSILON      0x1.0p-52
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	Macro for application
DBL_DIG	**CL_DBL_DIG**
DBL_MANT_DIG	**CL_DBL_MANT_DIG**
DBL_MAX_10_EXP	**CL_DBL_MAX_10_EXP**
DBL_MAX_EXP	**CL_DBL_MAX_EXP**
DBL_MIN_10_EXP	**CL_DBL_MIN_10_EXP**
DBL_MIN_EXP	**CL_DBL_MIN_EXP**
DBL_MAX	**CL_DBL_MAX**
DBL_MIN	**CL_DBL_MIN**
DBL_EPSILSON	**CL_DBL_EPSILON**

The following constants are also available. They are of type `double` and are accurate within the precision of the `double` type.

Constant	Description
M_E	Value of e
M_LOG2E	Value of $\log_2 e$
M_LOG10E	Value of $\log_{10} e$
M_LN2	Value of $\log_e 2$
M_LN10	Value of $\log_e 10$
M_PI	Value of π
M_PI_2	Value of $\pi / 2$
M_PI_4	Value of $\pi / 4$
M_1_PI	Value of $1 / \pi$
M_2_PI	Value of $2 / \pi$
M_2_SQRTPI	Value of $2 / \sqrt{\pi}$
M_SQRT2	Value of $\sqrt{2}$
M_SQRT1_2	Value of $1 / \sqrt{2}$

9.3.3 Common Functions[59]

The built-in common functions defined in *table 6.12* (also listed below) are extended to include appropriate versions of functions that take `double`, and `double{2|3|4|8|16}` as arguments and return values. `gentype` now also includes `double, double2, double3, double4, double8` and `double16`. These are described below.

Function	**Description**
gentype **clamp** (gentype *x*, gentype *minval*, gentype *maxval*) gentype **clamp** (gentype *x*, double *minval*, double *maxval*)	Returns **min**(**max**(*x*, *minval*), *maxval*) . Results are undefined in *minval* > *maxval.*
gentype **degrees** (gentype *radians*)	Converts *radians* to degrees, i.e. (180 / π) * *radians*.
gentype **max** (gentype *x*, gentype *y*) gentype **max** (gentype *x*, double *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*. If *x* and *y* are infinite or NaN, the return values are undefined.
gentype **min** (gentype *x*, gentype *y*) gentype **min** (gentype *x*, double *y*)	Returns *y* if *y* < *x*, otherwise it returns *x*. If *x* and *y* are infinite or NaN, the return values are undefined.
gentype **mix** (gentype *x*, gentype *y*, gentype *a*) gentype **mix** (gentype *x*, gentype *y*, double *a*)	Returns the linear blend of *x* & *y* implemented as: $x + (y - x) * a$ *a* must be a value in the range 0.0 … 1.0. If *a* is not in the range 0.0 … 1.0, the return values are undefined.
gentype **radians** (gentype *degrees*)	Converts *degrees* to radians, i.e. (π / 180) * *degrees*.
gentype **step** (gentype *edge*, gentype *x*) gentype **step** (double *edge*, gentype *x*)	Returns 0.0 if *x* < *edge*, otherwise it returns 1.0.
genType **smoothstep** (genType *edge0*, genType *edge1*, genType *x*) genType **smoothstep** (double *edge0*, double *edge1*,	Returns 0.0 if *x* <= *edge0* and 1.0 if *x* >= *edge1* and performs smooth Hermite interpolation between 0 and 1when *edge0* < *x* < *edge1*. This is useful in cases where you would want a threshold function with a smooth transition.

---

[59] The **mix** and **smoothstep** functions can be implemented using contractions such as **mad** or **fma**.

genType *x*)	This is equivalent to: gentype t; t = clamp ((x – edge0) / (edge1 – edge0), 0, 1); return t * t * (3 – 2 * t); Results are undefined if *edge0* >= *edge1*.
gentype **sign** (gentype *x*)	Returns 1.0 if $x > 0$, -0.0 if $x = -0.0$, +0.0 if $x = +0.0$, or –1.0 if $x < 0$. Returns 0.0 if *x* is a NaN.

Table 6.11 *Scalar and Vector Argument Built-in Common Function Table*

9.3.4 Geometric Functions[60]

The built-in geometric functions defined in *table 6.13* (also listed below) are extended to include appropriate versions of functions that take `double`, and `double{2|3|4}` as arguments and return values. `gentype` now also includes `double, double2, double3` and `double4`. These are described below.

Function	**Description**
double4 **cross** (double4 *p0*, double4 *p1*) double3 **cross** (double3 *p0*, double3 *p1*)	Returns the cross product of *p0.xyz* and *p1.xyz*. The *w* compoent of double4 result will be 0.0.
double **dot** (gentype *p0*, gentype *p1*)	Compute dot product.
double **distance** (gentype *p0*, gentype *p1*)	Returns the distance between *p0* and *p1*. This is calculated as **length**(*p0* – *p1*).
double **length** (gentype *p*)	Return the length of vector x, i.e., $\sqrt{p.x^2 + p.y^2 + \ldots}$
gentype **normalize** (gentype *p*)	Returns a vector in the same direction as *p* but with a length of 1.

Table 6.13 *Scalar and Vector Argument Built-in Geometric Function Table*

9.3.5 Relational Functions

The scalar and vector relational functions described in *table 6.14* are extended to include versions that take `double`, `double2`, `double3`, `double4`, `double8` and `double16` as arguments.

---

[60] The geometric functions can be implemented using contractions such as **mad** or **fma**.

The relational and equality operators (<, <=, >, >=, !=, ==) can be used with `double`*`n`* vector types and shall produce a vector `long`*`n`* result as described in *section 6.3*.

The functions described in *table 6.14* are extended to include the `double`*`n`* vector types.

Function	**Description**
int **isequal** (double *x*, double *y*) long*n* **isequal** (double*n x*, double*n y*)	Returns the component-wise compare of *x* == *y*.
int **isnotequal** (double *x*, double *y*) long*n* **isnotequal** (double*n x*, double*n y*)	Returns the component-wise compare of *x* != *y*.
int **isgreater** (double *x*, double *y*) long*n* **isgreater** (double*n x*, double*n y*)	Returns the component-wise compare of *x* > *y*.
int **isgreaterequal** (double *x*, double *y*) long*n* **isgreaterequal** (double*n x*, double*n y*)	Returns the component-wise compare of *x* >= *y*.
int **isless** (double *x*, double *y*) long*n* **isless** (double*n x*, double*n y*)	Returns the component-wise compare of *x* < *y*.
int **islessequal** (double *x*, double *y*) long*n* **islessequal** (double*n x*, double*n y*)	Returns the component-wise compare of *x* <= *y*.
int **islessgreater** (double *x*, double *y*) long*n* **islessgreater** (double*n x*, double*n y*)	Returns the component-wise compare of (*x* < *y*) || (*x* > *y*) .
int **isfinite** (double) long*n* **isfinite** (double*n*)	Test for finite value.
int **isinf** (double) long*n* **isinf** (double*n*)	Test for infinity value (+ve or –ve) .
int **isnan** (double) long*n* **isnan** (double*n*)	Test for a NaN.
int **isnormal** (double) long*n* **isnormal** (double*n*)	Test for a normal value.
int **isordered** (double *x*, double *y*) long*n* **isordered** (double*n x*, double*n y*)	Test if arguments are ordered. **isordered**() takes arguments *x* and *y*, and returns the result **isequal**(*x*, *x*) && **isequal**(*y*, *y*).
int **isunordered** (double *x*, double *y*) long*n* **isunordered** (double*n x*, double*n y*)	Test if arguments are unordered. **isunordered**() takes arguments *x* and *y*, returning non-zero if *x* or *y* is NaN, and zero otherwise.
int **signbit** (double) long*n* **signbit** (double*n*)	Test for sign bit. Returns -1 if the sign bit in the float is set; otherwise returns 0.
double*n* **bitselect** (double*n a*, double*n b*, double*n c*)	Each bit of the result is the corresponding bit of *a* if the corresponding bit of *c* is 0. Otherwise it is the corresponding bit of *b*.
double*n* **select** (double*n a*, double*n b*,	For each component, *result[i]* = if MSB of *c[i]* is set ? *b[i]* : *a[i]*.

long*n* *c*) double*n* **select** (double*n* *a*, double*n* *b*, ulong*n* *c*)	

Table 6.14 *Vector Relational Functions*

9.3.6 Vector Data Load and Store Functions

The vector data load (**vload*n***) and store (**vstore*n***) functions described in *table 6.15* (also listed below) are extended to include versions that read from or write to double scalar or vector values. The generic type `gentype` is extended to include `double`. The generic type `gentype`*`n`* is extended to include `double2`, `double3`, `double4`, `double8` and `double16`. The **vstore_half**, **vstore_half*n*** and **vstorea_half*n*** functions are extended to allow a double precision scalar or vector value to be written to memory as half values.

Function	**Description**
gentype*n* **vload*n*** (size_t *offset*, const __global gentype **p*) gentype*n* **vload*n*** (size_t *offset*, const __local gentype **p*) gentype*n* **vload*n*** (size_t *offset*, const __constant gentype **p*) gentype*n* **vload*n*** (size_t *offset*, const __private gentype **p*)	Return sizeof (gentype*n*) bytes of data read from address (*p* + (*offset* * *n*)). The address computed as (*p* + (*offset* * *n*)) must be 8-bit aligned if gentype is char, uchar; 16-bit aligned if gentype is short, ushort; 32-bit aligned if gentype is int, uint, float; 64-bit aligned if gentype is long, ulong or double.
void **vstore*n*** (gentype*n* *data*, size_t *offset*, __global gentype **p*) void **vstore*n*** (gentype*n* *data*, size_t *offset*, __local gentype **p*) void **vstore*n*** (gentype*n* *data*, size_t *offset*, __private gentype **p*)	Write sizeof (gentype*n*) bytes given by *data* to address (*p* + (*offset* * *n*)). The address computed as (*p* + (*offset* * *n*)) must be 8-bit aligned if gentype is char, uchar; 16-bit aligned if gentype is short, ushort; 32-bit aligned if gentype is int, uint, float; 64-bit aligned if gentype is long, ulong or double.
void **vstore_half** (double *data*, size_t *offset*, __global half **p*) void **vstore_half_*rte*** (double *data*, size_t *offset*, __global half **p*) void **vstore_half_*rtz*** (double *data*, size_t *offset*, __global half **p*)	The double value given by *data* is first converted to a half value using the appropriate rounding mode. The half value is then written to address computed as (*p* + *offset*). The address computed as (*p* + *offset*) must be16-bit aligned.

void **vstore_half_*rtp*** (double *data*, size_t *offset*, __global half *p) void **vstore_half_*rtn*** (double *data*, size_t *offset*, __global half *p) void **vstore_half** (double *data*, size_t *offset*, __local half *p) void **vstore_half_*rte*** (double *data*, size_t *offset*, __local half *p) void **vstore_half_*rtz*** (double *data*, size_t *offset*, __local half *p) void **vstore_half_*rtp*** (double *data*, size_t *offset*, __local half *p) void **vstore_half_*rtn*** (double *data*, size_t *offset*, __local half *p) void **vstore_half** (double *data*, size_t *offset*, __private half *p) void **vstore_half_*rte*** (double *data*, size_t *offset*, __private half *p) void **vstore_half_*rtz*** (double *data*, size_t *offset*, __private half *p) void **vstore_half_*rtp*** (double *data*, size_t *offset*, __private half *p) void **vstore_half_*rtn*** (double *data*, size_t *offset*, __private half *p)	**vstore_half** use the current rounding mode. The default current rounding mode is round to nearest even.
void **vstore_half*n*** (double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_*rte*** (double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_*rtz*** (double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_*rtp*** (double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_*rtn*** (double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*** (double*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_*rte*** (double*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_*rtz*** (double*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_*rtp*** (double*n* *data*, size_t *offset*, __local half *p)	The double*n* value given by *data* is converted to a half*n* value using the appropriate rounding mode. The half*n* value is then written to address computed as (*p* + (*offset* * *n*)). The address computed as (*p* + (*offset* * *n*)) must be 16-bit aligned. **vstore_half*n*** uses the current rounding mode. The default current rounding mode is round to nearest even.

void **vstore_half*n*_rtn** (double*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*** (double*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rte** (double*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rtz** (double*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rtp** (double*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rtn** (double*n* *data*, size_t *offset*, __private half *p)	
void **vstorea_half*n*** (double*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n*_rte** (double*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n*_rtz** (double*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n*_rtp** (double*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n*_rtn** (double*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n*** (double*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n*_rte** (double*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n*_rtz** (double*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n*_rtp** (double*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n*_rtn** (double*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n*** (double*n* *data*, size_t *offset*, __private half *p) void **vstorea_half*n*_rte** (double*n* *data*, size_t *offset*, __private half *p) void **vstorea_half*n*_rtz** (double*n* *data*, size_t *offset*, __private half *p) void **vstorea_half*n*_rtp** (double*n* *data*, size_t *offset*, __private half *p) void **vstorea_half*n*_rtn** (double*n* *data*,	The double*n* value is converted to a half*n* value using the appropriate rounding mode. For n = 1, 2, 4, 8 or 16, the half*n* value is written to the address computed as (p + (offset * n)). The address computed as (p + (offset * n)) must be aligned to sizeof (half*n*) bytes. For n = 3, the half*3* value is written to the address computed as (p + (offset * 4)). The address computed as (p + (offset * 4)) must be aligned to sizeof (half) * 4 bytes. **vstorea_half*n*** uses the current rounding mode. The default current rounding mode is round to nearest even.

size_t *offset*, __private half **p*)	

Table 6.15 *Vector Data Load and Store Functions*

NOTE: **vload*3*** reads *x*, *y*, *z* components from address (*p* + (*offset* * 3)) into a 3-component vector. **vstore*3***, and **vstore_half*3*** write *x*, *y*, *z* components from a 3-component vector to address (*p* + (*offset* * 3)).

In addition **vstorea_half*3*** writes *x*, *y*, *z* components from a 3-component vector to address (*p* + (*offset* * 4)).

9.3.7 Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch

The OpenCL C programming language implements the following functions that provide asynchronous copies between global and local memory and a prefetch from global memory.

The generic type `gentype` is extended to include `double`, `double2`, `double4`, `double8` and `double16`.

Function	**Description**
event_t **async_work_group_copy** (__local gentype **dst*, const __global gentype **src*, size_t *num_gentypes*, event_t *event*) event_t **async_work_group_copy** (__global gentype **dst*, const __local gentype **src*, size_t *num_gentypes*, event_t *event*)	Perform an async copy of *num_gentypes* `gentype` elements from *src* to *dst*. The async copy is performed by all work-items in a work-group and this built-in function must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined. Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. If *event* argument is not zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit synchronization of source data such as using a **barrier** before performing the copy.

event_t **async_work_group_strided_copy** (__local gentype **dst*, const __global gentype **src*, size_t *num_gentypes*, size_t *src_stride*, event_t *event*) event_t **async_work_group_strided_copy** (__global gentype **dst*, const __local gentype **src*, size_t *num_gentypes*, size_t *dst_stride*, event_t *event*)	Perform an async gather of *num_gentypes* `gentype` elements from *src* to *dst*. The *src_stride* is the stride in elements for each gentype element read from *src*. The async gather is performed by all work-items in a work-group and this built-in function must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined. Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_strided_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. If *event* argument is non-zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit synchronization of source data such as using a **barrier** before performing the copy. The behavior of **async_work_group_strided_copy** is undefined if *src_stride* or *dst_stride* is 0, or if the *src_stride* or *dst_stride* values cause the *src* or *dst* pointers to exceed the upper bounds of the address space during the copy.
void **wait_group_events** (int *num_events*, event_t **event_list*)	Wait for events that identify the **async_work_group_copy** operations to complete. The event objects specified in *event_list* will be released after the wait is performed. This function must be encountered by all work-items in a work-group executing the kernel with the same *num_events* and event objects specified in *event_list*; otherwise the results are undefined.
void **prefetch** (const __global gentype **p*, size_t *num_gentypes*)	Prefetch *num_gentypes* * sizeof(`gentype`) bytes into the global cache. The prefetch instruction is

	applied to a work-item in a work-group and does not affect the functional behavior of the kernel.

Table 6.19 *Built-in Async Copy and Prefetch functions*

9.3.8 IEEE754 Compliance

The following table entry describes the additions to *table 4.3,* which allows applications to query the configuration information using **clGetDeviceInfo** for an OpenCL device that supports double precision floating-point.

Op-code	Return Type	Description
CL_DEVICE_DOUBLE_FP_CONFIG	cl_device_fp_config	Describes double precision floating-point capability of the OpenCL device. This is a bit-field that describes one or more of the following values: CL_FP_DENORM – denorms are supported CL_FP_INF_NAN – INF and NaNs are supported. CL_FP_ROUND_TO_NEAREST – round to nearest even rounding mode supported. CL_FP_ROUND_TO_ZERO – round to zero rounding mode supported. CL_FP_ROUND_TO_INF – round to +ve and –ve infinity rounding modes supported. CP_FP_FMA – IEEE754-2008 fused multiply-add is supported. CL_FP_SOFT_FLOAT – Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. The mandated minimum double

		precision floating-point capability is CL_FP_FMA | CL_FP_ROUND_TO_NEAREST | CL_FP_ROUND_TO_ZERO | CL_FP_ROUND_TO_INF | CL_FP_INF_NAN | CL_FP_DENORM.

IEEE754 fused multiply-add, denorms, INF and NaNs are required to be supported for double precision floating-point numbers and operations on double precision floating-point numbers.

9.3.9 Relative Error as ULPs

In this section we discuss the maximum relative error defined as *ulp* (units in the last place). Addition, subtraction, multiplication, fused multiply-add and conversion between integer and a floating-point format are IEEE 754 compliant and are therefore correctly rounded using round-to-nearest even rounding mode.

The following table describes the minimum accuracy of double precision floating-point arithmetic operations given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

Function	Min Accuracy - ULP values[61]
$x + y$	Correctly rounded
$x - y$	Correctly rounded
$x * y$	Correctly rounded
1.0 $/ x$	Correctly rounded
x / y	Correctly rounded
acos	<= 4 ulp
acospi	<= 5 ulp
asin	<= 4 ulp
asinpi	<= 5 ulp
atan	<= 5 ulp
atan2	<= 6 ulp
atanpi	<= 5 ulp
atan2pi	<= 6 ulp
acosh	<= 4 ulp
asinh	<= 4 ulp
atanh	<= 5 ulp
cbrt	<= 2 ulp
ceil	Correctly rounded

---

[61] 0 ulp is used for math functions that do not require rounding.

copysign	0 ulp
cos	<= 4 ulp
cosh	<= 4 ulp
cospi	<= 4 ulp
erfc	<= 16 ulp
erf	<= 16 ulp
exp	<= 3 ulp
exp2	<= 3 ulp
exp10	<= 3 ulp
expm1	<= 3 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp
hypot	<= 4 ulp
ilogb	0 ulp
ldexp	Correctly rounded
log	<= 3 ulp
log2	<= 3 ulp
log10	<= 3 ulp
log1p	<= 2 ulp
logb	0 ulp
mad	Any value allowed (infinite ulp)
maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	<= 16 ulp
pown(x, y)	<= 16 ulp
powr(x, y)	<= 16 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	<= 16 ulp
round	Correctly rounded
rsqrt	<= 2 ulp
sin	<= 4 ulp
sincos	<= 4 ulp for sine and cosine values
sinh	<= 4 ulp

sinpi	<= 4 ulp
sqrt	Correctly rounded
tan	<= 5 ulp
tanh	<= 5 ulp
tanpi	<= 6 ulp
tgamma	<= 16 ulp
trunc	Correctly rounded

9.4 64-bit Atomics

The optional extensions **cl_khr_int64_base_atomics** and **cl_khr_int64_extended_atomics** implement atomic operations on 64-bit signed and unsigned integers to locations in `__global` and `__local` memory. .

An application that wants to use any of these extensions will need to include the **`#pragma OPENCL EXTENSION cl_khr_int64_base_atomics : enable`** or **`#pragma OPENCL EXTENSION cl_khr_int64_extended_atomics : enable`** directive in the OpenCL program source.

The atomic functions supported by the **cl_khr_int64_base_atomics** extension are described in *table 9.1*. All of the functions listed in *table 9.1* are performed in one atomic transaction.

The atomic functions supported by the **cl_khr_int64_extended_atomics** extension are described in *table 9.2*. All of the functions listed in *table 9.2* are performed in one atomic transaction.

These transactions are atomic for the device executing these atomic functions. There is no guarantee of atomicity if the atomic operations to the same memory location are being performed by kernels executing on multiple devices.

Function	**Description**
long **atom_add** (volatile __global long **p*, long *val*) long **atom_add** (volatile __local long **p*, long *val*) ulong **atom_add** (volatile __global ulong **p*, ulong *val*) ulong **atom_add** (volatile __local ulong **p*, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* + *val*) and store result at location pointed by *p*. The function returns *old*.
long **atom_sub** (volatile __global long **p*, long *val*) long **atom_sub** (volatile __local long **p*, long *val*) ulong **atom_sub** (volatile __global ulong **p*, ulong *val*) ulong **atom_sub** (volatile __local ulong **p*, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* - *val*) and store result at location pointed by *p*. The function returns *old*.
long **atom_xchg** (volatile __global long **p*, long *val*) long **atom_xchg** (volatile __local long **p*, long *val*) ulong **atom_xchg** (volatile __global ulong **p*, ulong *val*) ulong **atom_xchg** (volatile __local ulong **p*, ulong *val*)	Swaps the *old* value stored at location *p* with new value given by *val*. Returns *old* value.
long **atom_inc** (volatile __global long **p*) long **atom_inc** (volatile __local long **p*) ulong **atom_inc** (volatile __global ulong **p*) ulong **atom_inc** (volatile __local ulong **p*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* + *1*) and store result at location pointed by *p*. The function returns *old*.
long **atom_dec** (volatile __global long **p*) long **atom_dec** (volatile __local long **p*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*.

ulong **atom_dec** (volatile __global ulong **p*) ulong **atom_dec** (volatile __local ulong **p*)	Compute (*old* - *1*) and store result at location pointed by *p*. The function returns *old*.
long **atom_cmpxchg** (volatile __global long **p*, long *cmp*, long *val*) long **atom_cmpxchg** (volatile __local long **p*, long *cmp*, long *val*) ulong **atom_cmpxchg** (volatile __global ulong **p*, ulong *cmp*, ulong *val*) ulong **atom_cmpxchg** (volatile __local ulong **p*, ulong *cmp*, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* == *cmp*) ? *val* : *old* and store result at location pointed by *p*. The function returns *old*.

Table 9.1 *Built-in Atomic Functions for* **cl_khr_int64_base_atomics** *extension*

Function	**Description**
long **atom_min** (volatile __global long **p*, long *val*) long **atom_min** (volatile __local long **p*, long *val*) ulong **atom_min** (volatile __global ulong **p*, ulong *val*) ulong **atom_min** (volatile __local ulong **p*, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute **min**(*old*, *val*) and store minimum value at location pointed by *p*. The function returns *old*.
long **atom_max** (volatile __global long **p*, long *val*) long **atom_max** (volatile __local long **p*, long *val*) ulong **atom_max** (volatile __global ulong **p*, ulong *val*) ulong **atom_max** (volatile __local ulong **p*, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute **max**(*old*, *val*) and store maximum value at location pointed by *p*. The function returns *old*.
long **atom_and** (volatile __global long **p*, long *val*) long **atom_and** (volatile __local long **p*, long *val*) ulong **atom_and** (volatile __global ulong **p*, ulong *val*) ulong **atom_and** (volatile __local ulong **p*, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* & val) and store result at location pointed by *p*. The function returns *old*.
long **atom_or** (volatile __global long **p*, long *val*) long **atom_or** (volatile __local long **p*, long *val*) ulong **atom_or** (volatile __global ulong **p*, ulong *val*) ulong **atom_or** (volatile __local ulong **p*, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* | val) and store result at location pointed by *p*. The function returns *old*.
long **atom_xor** (volatile __global long **p*, long *val*) long **atom_xor** (volatile __local long **p*, long *val*) ulong **atom_xor** (volatile __global ulong **p*, ulong *val*) ulong **atom_xor** (volatile __local ulong **p*, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* ^ val) and store result at location pointed by *p*. The function returns *old*.

Table 9.2 *Built-in Atomic Functions for* **cl_khr_int64_extended_atomics** *extension*

Note: Atomic operations on 64-bit integers and 32-bit integers (and float) are also atomic w.r.t. each other.

9.5 Writing to 3D image memory objects

OpenCL 1.0 supports 2D image memory objects that can be read or written by kernels. Reads and writes to the same 2D image memory object are not supported in a kernel. OpenCL 1.0 also supports reads to 3D image memory objects in kernels. Writes to a 3D image memory object is not allowed. The **`cl_khr_3d_image_writes`** extension implements writes to 3D image memory objects. Reads and writes to the same 3D image memory object are not allowed in a kernel.

An application that wants to use this extension to write to 3D image memory objects will need to include the **`#pragma OPENCL EXTENSION cl_khr_3d_image_writes : enable`** directive in the OpenCL program source.

The built-in functions implemented by the **`cl_khr_3d_image_writes`** extension are described in the table below.

Function	Description
void **write_imagef** (image3d_t *image*, int4 *coord*, float4 *color*) void **write_imagei** (image3d_t *image*, int4 *coord*, int4 *color*) void **write_imageui** (image3d_t *image*, int4 *coord*, uint4 *color*)	Write *color* value to location specified by coordinate *(x, y, z)* in the 3D image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord.x, coord.y and coord.z* are considered to be unnormalized coordinates and must be in the range 0 ... image stream width – 1, 0 … image stream height – 1 and 0 … image stream depth – 1. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to CL_SNORM_INT8, CL_UNORM_INT8, CL_SNORM_INT16, CL_UNORM_INT16, CL_HALF_FLOAT or CL_FLOAT. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** 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. **write_imageui** 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. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects with *image_channel_data_type* values not specified in the description above or with (*x*, *y*, *z*) coordinate values that are not in the range (0 … image width – 1, 0 … image height – 1, 0 … image depth – 1) respectively is undefined.

9.6 Half Floating-Point

This extension adds support for half scalar and vector types as built-in types that can be used for arithmetic operations, conversions etc. An application that wants to use `half` and `halfn` types will need to include the **`#pragma OPENCL EXTENSION cl_khr_fp16 : enable`** directive.

The list of built-in scalar, and vector data types defined in *tables 6.1*, and *6.2* are extended to include the following:

Type	Description
half2	A 2-component half-precision floating-point vector.
half3	A 3-component half-precision floating-point vector.
half4	A 4-component half-precision floating-point vector.
half8	A 8-component half-precision floating-point vector.
half16	A 16-component half-precision floating-point vector.

The built-in vector data types for `halfn` are also declared as appropriate types in the OpenCL API (and header files) that can be used by an application. The following table describes the built-in vector data types for `halfn` as defined in the OpenCL C programming language and the corresponding data type available to the application:

Type in OpenCL Language	API type for application
half2	**cl_half2**
half 3	**cl_half3**
half 4	**cl_half4**
half 8	**cl_half8**
half16	**cl_half16**

9.6.1 Conversions

The implicit conversion rules specified in *section 6.2.1* now include the `half` scalar and `half`*`n`* vector data types.

The explicit casts described in *section 6.2.2* are extended to take a `half` scalar data type and a `half`*`n`* vector data type.

The explicit conversion functions described in *section 6.2.3* are extended to take a `half` scalar data type and a `half`*`n`* vector data type.

The `as_typen()` function for re-interpreting types as described in *section 6.2.4.2* is extended to allow conversion-free casts between `short`*n*, `ushort`*n* and `half`*n* scalar and vector data types.

9.6.2 Math Functions

The built-in math functions defined in *table 6.8* (also listed below) are extended to include appropriate versions of functions that take `half,` and `half{2|3|4|8|16}` as arguments and return values. `gentype` now also includes `half, half2, half3, half4, half8` and `half16`.

For any specific use of a function, the actual type has to be the same for all arguments and the return type.

Function	Description
gentype **acos** (gentype)	Arc cosine function.
gentype **acosh** (gentype)	Inverse hyperbolic cosine.
gentype **acospi** (gentype *x*)	Compute **acos** (*x*) / π.
gentype **asin** (gentype)	Arc sine function.
gentype **asinh** (gentype)	Inverse hyperbolic sine.
gentype **asinpi** (gentype *x*)	Compute **asin** (*x*) / π.
gentype **atan** (gentype *y_over_x*)	Arc tangent function.
gentype **atan2** (gentype *y*, gentype *x*)	Arc tangent of *y* / *x*.
gentype **atanh** (gentype)	Hyperbolic arc tangent.
gentype **atanpi** (gentype *x*)	Compute **atan** (*x*) / π.
gentype **atan2pi** (gentype *y*, gentype *x*)	Compute **atan2** (*y*, *x*) / π.
gentype **cbrt** (gentype)	Compute cube-root.
gentype **ceil** (gentype)	Round to integral value using the round to +ve infinity rounding mode.
gentype **copysign** (gentype *x*, gentype *y*)	Returns *x* with its sign changed to match the sign of *y*.
gentype **cos** (gentype)	Compute cosine.
gentype **cosh** (gentype)	Compute hyperbolic consine.
gentype **cospi** (gentype *x*)	Compute **cos** (π *x*).
gentype **erfc** (gentype)	Complementary error function.
gentype **erf** (gentype)	Error function encountered in integrating the normal distribution.
gentype **exp** (gentype *x*)	Compute the base- e exponential of *x*.
gentype **exp2** (gentype)	Exponential base 2 function.
gentype **exp10** (gentype)	Exponential base 10 function.
gentype **expm1** (gentype *x*)	Compute e^x- 1.0.
gentype **fabs** (gentype)	Compute absolute value of a floating-point number.
gentype **fdim** (gentype *x*, gentype *y*)	*x* - *y* if *x* > *y*, +0 if x is less than or equal to y.

gentype **floor** (gentype)	Round to integral value using the round to –ve infinity rounding mode.
gentype **fma** (gentype *a*, gentype *b*, gentype *c*)	Returns the correctly rounded floating-point representation of the sum of *c* with the infinitely precise product of *a* and *b*. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.
gentype **fmax** (gentype x, gentype y) gentype **fmax** (gentype *x*, half *y*)	Returns *y* if $x < y$, otherwise it returns *x*. If one argument is a NaN, **fmax()** returns the other argument. If both arguments are NaNs, **fmax()** returns a NaN.
gentype **fmin** (gentype *x*, gentype *y*) gentype **fmin** (gentype *x*, half *y*)	Returns *y* if $y < x$, otherwise it returns *x*. If one argument is a NaN, **fmin()** returns the other argument. If both arguments are NaNs, **fmin()** returns a NaN.
gentype **fmod** (gentype *x*, gentype *y*)	Modulus. Returns $x – y$ * **trunc** (*x*/*y*) .
gentype **fract** (gentype *x*, __global gentype **iptr*) gentype **fract** (gentype *x*, __local gentype **iptr*) gentype **fract** (gentype *x*, __private gentype **iptr*)	Returns **fmin**(*x* – **floor** (*x*), 0x1.ffcp-1f). **floor**(x) is returned in *iptr*.
gentype **frexp** (gentype *x*, __global int*n* *exp) gentype **frexp** (gentype *x*, __local int*n* *exp) gentype **frexp** (gentype *x*, __private int*n* *exp)	Extract mantissa and exponent from *x*. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of *x* equals mantissa returned * 2^{exp}.
gentype **hypot** (gentype *x*, gentype *y*)	Compute the value of the square root of $x^2+ y^2$ without undue overflow or underflow.
int*n* **ilogb** (gentype *x*)	Return the exponent as an integer value.
gentype **ldexp** (gentype *x*, int*n n*) gentype **ldexp** (gentype *x*, int *n*)	Multiply *x* by 2 to the power *n*.
gentype **lgamma** (gentype *x*) gentype **lgamma_r** (gentype *x*, __global int*n* **signp*) gentype **lgamma_r** (gentype *x*, __local int*n* **signp*) gentype **lgamma_r** (gentype *x*, __private int*n* **signp*)	Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the *signp* argument of **lgamma_r**.
gentype **log** (gentype)	Compute natural logarithm.
gentype **log2** (gentype)	Compute a base 2 logarithm.
gentype **log10** (gentype)	Compute a base 10 logarithm.
gentype **log1p** (gentype *x*)	Compute $\log_e(1.0 + x)$.
gentype **logb** (gentype *x*)	Compute the exponent of *x*, which is the integral

	part of $\log_r \lvert x \rvert$.
gentype **mad** (gentype a, gentype b, gentype c)	**mad** approximates $a * b + c$. Whether or how the product of $a * b$ is rounded and how supernormal or subnormal intermediate products are handled is not defined. **mad** is intended to be used where speed is preferred over accuracy[62].
gentype **maxmag** (gentype x, gentype y)	Returns x if $\lvert x \rvert > \lvert y \rvert$, y if $\lvert y \rvert > \lvert x \rvert$, otherwise **fmax**($x$, y).
gentype **minmag** (gentype x, gentype y)	Returns x if $\lvert x \rvert < \lvert y \rvert$, y if $\lvert y \rvert < \lvert x \rvert$, otherwise **fmin**($x$, y).
gentype **modf** (gentype x, __global gentype *iptr) gentype **modf** (gentype x, __local gentype *iptr) gentype **modf** (gentype x, __private gentype *iptr)	Decompose a floating-point number. The **modf** function breaks the argument x into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by *iptr*.
halfn **nan** (ushortn nancode)	Returns a quiet NaN. The *nancode* may be placed in the significand of the resulting NaN.
gentype **nextafter** (gentype x, gentype y)	Computes the next representable single-precision floating-point value following x in the direction of y. Thus, if y is less than x, **nextafter**() returns the largest representable floating-point number less than x.
gentype **pow** (gentype x, gentype y)	Compute x to the power y.
gentype **pown** (gentype x, intn y)	Compute x to the power y, where y is an integer.
gentype **powr** (gentype x, gentype y)	Compute x to the power y, where x is >= 0.
gentype **remainder** (gentype x, gentype y)	Compute the value r such that $r = x - n*y$, where n is the integer nearest the exact value of x/y. If there are two integers closest to x/y, n shall be the even one. If r is zero, it is given the same sign as x.
gentype **remquo** (gentype x, gentype y, __global intn *quo) gentype **remquo** (gentype x, gentype y, __local intn *quo) gentype **remquo** (gentype x, gentype y, __private intn *quo)	The **remquo** function computes the value r such that $r = x - n*y$, where n is the integer nearest the exact value of x/y. If there are two integers closest to x/y, n shall be the even one. If r is zero, it is given the same sign as x. This is the same value that is returned by the **remainder** function. **remquo** also calculates the lower seven bits of the integral quotient x/y, and gives that value the same sign as x/y. It stores this signed value in the object pointed to by *quo*.
gentype **rint** (gentype)	Round to integral value (using round to nearest

---

[62] The user is cautioned that for some usages, e.g. **mad**(a, b, -a*b), the definition of **mad**() is loose enough that almost any result is allowed from **mad**() for some values of a and b.

	even rounding mode) in floating-point format. Refer to section 7.1 for description of rounding modes.
gentype **rootn** (gentype *x*, int*n y*)	Compute *x* to the power 1/*y*.
gentype **round** (gentype *x*)	Return the integral value nearest to *x* rounding halfway cases away from zero, regardless of the current rounding direction.
gentype **rsqrt** (gentype)	Compute inverse square root.
gentype **sin** (gentype)	Compute sine.
gentype **sincos** (gentype *x*, __global gentype **cosval*) gentype **sincos** (gentype *x*, __local gentype **cosval*) gentype **sincos** (gentype *x*, __private gentype **cosval*)	Compute sine and cosine of x. The computed sine is the return value and computed cosine is returned in *cosval*.
gentype **sinh** (gentype)	Compute hyperbolic sine.
gentype **sinpi** (gentype *x*)	Compute **sin** (π *x*).
gentype **sqrt** (gentype)	Compute square root.
gentype **tan** (gentype)	Compute tangent.
gentype **tanh** (gentype)	Compute hyperbolic tangent.
gentype **tanpi** (gentype *x*)	Compute **tan** (π *x*).
gentype **tgamma** (gentype)	Compute the gamma function.
gentype **trunc** (gentype)	Round to integral value using the round to zero rounding mode.

Table 6.8 *Scalar and Vector Argument Built-in Math Function Table*

The **FP_FAST_FMA_HALF** macro indicates whether the **fma()** family of functions are fast compared with direct code for half precision floating-point. If defined, the **FP_FAST_FMA_HALF** macro shall indicate that the **fma()** function generally executes about as fast as, or faster than, a multiply and an add of **half** operands

9.6.3 Common Functions[63]

The built-in common functions defined in *table 6.12* (also listed below) are extended to include appropriate versions of functions that take `half`, and `half{2|3|4|8|16}` as arguments and return values. `gentype` now also includes `half, half2, half3, half4, half8` and `half16`. These are described below.

Function	**Description**
gentype **clamp** (gentype *x*,	Returns **min**(**max**(*x*, *minval*), *maxval*) .

---

[63] The **mix** and **smoothstep** functions can be implemented using contractions such as **mad** or **fma**.

gentype *minval*, gentype *maxval*) gentype **clamp** (gentype *x*, half *minval*, half *maxval*)	Results are undefined in *minval* > *maxval*.
gentype **degrees** (gentype *radians*)	Converts *radians* to degrees, i.e. (180 / π) * *radians*.
gentype **max** (gentype *x*, gentype *y*) gentype **max** (gentype *x*, half *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*. If *x* and *y* are infinite or NaN, the return values are undefined.
gentype **min** (gentype *x*, gentype *y*) gentype **min** (gentype *x*, half *y*)	Returns *y* if *y* < *x*, otherwise it returns *x*. If *x* and *y* are infinite or NaN, the return values are undefined.
gentype **mix** (gentype *x*, gentype *y*, gentype *a*) gentype **mix** (gentype *x*, gentype *y*, half *a*)	Returns the linear blend of *x* & *y* implemented as: $x + (y – x) * a$ *a* must be a value in the range 0.0 … 1.0. If *a* is not in the range 0.0 … 1.0, the return values are undefined.
gentype **radians** (gentype *degrees*)	Converts *degrees* to radians, i.e. (π / 180) * *degrees*.
gentype **step** (gentype *edge*, gentype *x*) gentype **step** (half *edge*, gentype *x*)	Returns 0.0 if *x* < *edge*, otherwise it returns 1.0.
genType **smoothstep** (genType *edge0*, genType *edge1*, genType *x*) genType **smoothstep** (half *edge0*, half *edge1*, genType *x*)	Returns 0.0 if *x* <= *edge0* and 1.0 if *x* >= *edge1* and performs smooth Hermite interpolation between 0 and 1when *edge0* < *x* < *edge1*. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to: gentype t; t = clamp ((x – edge0) / (edge1 – edge0), 0, 1); return t * t * (3 – 2 * t); Results are undefined if *edge0* >= *edge1*.
gentype **sign** (gentype *x*)	Returns 1.0 if *x* > 0, -0.0 if *x* = -0.0, +0.0 if *x* = +0.0, or –1.0 if *x* < 0. Returns 0.0 if *x* is a NaN.

Table 6.12 *Scalar and Vector Argument Built-in Common Function Table*

9.6.4 Geometric Functions[64]

The built-in geometric functions defined in *table 6.13* (also listed below) are extended to include appropriate versions of functions that take `half`, and `half{2|3|4}` as arguments and return values. `gentype` now also includes `half, half2, half3` and `half4`. These are described below.

Function	Description
half4 **cross** (half4 *p0*, half4 *p1*) half3 **cross** (half3 *p0*, half3 *p1*)	Returns the cross product of *p0.xyz* and *p1.xyz*. The *w* compoent of double result will be 0.0.
half **dot** (gentype *p0*, gentype *p1*)	Compute dot product.
half **distance** (gentype *p0*, gentype *p1*)	Returns the distance between *p0* and *p1*. This is calculated as **length**(*p0* – *p1*).
half **length** (gentype *p*)	Return the length of vector x, i.e., $\sqrt{p.x^2 + p.y^2 + \dots}$
gentype **normalize** (gentype *p*)	Returns a vector in the same direction as *p* but with a length of 1.

Table 6.13 *Scalar and Vector Argument Built-in Geometric Function Table*

9.6.5 Relational Functions

The scalar and vector relational functions described in *table 6.14* are extended to include versions that take `half`, `half2`, `half3`, `half4`, `half8` and `half16` as arguments.

The relational and equality operators (<, <=, >, >=, !=, ==) can be used with `halfn` vector types and shall produce a vector `shortn` result as described in *section 6.3*.

The functions described in *table 6.14* are extended to include the `halfn` vector types.

Function	Description
int **isequal** (half *x*, half *y*) short*n* **isequal** (half*n x*, half*n y*)	Returns the component-wise compare of *x* == *y*.
int **isnotequal** (half *x*, half *y*) short*n* **isnotequal** (half*n x*, half*n y*)	Returns the component-wise compare of *x* != *y*.
int **isgreater** (half *x*, half *y*) short*n* **isgreater** (half*n x*, half*n y*)	Returns the component-wise compare of *x* > *y*.
int **isgreaterequal** (half *x*,	Returns the component-wise compare of *x* >= *y*.

---

[64] The geometric functions can be implemented using contractions such as **mad** or **fma**.

half *y*) short*n* **isgreaterequal** (half*n* *x*, half*n* *y*)	
int **isless** (half *x*, half *y*) short*n* **isless** (half*n* *x*, half*n* *y*)	Returns the component-wise compare of $x < y$.
int **islessequal** (half *x*, half *y*) short*n* **islessequal** (half*n* *x*, half*n* *y*)	Returns the component-wise compare of $x <= y$.
int **islessgreater** (half *x*, half *y*) short*n* **islessgreater** (half*n* *x*, half*n* *y*)	Returns the component-wise compare of $(x < y) \|\| (x > y)$.
int **isfinite** (half) short*n* **isfinite** (half*n*)	Test for finite value.
int **isinf** (half) short*n* **isinf** (half*n*)	Test for infinity value (+ve or –ve) .
int **isnan** (half) short*n* **isnan** (half*n*)	Test for a NaN.
int **isnormal** (half) short*n* **isnormal** (half*n*)	Test for a normal value.
int **isordered** (half *x*, half *y*) short*n* **isordered** (half*n* *x*, half*n* *y*)	Test if arguments are ordered. **isordered**() takes arguments *x* and *y*, and returns the result **isequal**(*x*, *x*) && **isequal**(*y*, *y*).
int **isunordered** (half *x*, half *y*) short*n* **isunordered** (half*n* *x*, half*n* *y*)	Test if arguments are unordered. **isunordered**() takes arguments *x* and *y*, returning non-zero if *x* or *y* is a NaN, and zero otherwise.
int **signbit** (half) short*n* **signbit** (half*n*)	Test for sign bit. Returns -1 if the sign bit in the float is set; otherwise returns 0.
half*n* **bitselect** (half*n* *a*, half*n* *b*, half*n* *c*)	Each bit of the result is the corresponding bit of *a* if the corresponding bit of *c* is 0. Otherwise it is the corresponding bit of *b*.
half*n* **select** (half*n* *a*, half*n* *b*, short*n* *c*) half*n* **select** (half*n* *a*, half*n* *b*, ushort*n* *c*)	For each component, *result[i]* = if MSB of *c[i]* is set ? *b[i]* : *a[i]*.

Table 6.14 *Vector Relational Functions*

9.6.6 Vector Data Load and Store Functions

The vector data load (**vload*n***) and store (**vstore*n***) functions described in *table 6.14* (also listed below) are extended to include versions that read from or write to half scalar or vector values. The generic type `gentype` is extended to include `half`. The generic type `gentypen` is extended to include `half`,`half2`, `half3`, `half4`, `half8` and `half16`.

Function	**Description**
gentype*n* **vload*n*** (size_t *offset*, const __global gentype **p*) gentype*n* **vload*n*** (size_t *offset*, const __local gentype **p*) gentype*n* **vload*n*** (size_t *offset*, const __constant gentype **p*) gentype*n* **vload*n*** (size_t *offset*, const __private gentype **p*)	Return sizeof (gentype*n*) bytes of data read from address (*p* + (*offset* * *n*)). The read address computed as (*p* + (*offset* * *n*)) must be 16-bit aligned.
void **vstore*n*** (gentype*n* *data*, size_t *offset*, __global gentype **p*) void **vstore*n*** (gentype*n* *data*, size_t *offset*, __local gentype **p*) void **vstore*n*** (gentype*n* *data*, size_t *offset*, __private gentype **p*)	Write sizeof (gentype*n*) bytes given by *data* to address (*p* + (*offset* * *n*)). The write address computed as (*p* + (*offset* * *n*)) must be 16-bit aligned.

Table 6.15 *Vector Data Load and Store Functions*

NOTE: **vload*3*** reads *x*, *y*, *z* components from address (*p* + (*offset* * 3)) into a 3-component vector and **vstore*3*** writes *x*, *y*, *z* components from a 3-component vector to address (*p* + (*offset* * 3)).

9.6.7 Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch

The OpenCL C programming language implements the following functions that provide asynchronous copies between global and local memory and a prefetch from global memory.

The generic type `gentype` is extended to include `half2`, `half4`, `half8` and `half16`.

Function	**Description**
event_t **async_work_group_copy** (__local gentype **dst*, const __global gentype **src*,	Perform an async copy of *num_gentypes* `gentype` elements from *src* to *dst*. The async

size_t *num_gentypes*, event_t *event*) event_t **async_work_group_copy** (__global gentype **dst*, const __local gentype **src*, size_t *num_gentypes*, event_t *event*)	copy is performed by all work-items in a work-group and this built-in function must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined. Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. If *event* argument is not zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit synchronization of source data such as using a **barrier** before performing the copy.
event_t **async_work_group_strided_copy** (__local gentype **dst*, const __global gentype **src*, size_t *num_gentypes*, size_t *src_stride*, event_t *event*) event_t **async_work_group_strided_copy** (__global gentype **dst*, const __local gentype **src*, size_t *num_gentypes*, size_t *dst_stride*, event_t *event*)	Perform an async gather of *num_gentypes* `gentype` elements from *src* to *dst*. The *src_stride* is the stride in elements for each gentype element read from *src*. The async gather is performed by all work-items in a work-group and this built-in function must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined. Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_strided_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. If *event* argument is not zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit

	synchronization of source data such as using a **barrier** before performing the copy. The behavior of **async_work_group_strided_copy** is undefined if *src_stride* or *dst_stride* is 0, or if the *src_stride* or *dst_stride* values cause the *src* or *dst* pointers to exceed the upper bounds of the address space during the copy.
void **wait_group_events** (int *num_events*, event_t **event_list*)	Wait for events that identify the **async_work_group_copy** operations to complete. The event objects specified in *event_list* will be released after the wait is performed. This function must be encountered by all work-items in a work-group executing the kernel with the same *num_events* and event objects specified in *event_list*; otherwise the results are undefined.
void **prefetch** (const __global gentype **p*, size_t *num_gentypes*)	Prefetch *num_gentypes* * sizeof(`gentype`) bytes into the global cache. The prefetch instruction is applied to a work-item in a work-group and does not affect the functional behavior of the kernel.

Table 6.18 *Built-in Async Copy and Prefetch functions*

9.6.8 Image Read and Write Functions

The image read and write functions defined in *table 6.22* are extended to support image color values that are a `half` type.

Function	**Description**
half4 **read_imageh** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) half4 **read_imageh** (image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate *(x, y)* to do an element lookup in the 2D image object specified by *image*. **read_imageh** returns half 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, CL_UNORM_INT8, or CL_UNORM_INT16. **read_imageh** returns half 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_imageh** returns half floating-point values for image objects created with *image_channel_data_type* set to CL_HALF_FLOAT. The **read_imageh** calls that take integer coordinates must use a sampler with filter mode set to CLK_FILTER_NEAREST, normalized coordinates set to CLK_NORMALIZED_COORDS_FALSE and addressing mode set to CLK_ADDRESS_CLAMP_TO_EDGE, CLK_ADDRESS_CLAMP or CLK_ADDRESS_NONE; otherwise the values returned are undefined. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description above are undefined.
void **write_imageh** (image2d_t *image*, int2 *coord*, half4 *color*)	Write *color* value to location specified by coordinate *(x, y)* in the 2D image specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *x* & *y* are considered to be unnormalized coordinates and must be in the range 0 ... width – 1, and 0 … height – 1. **write_imageh** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to CL_SNORM_INT8, CL_UNORM_INT8, CL_SNORM_INT16, CL_UNORM_INT16, or CL_HALF_FLOAT. The behavior of **write_imageh** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*) coordinate values that are not in the range (0 … width – 1, 0 … height – 1) respectively, is undefined.

Table 6.22 *Built-in Image Read and Write Functions*

9.6.9 IEEE754 Compliance

The following table entry describes the additions to *table 4.3,* which allows applications to query the configuration information using **clGetDeviceInfo** for an OpenCL device that supports half precision floating-point.

Op-code	Return Type	Description
CL_DEVICE_HALF_FP_CONFIG	cl_device_ fp_config	Describes half precision floating-point capability of the OpenCL device. This is a bit-field that describes one or more of the following values: CL_FP_DENORM – denorms are supported CL_FP_INF_NAN – INF and NaNs are supported CL_FP_ROUND_TO_NEAREST – round to nearest even rounding mode supported CL_FP_ROUND_TO_ZERO – round to zero rounding mode supported CL_FP_ROUND_TO_INF – round to +ve and –ve infinity rounding modes supported CP_FP_FMA – IEEE754-2008 fused multiply-add is supported. CL_FP_SOFT_FLOAT – Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. The required minimum half precision floating-point capability as implemented by this extension is CL_FP_ROUND_TO_ZERO or CL_FP_ROUND_TO_NEAREST | CL_FP_INF_NAN.

9.6.10 Relative Error as ULPs

In this section we discuss the maximum relative error defined as *ulp* (units in the last place). If CL_FP_ROUND_TO_NEAREST is supported, the default rounding mode will be round to nearest even; otherwise the default rounding mode will be round to zero. Addition, subtraction, multiplication, fused multiply-add operations on half types are required to be correctly rounded using the current rounding mode. Conversions to half floating point format must be correctly rounded using the indicated `convert_` operator rounding mode or the current rounding mode if no rounding mode is specified by the operator, or a C-style cast is used. Conversions from half to integer format shall correctly round using the indicated `convert_` operator rounding mode, or towards zero if not rounding modeis specified by the operator or a C-style cast is used. All conversions from half to floating point formats are exact.

The following table describes the minimum accuracy of half precision floating-point arithmetic operations given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

Function	Min Accuracy - ULP values[65]
x* + *y	Correctly rounded
x* – *y	Correctly rounded
x* * *y	Correctly rounded
1.0 / *x*	Correctly rounded
x* / *y	Correctly rounded
acos	<= 2 ulp
acospi	<= 2 ulp
asin	<= 2 ulp
asinpi	<= 2 ulp
atan	<= 2 ulp
atan2	<= 2 ulp
atanpi	<= 2 ulp
atan2pi	<= 2 ulp
acosh	<= 2 ulp
asinh	<= 2 ulp
atanh	<= 2 ulp
cbrt	<= 2 ulp
ceil	Correctly rounded
copysign	0 ulp
cos	<= 2 ulp
cosh	<= 2 ulp
cospi	<= 2 ulp
erfc	<= 4 ulp

---

[65] 0 ulp is used for math functions that do not require rounding.

erf	<= 4 ulp
exp	<= 2 ulp
exp2	<= 2 ulp
exp10	<= 2 ulp
expm1	<= 2 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp
hypot	<= 2 ulp
ilogb	0 ulp
ldexp	Correctly rounded
log	<= 2 ulp
log2	<= 2 ulp
log10	<= 2 ulp
log1p	<= 2 ulp
logb	0 ulp
mad	Any value allowed (infinite ulp)
maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	<= 4 ulp
pown(x, y)	<= 4 ulp
powr(x, y)	<= 4 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	<= 4 ulp
round	Correctly rounded
rsqrt	<=1 ulp
sin	<= 2 ulp
sincos	<= 2 ulp for sine and cosine values
sinh	<= 2 ulp
sinpi	<= 2 ulp
sqrt	Correctly rounded
tan	<= 2 ulp
tanh	<= 2 ulp
tanpi	<= 2 ulp

tgamma	<= 4 ulp
trunc	Correctly rounded

NOTE: *Implementations may perform floating-point operations on* `half` *scalar or vector data types by converting the* `half` *values to single precision floating-point values and performing the operation in single precision floating-point. In this case, the implementation will use the* `half` *scalar or vector data type as a storage only format.*

9.7 Creating CL context from a GL context or share group

9.7.1 Overview

The OpenCL specification in *section 9.8* defines how to share data with texture and buffer objects in a parallel OpenGL implementation, but does not define how the association between an OpenCL context and an OpenGL context or share group is established. This extension defines optional attributes to OpenCL context creation routines which associate a GL context or share group object with a newly created OpenCL context. If this extension is supported by an implementation, the string **cl_khr_gl_sharing** will be present in the CL_PLATFORM_EXTENSIONS or CL_DEVICE_EXTENSIONS string described in *table 4.3*.

An OpenGL implementation supporting buffer objects and sharing of texture and buffer object images with OpenCL is required by this extension.

9.7.2 New Procedures and Functions

cl_int **clGetGLContextInfoKHR** (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*);

9.7.3 New Tokens

Returned by **clCreateContext**, **clCreateContextFromType**, and **clGetGLContextInfoKHR** when an invalid OpenGL context or share group object handle is specified in *properties*:

CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR	-1000

Accepted as the *param_name* argument of **clGetGLContextInfoKHR**:

CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR	0x2006
CL_DEVICES_FOR_GL_CONTEXT_KHR	0x2007

Accepted as an attribute name in the *properties* argument of **clCreateContext** and **clCreateContextFromType**:

CL_GL_CONTEXT_KHR	0x2008

CL_EGL_DISPLAY_KHR	0x2009
CL_GLX_DISPLAY_KHR	0x200A
CL_WGL_HDC_KHR	0x200B
CL_CGL_SHAREGROUP_KHR	0x200C

9.7.4 Additions to Chapter 4 of the OpenCL 1.1 Specification

In *section 4.3*, replace the description of *properties* under **clCreateContext** with:

"*properties* points to an attribute list, which is a array of ordered <attribute name, value> pairs terminated with zero. If an attribute is not specified in *properties*, then its default value (listed in *table 4.4*) is used (it is said to be specified implicitly). If *properties* is NULL or empty (points to a list whose first value is zero), all attributes take on their default values.

Attributes control sharing of OpenCL memory objects with OpenGL buffer, texture, and renderbuffer objects as described in *section 9.8*. Depending on the platform-specific API used to bind OpenGL contexts to the window system, the following attributes may be set to identify an OpenGL context:

- When the CGL binding API is supported, the attribute CL_CGL_SHAREGROUP_KHR should be set to a CGLShareGroup handle to a CGL share group object.

- When the EGL binding API is supported, the attribute CL_GL_CONTEXT_KHR should be set to an EGLContext handle to an OpenGL ES or OpenGL context, and the attribute CL_EGL_DISPLAY_KHR should be set to the EGLDisplay handle of the display used to create the OpenGL ES or OpenGL context.

- When the GLX binding API is supported, the attribute CL_GL_CONTEXT_KHR should be set to a GLXContext handle to an OpenGL context, and the attribute CL_GLX_DISPLAY_KHR should be set to the Display handle of the X Window System display used to create the OpenGL context.

- When the WGL binding API is supported, the attribute CL_GL_CONTEXT_KHR should be set to an HGLRC handle to an OpenGL context, and the attribute CL_WGL_HDC_KHR should be set to the HDC handle of the display used to create the OpenGL context.

Memory objects created in the context so specified may be shared with the specified OpenGL or OpenGL ES context (as well as with any other OpenGL contexts on the share list of that context, according to the description of sharing in the GLX 1.4 and EGL 1.4 specifications, and the WGL documentation for OpenGL implementations on Microsoft Windows), or with the explicitly identified OpenGL share group for CGL. If no OpenGL or OpenGL ES context or share group is specified in the attribute list, then memory objects may not be shared, and calling any of the commands in *section 9.8* will result in a CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR error."

Add to *table 4.4*:

Attribute Name	**Allowed Values (Default value is in bold)**	**Description**
CL_GL_CONTEXT_KHR	**0**, OpenGL context handle	OpenGL context to associated the OpenCL context with
CL_CGL_SHAREGROUP_KHR	**0**, CGL share group handle	CGL share group to associate the OpenCL context with
CL_EGL_DISPLAY_KHR	**EGL_NO_DISPLAY**, EGLDisplay handle	EGLDisplay an OpenGL context was created with respect to
CL_GLX_DISPLAY_KHR	**None**, X handle	X Display an OpenGL context was created with respect to
CL_WGL_HDC_KHR	**0**, HDC handle	HDC an OpenGL context was created with respect to

Table 4.4: *Context creation attributes*

Replace the first error in the list for **clCreateContext** with:

"*errcode_ret* returns CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR if a context was specified by any of the following means:

- A context was specified for an EGL-based OpenGL ES or OpenGL implementation by setting the attributes CL_GL_CONTEXT_KHR and CL_EGL_DISPLAY_KHR.

- A context was specified for a GLX-based OpenGL implementation by setting the attributes CL_GL_CONTEXT_KHR and CL_GLX_DISPLAY_KHR.

- A context was specified for a WGL-based OpenGL implementation by setting the attributes CL_GL_CONTEXT_KHR and CL_WGL_HDC_KHR

and any of the following conditions hold:

- The specified display and context attributes do not identify a valid OpenGL or OpenGL ES context.

- The specified context does not support buffer and renderbuffer objects.

- The specified context is not compatible with the OpenCL context being created (for example, it exists in a physically distinct address space, such as another hardware device; or it does not support sharing data with OpenCL due to implementation restrictions).

errcode_ret returns CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR if a share group was specified for a CGL-based OpenGL implementation by setting the attribute CL_CGL_SHAREGROUP_KHR, and the specified share group does not identify a valid CGL share group object.

errcode_ret returns CL_INVALID_OPERATION if a context was specified as described above and any of the following conditions hold:

- A context or share group object was specified for one of CGL, EGL, GLX, or WGL and the OpenGL implementation does not support that window-system binding API.
- More than one of the attributes CL_CGL_SHAREGROUP_KHR, CL_EGL_DISPLAY_KHR, CL_GLX_DISPLAY_KHR, and CL_WGL_HDC_KHR is set to a non-default value.
- Both of the attributes CL_CGL_SHAREGROUP_KHR and CL_GL_CONTEXT_KHR are set to non-default values.
- Any of the devices specified in the *devices* argument cannot support OpenCL objects which share the data store of an OpenGL object, as described in *section 9.8*.

errcode_ret returns CL_INVALID_PROPERTY if an attribute name other than those specified in *table 4.4* is specified in *properties*."

Replace the description of *properties* under **clCreateContextFromType** with:

"*properties* points to an attribute list whose format and valid contents are identical to the **properties** argument of **clCreateContext**."

Replace the first error in the list for **clCreateContextFromType** with the same two new errors described above for **clCreateContext**.

9.7.5 Additions to section 9.8 of the OpenCL 1.1 Specification

Add new *section 9.8.7* :

"OpenCL device(s) corresponding to an OpenGL context may be queried. Such a device may not always exist (for example, if an OpenGL context is specified on a GPU not supporting OpenCL command queues, but which does support shared CL/GL objects), and if it does exist, may change over time. When such a device does exist, acquiring and releasing shared CL/GL objects may be faster on a command queue corresponding to this device than on command queues corresponding to other devices available to an OpenCL context. To query the currently corresponding device, use the function

cl_int **clGetGLContextInfoKHR** (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*)

properties points to an attribute list whose format and valid contents are identical to the *properties* argument of **clCreateContext**. *properties* must identify a single valid GL context or GL share group object.

param_name is a constant that specifies the GL context information to query, and must be one of the values shown in *table 9.ctxprop*.

param_value is a pointer to memory where the result of the query is returned as described in *table 9.ctxprop*. If *param_value* is NULL, it is ignored.

param_value_size specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type described in *table 9.ctxprop*.

param_value_size_ret returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is NULL, it is ignored.

param_name	**Return Type**	**Information returned in param_value**
CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR	cl_device_id	Return the CL device currently associated with the specified OpenGL context.
CL_DEVICES_FOR_GL_CONTEXT_KHR	cl_device_id[]	List of all CL devices which may be associated with the specified OpenGL context.

Table 9.ctxprop: *GL context information that can be queried with* ***clGetGLContextInfoKHR***

clGetGLContextInfoKHR returns CL_SUCCESS if the function is executed successfully. If no device(s) exist corresponding to *param_name*, the call will not fail, but the value of *param_value_size_ret* will be zero.

clGetGLContextInfoKHR returns CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR if a context was specified by any of the following means:

- A context was specified for an EGL-based OpenGL ES or OpenGL implementation by setting the attributes CL_GL_CONTEXT_KHR and CL_EGL_DISPLAY_KHR.

- A context was specified for a GLX-based OpenGL implementation by setting the attributes CL_GL_CONTEXT_KHR and CL_GLX_DISPLAY_KHR.

- A context was specified for a WGL-based OpenGL implementation by setting the attributes CL_GL_CONTEXT_KHR and CL_WGL_HDC_KHR.

and any of the following conditions hold:

- The specified display and context attributes do not identify a valid OpenGL or OpenGL ES context.

- The specified context does not support buffer and renderbuffer objects.

- The specified context is not compatible with the OpenCL context being created (for example, it exists in a physically distinct address space, such as another hardware device; or it does not support sharing data with OpenCL due to implementation restrictions).

clGetGLContextInfoKHR returns CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR if a share group was specified for a CGL-based OpenGL implementation by setting the attribute CL_CGL_SHAREGROUP_KHR, and the specified share group does not identify a valid CGL share group object.

clGetGLContextInfoKHR returns CL_INVALID_OPERATION if a context was specified as described above and any of the following conditions hold:

- A context or share group object was specified for one of CGL, EGL, GLX, or WGL and the OpenGL implementation does not support that window-system binding API.

- More than one of the attributes CL_CGL_SHAREGROUP_KHR, CL_EGL_DISPLAY_KHR, CL_GLX_DISPLAY_KHR, and CL_WGL_HDC_KHR is set to a non-default value.

- Both of the attributes CL_CGL_SHAREGROUP_KHR and CL_GL_CONTEXT_KHR are set to non-default values.

- Any of the devices specified in the <devices> argument cannot support OpenCL objects which share the data store of an OpenGL object, as described in *section 9.8*.

clGetGLContextInfoKHR returns CL_INVALID_VALUE if an attribute name other than those specified in *table 4.4* is specified in *properties*.

Additionally, **clGetGLContextInfoKHR** returns CL_INVALID_VALUE if *param_name* is not one of the values listed in *table 9.ctxprop*, or if the size in bytes specified by *param_value_size* is less than the size of the return type shown in *table 9.ctxprop*, and *param_value* is not a NULL value, CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device, or CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host."

9.7.6 Issues

1. How should the OpenGL context be identified when creating an associated OpenCL context?

RESOLVED: by using a (display,context handle) attribute pair to identify an arbitrary OpenGL or OpenGL ES context with respect to one of the window-system binding layers EGL, GLX, or WGL, or a share group handle to identify a CGL share group. If a context is specified, it need not be current to the thread calling clCreateContext*.

A previously suggested approach would use a single boolean attribute CL_USE_GL_CONTEXT_KHR to allow creating a context associated with the currently bound OpenGL context. This may still be implemented as a separate extension, and might allow more efficient acquire/release behavior in the special case where they are being executed in the same thread as the bound GL context used to create the CL context.

2. What should the format of an attribute list be?

After considerable discussion, we think we can live with a list of <attribute name,value> pairs terminated by zero. The list is passed as 'cl_context_properties **properties*', where cl_context_properties is typedefed to be 'intptr_t' in cl.h.

This effectively allows encoding all scalar integer, pointer, and handle values in the host API into the argument list and is analogous to the structure and type of EGL attribute lists. NULL attribute lists are also allowed. Again as for EGL, any attributes not explicitly passed in the list will take on a defined default value that does something reasonable.

Experience with EGL, GLX, and WGL has shown attribute lists to be a sufficiently flexible and general mechanism to serve the needs of management calls such as context creation. It is not completely general (encoding floating-point and non-scalar attribute values is not straightforward), and other approaches were suggested such as opaque attribute lists with getter/setter methods, or arrays of varadic structures.

3. What's the behavior of an associated OpenGL or OpenCL context when using resources defined by the other associated context, and that context is destroyed?

RESOLVED: As described in *section 9.8*, OpenCL objects place a reference on the data store underlying the corresponding GL object when they're created. The GL name corresponding to that data store may be deleted, but the data store itself remains so long as any CL object has a reference to it. However, destroying all GL contexts in the share group corresponding to a CL context results in implementation-dependent behavior when using a corresponding CL object, up to and including program termination.

4. How about sharing with D3D?

Sharing between D3D and OpenCL should use the same attribute list mechanism, though obviously with different parameters, and be exposed as a similar parallel OpenCL extension.

There may be an interaction between that extension and this one since it's not yet clear if it will be possible to create a CL context simultaneously sharing GL and D3D objects.

5. Under what conditions will context creation fail due to sharing?

RESOLVED: Several cross-platform failure conditions are described (GL context or CGL share group doesn't exist, GL context doesn't support types of GL objects required by the *section 9.8* interfaces, GL context implementation doesn't allow sharing), but additional failures may result due to implementation-dependent reasons and should be added to this extension as such failures are discovered. Sharing between OpenCL and OpenGL requires integration at the driver internals level.

6. What command queues can **clEnqueueAcquire/ReleaseGLObjects** be placed on?

RESOLVED: All command queues. This restriction is enforced at context creation time. If any device passed to context creation cannot support shared CL/GL objects, context creation will fail with a CL_INVALID_OPERATION error.

7. How can applications determine which command queue to place an Acquire/Release on?

RESOLVED: The **clGetGLContextInfoKHR** returns either the CL device currently corresponding to a specified GL context (typically the display it's running on), or a list of all the CL devices the specified context might run on (potentially useful in multiheaded / "virtual screen" environments). This command is not simply placed in *section 9.8* because it relies on the same property-list method of specifying a GL context introduced by this extension.

If no devices are returned, it means that the GL context exists on an older GPU not capable of running OpenCL, but still capable of sharing objects between GL running on that GPU and CL running elsewhere.

8. What is the meaning of the CL_DEVICES_FOR_GL_CONTEXT_KHR query?

RESOLVED: The list of all CL devices that may ever be associated with a specific GL context. On platforms such as MacOS X, the "virtual screen" concept allows multiple GPUs to back a single virtual display. Similar functionality might be implemented on other windowing systems, such as a transparent heterogenous multiheaded X server. Therefore the exact meaning of this query is interpreted relative to the binding layer API in use.

9) Miscellaneous issues during syncing of version 12 with the OpenCL 1.0 revision 47 spec language and the minor changes made including this extension as section 9.11 of that spec:

- Rev47 spec numbers table 9.ctxprop as "9.7" but this depends on the core spec revision.
- Rev47 spec uses 'cl_context' as the return type for **clGetGLContextInfoKHR** param names, but cl_device_id / cl_device_id[] are the proper types.
- Rev47 spec omits the paragraph describing CL_SUCCESS return from **clGetGLContextInfoKHR**.

9.8 Sharing Memory Objects with OpenGL / OpenGL ES Buffer, Texture and Renderbuffer Objects

This section discusses OpenCL functions that allow applications to use OpenGL buffer, texture and renderbuffer objects as OpenCL memory objects. This allows efficient sharing of data between OpenCL and OpenGL. The OpenCL API may be used to execute kernels that read and/or write memory objects that are also OpenGL objects.

An OpenCL image object may be created from an OpenGL texture or renderbuffer object. An OpenCL buffer object may be created from an OpenGL buffer object.

OpenCL memory objects may be created from OpenGL objects if and only if the OpenCL context has been created from an OpenGL share group object or context. OpenGL share groups and contexts are created using platform specific APIs such as EGL, CGL, WGL, and GLX. On MacOS X, an OpenCL context may be created from an OpenGL share group object using the OpenCL platform extension **cl_apple_gl_sharing**. On other platforms including Microsoft Windows, Linux/Unix and others, an OpenCL context may be created from an OpenGL context using the Khronos platform extension **cl_khr_gl_sharing**. Refer to the platform documentation for your OpenCL implementation, or visit the Khronos Registry at http://www.khronos.org/registry/cl/ for more information.

Any supported OpenGL object defined within the GL share group object, or the share group associated with the GL context from which the CL context is created, may be shared, with the exception of the default OpenGL objects (i.e. objects named zero), which may not be shared.

9.8.1 Lifetime of Shared Objects

An OpenCL memory object created from an OpenGL object (hereinafter refered to as a "shared CL/GL object") remains valid as long as the corresponding GL object has not been deleted. If the GL object is deleted through the GL API (e.g. **glDeleteBuffers**, **glDeleteTextures,** or **glDeleteRenderbuffers**), subsequent use of the CL buffer or image object will result in undefined behavior, including but not limited to possible CL errors and data corruption, but may not result in program termination.

The CL context and corresponding command-queues are dependent on the existence of the GL share group object, or the share group associated with the GL context from which the CL context is created. If the GL share group object or all GL contexts in the share group are destroyed, any use of the CL context or command-queue(s) will result in undefined behavior, which may include program termination. Applications should destroy the CL command-queue(s) and CL context before destroying the corresponding GL share group or contexts

9.8.2 CL Buffer Objects → GL Buffer Objects

The function

```
cl_mem      clCreateFromGLBuffer (cl_context context,
                                  cl_mem_flags flags,
                                  GLuint bufobj,
                                  cl_int *errcode_ret)
```

creates an OpenCL buffer object from an OpenGL buffer object.

context is a valid OpenCL context created from an OpenGL context.

flags is a bit-field that is used to specify usage information. Refer to *table 5.3* for a description of *flags*. Only CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE values specified in *table 5.3* can be used.

bufobj is the name of a GL buffer object. The data store of the GL buffer object must have have been previously created by calling **glBufferData**, although its contents need not be initialized. The size of the data store will be used to determine the size of the CL buffer object.

errcode_ret will return an appropriate error code as described below. If *errcode_ret* is NULL, no error code is returned.

clCreateFromGLBuffer returns a valid non-zero OpenCL buffer object and *errcode_ret* is set to CL_SUCCESS if the buffer object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context or was not created from a GL context.
- CL_INVALID_VALUE if values specified in *flags* are not valid.
- CL_INVALID_GL_OBJECT if *bufobj* is not a GL buffer object or is a GL buffer object but does not have an existing data store.
- 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 size of the GL buffer object data store at the time **clCreateFromGLBuffer** is called will be used as the size of buffer object returned by **clCreateFromGLBuffer**. If the state of a GL buffer object is modified through the GL API (e.g. **glBufferData**) while there exists a corresponding CL buffer object, subsequent use of the CL buffer object will result in undefined behavior.

The **clRetainMemObject** and **clReleaseMemObject** functions can be used to retain and release the buffer object.

9.8.3 CL Image Objects → GL Textures

The function

```
cl_mem      clCreateFromGLTexture2D (cl_context context,
                                     cl_mem_flags flags,
                                     GLenum texture_target,
                                     GLint miplevel,
                                     GLuint texture,
                                     cl_int *errcode_ret)
```

creates an OpenCL 2D image object from an OpenGL 2D texture object, or a single face of an OpenGL cubemap texture object..

context is a valid OpenCL context created from an OpenGL context.

flags is a bit-field that is used to specify usage information. Refer to *table 5.3* for a description of *flags*. Only CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE values specified in *table 5.3* may be used.

texture_target must be one of GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, or GL_TEXTURE_RECTANGLE[66]. *texture_target* is used only to define the image type of *texture*. No reference to a bound GL texture object is made or implied by this parameter.

miplevel is the mipmap level to be used[67].

texture is the name of a GL 2D, cubemap or rectangle texture object. The texture object must be a complete texture as per OpenGL rules on texture completeness. The *texture* format and dimensions defined by OpenGL for the specified *miplevel* of the texture will be used to create the 2D image object. Only GL texture objects with an internal format that maps to appropriate image channel order and data type specified in *tables 5.5* and *5.6* may be used to create a 2D image object.

errcode_ret will return an appropriate error code as described below. If *errcode_ret* is NULL, no error code is returned.

---

[66] Requires OpenGL 3.1. Alternatively, GL_TEXTURE_RECTANGLE_ARB may be specified if the OpenGL extension **GL_ARB_texture_rectangle** is supported.

[67] Implementations may return CL_INVALID_OPERATION for miplevel values > 0.

clCreateFromGLTexture2D returns a valid non-zero OpenCL image object and *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context or was not created from a GL context.
- CL_INVALID_VALUE if values specified in *flags* are not valid or if value specified in *texture_target* is not one of the values specified in the description of *texture_target*.
- CL_INVALID_MIP_LEVEL if *miplevel* is less than the value of $level_{base}$ (for OpenGL implementations) or zero (for OpenGL ES implementations); or greater than the value of q (for both OpenGL and OpenGL ES). $level_{base}$ and q are defined for the texture in *section 3.8.10* (Texture Completeness) of the OpenGL 2.1 specification and *section 3.7.10* of the OpenGL ES 2.0.
- CL_INVALID_MIP_LEVEL if *miplevel* is greather than zero and the OpenGL implementation does not support creating from non-zero mipmap levels.
- CL_INVALID_GL_OBJECT if *texture* is not a GL texture object whose type matches *texture_target*, if the specified *miplevel* of *texture* is not defined, or if the width or height of the specified *miplevel* is zero.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if the OpenGL texture internal format does not map to a supported OpenCL image format.
- CL_INVALID_OPERATION if *texture* is a GL texture object created with a border width value greater than zero.
- 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 function

cl_mem **clCreateFromGLTexture3D** (cl_context *context*,
cl_mem_flags *flags*,
GLenum *texture_target*,
GLint *miplevel*,
GLuint *texture*,
cl_int **errcode_ret*)

creates an OpenCL 3D image object from an OpenGL 3D texture object.

context is a valid OpenCL context created from an OpenGL context.

flags is a bit-field that is used to specify usage information. Refer to *table 5.3* for a description of *flags*. Only CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE values specified in *table 5.3* can be used.

texture_target must be GL_TEXTURE_3D. *texture_target* is used only to define the image type of *texture*. No reference to a bound GL texture object is made or implied by this parameter.

miplevel is the mipmap level to be used.

texture is the name of a GL 3D texture object. The texture object must be a complete texture as per OpenGL rules on texture completeness. The *texture* format and dimensions defined by OpenGL for the specified *miplevel* of the texture will be used to create the 3D image object. Only GL texture objects with an internal format that maps to appropriate image channel order and data type specified in *tables 5.5* and *5.6* can be used to create the 3D image object.

errcode_ret will return an appropriate error code as described below. If *errcode_ret* is NULL, no error code is returned.

clCreateFromGLTexture3D returns a valid non-zero image object and *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context or was not created from a GL context.

- CL_INVALID_VALUE if values specified in *flags* are not valid or if value specified in *texture_target* is not one of the values specified in the description of *texture_target*.

- CL_INVALID_MIP_LEVEL if *miplevel* is less than the value of $level_{base}$ (for OpenGL implementations) or zero (for OpenGL ES implementations); or greater than the value of *q* (for both OpenGL and OpenGL ES). $level_{base}$ and *q* are defined for the texture in *section 3.8.10* (Texture Completeness) of the OpenGL 2.1 specification and *section 3.7.10* of the OpenGL ES 2.0.

- CL_INVALID_MIP_LEVEL if *miplevel* is greather than zero and the OpenGL implementation does not support creating from non-zero mipmap levels.

- CL_INVALID_GL_OBJECT if *texture* is not a GL texture object whose type matches *texture_target*, if the specified *miplevel* of *texture* is not defined, or if the width, height or depth of the specified *miplevel* is zero.

- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if the OpenGL texture internal format does not map to a supported OpenCL image format.

- CL_INVALID_OPERATION if *texture* is a GL texture object created with a border width value greater than zero.
- 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.

If the state of a GL texture object is modified through the GL API (e.g. **glTexImage2D**, **glTexImage3D** or the values of the texture parameters GL_TEXTURE_BASE_LEVEL or GL_TEXTURE_MAX_LEVEL are modified) while there exists a corresponding CL image object, subsequent use of the CL image object will result in undefined behavior.

The **clRetainMemObject** and **clReleaseMemObject** functions can be used to retain and release the image objects.

9.8.3.1 List of OpenGL and corresponding OpenCL Image Formats

Table 9.4 describes the list of GL texture internal formats and the corresponding CL image formats. If a GL texture object with an internal format from *table 9.4* is successfully created by OpenGL, then there is guaranteed to be a mapping to one of the corresponding CL image format(s) in that table. Texture objects created with other OpenGL internal formats may (but are not guaranteed to) have a mapping to a CL image format; if such mappings exist, they are guaranteed to preserve all color components, data types, and at least the number of bits/component actually allocated by OpenGL for that format.

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

Table 9.4 *Mapping of GL internal format to CL image format*

9.8.4 CL Image Objects → GL Renderbuffers

The function

```
cl_mem      clCreateFromGLRenderbuffer (cl_context context,
                                        cl_mem_flags flags,
                                        GLuint renderbuffer,
                                        cl_int *errcode_ret)
```

creates an OpenCL 2D image object from an OpenGL renderbuffer object.

context is a valid OpenCL context created from an OpenGL context.

flags is a bit-field that is used to specify usage information. Refer to *table 5.3* for a description of *flags*. Only CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE values specified in *table 5.3* can be used.

renderbuffer is the name of a GL renderbuffer object. The renderbuffer storage must be specified before the image object can be created. The *renderbuffer* format and dimensions defined by OpenGL will be used to create the 2D image object. Only GL renderbuffers with internal formats that maps to appropriate image channel order and data type specified in *tables 5.5* and *5.6* can be used to create the 2D image object.

errcode_ret will return an appropriate error code as described below. If *errcode_ret* is NULL, no error code is returned.

clCreateFromGLRenderbuffer returns a valid non-zero OpenCL image object and *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context or was not created from a GL context.
- CL_INVALID_VALUE if values specified in *flags* are not valid.
- CL_INVALID_GL_OBJECT if *renderbuffer* is not a GL renderbuffer object or if the width or height of *renderbuffer* is zero.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if the OpenGL renderbuffer internal format does not map to a supported OpenCL image format.
- CL_INVALID_OPERATION if *renderbuffer* is a multi-sample GL renderbuffer 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.

If the state of a GL renderbuffer object is modified through the GL API (i.e. changes to the dimensions or format used to represent pixels of the GL renderbuffer using appropriate GL API calls such as **glRenderbufferStorage**) while there exists a corresponding CL image object, subsequent use of the CL image object will result in undefined behavior.

The **clRetainMemObject** and **clReleaseMemObject** functions can be used to retain and release the image objects.

Table 9.4 describes the list of GL renderbuffer internal formats and the corresponding CL image formats. If a GL renderbuffer object with an internal format from *table 9.4* is successfully created by OpenGL, then there is guaranteed to be a mapping to one of the corresponding CL image format(s) in that table. Renderbuffer objects created with other OpenGL internal formats may (but are not guaranteed to) have a mapping to a CL image format; if such mappings exist, they are guaranteed to preserve all color components, data types, and at least the number of bits/component actually allocated by OpenGL for that format.

9.8.5 Querying GL object information from a CL memory object

The OpenGL object used to create the OpenCL memory object and information about the object type i.e. whether it is a texture, renderbuffer or buffer object can be queried using the following function.

cl_int **clGetGLObjectInfo** (cl_mem *memobj*,
cl_gl_object_type **gl_object_type*,
GLuint **gl_object_name*)

gl_object_type returns the type of GL object attached to *memobj* and can be CL_GL_OBJECT_BUFFER, CL_GL_OBJECT_TEXTURE2D, CL_GL_OBJECT_TEXTURE3D, or CL_GL_OBJECT_RENDERBUFFER. If *gl_object_type* is NULL, it is ignored

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

clGetGLObjectInfo returns CL_SUCCESS if the call was executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_MEM_OBJECT if *memobj* is not a valid OpenCL memory object.

- CL_INVALID_GL_OBJECT if there is no GL object associated with *memobj*.

- 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 function

cl_int **clGetGLTextureInfo** (cl_mem *memobj*,
cl_gl_texture_info *param_name*,
size_t *param_value_size*,
void **param_value*,
size_t **param_value_size_ret*)

returns additional information about the GL texture object associated with *memobj*.

param_name specifies what additional information about the GL texture object associated with *memobj* to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetGLTextureInfo** is described in *table 9.5* below.

param_value is a pointer to memory where the result being queried is returned. If *param_value* is NULL, it is ignored.

param_value_size is used to specify the size in bytes of memory pointed to by *param_value*. This size must be >= size of return type as described in *table 9.5* below.

param_value_size_ret returns the actual size in bytes of data copied to *param_value*. If *param_value_size_ret* is NULL, it is ignored.

cl_gl_texture_info	Return Type	Info. returned in *param_value*
CL_GL_TEXTURE_TARGET	GLenum	The *texture_target* argument specified in **clCreateFromGLTexture2D**, **clCreateFromGLTexture3D**.
CL_GL_MIPMAP_LEVEL	GLint	The *miplevel* argument specified in **clCreateFromGLTexture2D**, **clCreateFromGLTexture3D**.

Table *9.5* *List of supported param_names by clGetGLTextureInfo*

clGetGLTextureInfo 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 OpenCL memory object.

- CL_INVALID_GL_OBJECT if there is no GL texture object associated with *memobj*.
- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 9.5* and *param_value* is not NULL, or if *param_value* and *param_value_size_ret* are 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.

9.8.6 Sharing memory objects that map to GL objects between GL and CL contexts

The function

cl_int **clEnqueueAcquireGLObjects** (cl_command_queue *command_queue*,
cl_uint *num_objects*.
const cl_mem **mem_objects,*
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

is used to acquire OpenCL memory objects that have been created from OpenGL objects. These objects need to be acquired before they can be used by any OpenCL commands queued to a command-queue. The OpenGL objects are acquired by the OpenCL context associated with *command_queue* and can therefore be used by all command-queues associated with the OpenCL context.

command_queue is a valid command-queue. All devices used to create the OpenCL context associated with *command_queue* must support acquiring shared CL/GL objects. This constraint is enforced at context creation time.

num_objects is the number of memory objects to be acquired in *mem_objects*.

mem_objects is a pointer to a list of CL memory objects that correspond to GL objects.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.

event returns an event object that identifies this command and can be used to query or queue a wait for the 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.

clEnqueueAcquireGLObjects returns CL_SUCCESS if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is NULL the function does nothing and returns CL_SUCCESS. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *num_objects* is zero and *mem_objects* is not a NULL value or if *num_objects* > 0 and *mem_objects* is NULL.
- CL_INVALID_MEM_OBJECT if memory objects in *mem_objects* are not valid OpenCL memory objects.
- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* was not created from an OpenGL context
- CL_INVALID_GL_OBJECT if memory objects in *mem_objects* have not been created from a GL object(s).
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- 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 function

cl_int **clEnqueueReleaseGLObjects** (cl_command_queue *command_queue*,
cl_uint *num_objects*.
const cl_mem **mem_objects*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

is used to release OpenCL memory objects that have been created from OpenGL objects. These objects need to be released before they can be used by OpenGL. The OpenGL objects are released by the OpenCL context associated with *command_queue*.

num_objects is the number of memory objects to be released in *mem_objects*.

mem_objects is a pointer to a list of CL memory objects that correpond to GL objects.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.

event returns an event object that identifies this particular read / write command and can be used to query or queue a wait for the 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.

clEnqueueReleaseGLObjects returns CL_SUCCESS if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is NULL the function does nothing and returns CL_SUCCESS. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *num_objects* is zero and *mem_objects* is not a NULL value or if *num_objects* > 0 and *mem_objects* is NULL.

- CL_INVALID_MEM_OBJECT if memory objects in *mem_objects* are not valid OpenCL memory objects.

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- CL_INVALID_CONTEXT if context associated with *command_queue* was not created from an OpenGL context

- CL_INVALID_GL_OBJECT if memory objects in *mem_objects* have not been created from a GL object(s).

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.

- 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.

9.8.6.1 Synchronizing OpenCL and OpenGL Access to Shared Objects

In order to ensure data integrity, the application is responsible for synchronizing access to shared CL/GL objects by their respective APIs. Failure to provide such synchronization may result in race conditions and other undefined behavior including non-portability between implementations.

Prior to calling **clEnqueueAcquireGLObjects**, the application must ensure that any pending GL operations which access the objects specified in *mem_objects* have completed. This may be accomplished portably by issuing and waiting for completion of a **glFinish** command on all GL contexts with pending references to these objects. Implementations may offer more efficient synchronization methods; for example on some platforms calling **glFlush** may be sufficient, or synchronization may be implicit within a thread, or there may be vendor-specific extensions that enable placing a fence in the GL command stream and waiting for completion of that fence in the CL command queue. Note that no synchronization methods other than **glFinish** are portable between OpenGL implementations at this time.

Similarly, after calling **clEnqueueReleaseGLObjects**, the application is responsible for ensuring that any pending OpenCL operations which access the objects specified in *mem_objects* have completed prior to executing subsequent GL commands which reference these objects. This may be accomplished portably by calling **clWaitForEvents** with the event object returned by **clEnqueueReleaseGLObjects,** or by calling **clFinish**. As above, some implementations may offer more efficient methods.

The application is responsible for maintaining the proper order of operations if the CL and GL contexts are in separate threads.

If a GL context is bound to a thread other than the one in which **clEnqueueReleaseGLObjects** is called, changes to any of the objects in *mem_objects* may not be visible to that context without additional steps being taken by the application. For an OpenGL 3.1 (or later) context, the requirements are described in Appendix G ("Shared Objects and Multiple Contexts") of the OpenGL 3.1 Specification. For prior versions of OpenGL, the requirements are implementation-dependent.

Attempting to access the data store of an OpenGL object after it has been acquired by OpenCL and before it has been released will result in undefined behavior. Similarly, attempting to access a shared CL/GL object from OpenCL before it has been acquired by the OpenCL command queue, or after it has been released, will result in undefined behavior.

9.9 Creating CL event objects from GL sync objects

9.9.1 Overview

This extension allows creating OpenCL event objects linked to OpenGL fence sync objects, potentially improving efficiency of sharing images and buffers between the two APIs. The companion **GL_ARB_cl_event** extension provides the complementary functionality of creating an OpenGL sync object from an OpenCL event object.

In addition, this extension modifies the behavior of **clEnqueueAcquireGLObjects** and **clEnqueueReleaseGLObjects** to implicitly guarantee synchronization with an OpenGL context bound in the same thread as the OpenCL context.

If this extension is supported by an implementation, the string **cl_khr_gl_event** will be present in the CL_PLATFORM_EXTENSIONS or CL_DEVICE_EXTENSIONS string described in *table 4.3*.

9.9.2 New Procedures and Functions

```
cl_event    clCreateEventFromGLsyncKHR (cl_context context,
                                        GLsync sync,
                                        cl_int *errcode_ret);
```

9.9.3 New Tokens

Returned by **clGetEventInfo** when *param_name* is CL_EVENT_COMMAND_TYPE:

CL_COMMAND_GL_FENCE_SYNC_OBJECT_KHR 0x200D

9.9.4 Additions to Chapter 5 of the OpenCL 1.1 Specification

Add following to the fourth paragraph of *section 5.9* (prior to the description of **clWaitForEvents**):

"Event objects can also be used to reflect the status of an OpenGL sync object. The sync object in turn refers to a fence command executing in an OpenGL command stream. This provides another method of coordinating sharing of buffers and images between OpenGL and OpenCL (see *section 9.8.6.1*)."

Add CL_COMMAND_GL_FENCE_SYNC_OBJECT_KHR to the valid *param_value* values returned by **clGetEventInfo** for *param_name* CL_EVENT_COMMAND_TYPE (in the second row and third column of *table 5.15*).

Add new *subsection 5.9.1*:

"5.9.1 Linking Event Objects to OpenGL Synchronization Objects

An event object may be created by linking to an OpenGL **sync object**. Completion of such an event object is equivalent to waiting for completion of the fence command associated with the linked GL sync object.

The function

```
cl_event    clCreateEventFromGLsyncKHR (cl_context context,
                                        GLsync sync,
                                        cl_int *errcode_ret)
```

creates a linked event object.

context is a valid OpenCL context created from an OpenGL context or share group, using the **cl_khr_gl_sharing** extension.

sync is the name of a sync object in the GL share group associated with *context*.

clCreateEventFromGLsyncKHR returns a valid OpenCL event object and *errcode_ret* is set to CL_SUCCESS if the event object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context, or was not created from a GL context.
- CL_INVALID_GL_OBJECT if *sync* is not the name of a sync object in the GL share group associated with *context*.

The parameters of an event object linked to a GL sync object will return the following values when queried with **clGetEventInfo**:

- The CL_EVENT_COMMAND_QUEUE of a linked event is NULL, because the event is not associated with any OpenCL command queue.
- The CL_EVENT_COMMAND_TYPE of a linked event is CL_COMMAND_GL_FENCE_SYNC_OBJECT_KHR, indicating that the event is associated with a GL sync object, rather than an OpenCL command.
- The CL_EVENT_COMMAND_EXECUTION_STATUS of a linked event is either

CL_SUBMITTED, indicating that the fence command associated with the sync object has not yet completed, or CL_COMPLETE, indicating that the fence command has completed.

clCreateEventFromGLsyncKHR performs an implicit **clRetainEvent** on the returned event object. Creating a linked event object also places a reference on the linked GL sync object. When the event object is deleted, the reference will be removed from the GL sync object.

Events returned from **clCreateEventFromGLsyncKHR** may only be consumed by **clEnqueueAcquireGLObjects**. Passing such events to any other CL API will generate a CL_INVALID_EVENT error."

9.9.5 Additions to Chapter 9 of the OpenCL 1.1 Specification

Add following the paragraph describing parameter *event* to **clEnqueueAcquireGLObjects**:

"If an OpenGL context is bound to the current thread, then any OpenGL commands which

1. affect or access the contents of a memory object listed in the *mem_objects* list, and
2. were issued on that OpenGL context prior to the call to **clEnqueueAcquireGLObjects**

will complete before execution of any OpenCL commands following the **clEnqueueAcquireGLObjects** which affect or access any of those memory objects. If a non-NULL *event* object is returned, it will report completion only after completion of such OpenGL commands."

Add following the paragraph describing parameter *event* to **clEnqueueReleaseGLObjects**:

"If an OpenGL context is bound to the current thread, then then any OpenGL commands which

1. affect or access the contents of the memory objects listed in the *mem_objects* list, and
2. are issued on that context after the call to **clEnqueueReleaseGLObjects**

will not execute until after execution of any OpenCL commands preceding the **clEnqueueReleaseGLObjects** which affect or access any of those memory objects. If a non-NULL *event* object is returned, it will report completion before execution of such OpenGL commands."

Replace the second paragraph of *section 9.8.6.1* (Synchronizing OpenCL and OpenGL Access to Shared Objects) with:

"Prior to calling **clEnqueueAcquireGLObjects**, the application must ensure that any pending OpenGL operations which access the objects specified in *mem_objects* have completed.

If the **cl_khr_gl_event** extension is supported, then the OpenCL implementation will ensure that any such pending OpenGL operations are complete for an OpenGL context bound to the same thread as the OpenCL context. This is referred to as *implicit synchronization*.

If the **cl_khr_gl_event** extension is supported and the OpenGL context in question supports fence sync objects, completion of OpenGL commands may also be determined by placing a GL fence command after those commands using **glFenceSync**, creating an event from the resulting GL sync object using **clCreateEventFromGLsyncKHR**, and determining completion of that event object via **clEnqueueAcquireGLObjects**. This method may be considerably more efficient than calling **glFinish**, and is referred to as *explicit synchronization*. Explicit synchronization is most useful when an OpenGL context bound to another thread is accessing the memory objects.

If the **cl_khr_gl_event** extension is not supported, completion of OpenGL commands may be determined by issuing and waiting for completion of a **glFinish** command on all OpenGL contexts with pending references to these objects. Some implementations may offer other efficient synchronization methods. If such methods exist they will be described in platform-specific documentation.

Note that no synchronization method other than **glFinish** is portable between all OpenGL implementations and all OpenCL implementations. While this is the only way to ensure completion that is portable to all platforms, **glFinish** is an expensive operation and its use should be avoided if the **cl_khr_gl_event** extension is supported on a platform."

9.9.6 Issues

1) How are references between CL events and GL syncs handled?

PROPOSED: The linked CL event places a single reference on the GL sync object. That reference is removed when the CL event is deleted. A more expensive alternative would be to reflect changes in the CL event reference count through to the GL sync.

2) How are linkages to synchronization primitives in other APIs handled?

UNRESOLVED. We will at least want to have a way to link events to EGL sync objects. There is probably no analogous DX concept. There would be an entry point for each type of synchronization primitive to be linked to, such as clCreateEventFromEGLSyncKHR.

An alternative is a generic clCreateEventFromExternalEvent taking an attribute list. The attribute list would include information defining the type of the external primitive and additional information (GL sync object handle, EGL display and sync object handle, etc.) specific to that type. This allows a single entry point to be reused.

These will probably be separate extensions following the API proposed here.

3) Should the CL_EVENT_COMMAND_TYPE correspond to the type of command (fence) or the type of the linked sync object?

PROPOSED: To the type of the linked sync object.

4) Should we support both explicit and implicit synchronization?

PROPOSED: Yes. Implicit synchronization is suitable when GL and CL are executing in the same application thread. Explicit synchronization is suitable when they are executing in different threads but the expense of glFinish is too high.

5) Should this be a platform or device extension?

PROPOSED: Platform extension. This may result in considerable under-the-hood work to implement the sync->event semantics using only the public GL API, however, when multiple drivers and devices with different GL support levels coexist in the same runtime.

6) Where can events generated from GL syncs be usable?

PROPOSED: Only with clEnqueueAcquireGLObjects, and attempting to use such an event elsewhere will generate an error. There is no apparent use case for using such events elsewhere, and possibly some cost to supporting it, balanced by the cost of checking the source of events in all other commands accepting them as parameters.

9.10 Sharing Memory Objects with Direct3D 10

9.10.1 Overview

The goal of this extension is to provide interoperability between OpenCL and Direct3D 10. This is designed to function analogously to the OpenGL interoperability as defined in *sections 9.7* and *9.8*. If this extension is supported by an implementation, the string **cl_khr_d3d10_sharing** will be present in the CL_PLATFORM_EXTENSIONS or CL_DEVICE_EXTENSIONS string described in *table 4.3*.

9.10.2 Header File

As currently proposed the interfaces for this extension would be provided in `cl_d3d10.h`.

9.10.3 New Procedures and Functions

cl_int **clGetDeviceIDsFromD3D10KHR** (cl_platform_id *platform*,
cl_d3d10_device_source_khr *d3d_device_source*,
void **d3d_object*,
cl_d3d10_device_set_khr *d3d_device_set*,
cl_uint *num_entries*,
cl_device_id **devices*,
cl_uint **num_devices*)

cl_mem **clCreateFromD3D10BufferKHR** (cl_context *context*,
cl_mem_flags *flags*,
ID3D10Buffer **resource*,
cl_int **errcode_ret*)

cl_mem **clCreateFromD3D10Texture2DKHR** (cl_context *context*,
cl_mem_flags *flags*,
ID3D10Texture2D **resource*,
UINT *subresource*,
cl_int **errcode_ret*)

cl_mem **clCreateFromD3D10Texture3DKHR** (cl_context *context*,
cl_mem_flags *flags*,
ID3D10Texture3D **resource*,
UINT *subresource*,
cl_int **errcode_ret*)

```
cl_int  clEnqueueAcquireD3D10ObjectsKHR (cl_command_queue command_queue,
                                         cl_uint num_objects,
                                         const cl_mem *mem_objects,
                                         cl_uint num_events_in_wait_list,
                                         const cl_event *event_wait_list,
                                         cl_event *event)

cl_int  clEnqueueReleaseD3D10ObjectsKHR (cl_command_queue command_queue,
                                         cl_uint num_objects,
                                         const cl_mem *mem_objects,
                                         cl_uint num_events_in_wait_list,
                                         const cl_event *event_wait_list,
                                         cl_event *event)
```

9.10.4 New Tokens

Accepted as a Direct3D 10 device source in the *d3d_device_source* parameter of **clGetDeviceIDsFromD3D10KHR**:

CL_D3D10_DEVICE_KHR	0x4010
CL_D3D10_DXGI_ADAPTER_KHR	0x4011

Accepted as a set of Direct3D 10 devices in the *d3d_device_set* parameter of **clGetDeviceIDsFromD3D10KHR**:

CL_PREFERRED_DEVICES_FOR_D3D10_KHR	0x4012
CL_ALL_DEVICES_FOR_D3D10_KHR	0x4013

Accepted as a property name in the *properties* parameter of **clCreateContext** and **clCreateContextFromType**:

CL_CONTEXT_D3D10_DEVICE_KHR	0x4014

Accepted as a property name in the *param_name* parameter of **clGetContextInfo**:

CL_CONTEXT_D3D10_PREFER_SHARED_RESOURCES_KHR	0x402C

Accepted as the property being queried in the *param_name* parameter of **clGetMemObjectInfo**:

CL_MEM_D3D10_RESOURCE_KHR	0x4015

Accepted as the property being queried in the *param_name* parameter of **clGetImageInfo**:

CL_IMAGE_D3D10_SUBRESOURCE_KHR	0x4016

Returned in the *param_value* parameter of **clGetEventInfo** when *param_name* is CL_EVENT_COMMAND_TYPE:

CL_COMMAND_ACQUIRE_D3D10_OBJECTS_KHR	0x4017
CL_COMMAND_RELEASE_D3D10_OBJECTS_KHR	0x4018

Returned by **clCreateContext** and **clCreateContextFromType** if the Direct3D 10 device specified for interoperability is not compatible with the devices against which the context is to be created:

CL_INVALID_D3D10_DEVICE_KHR	-1002

Returned by **clCreateFromD3D10BufferKHR** when *resource* is not a Direct3D 10 buffer object, and by **clCreateFromD3D10Texture2DKHR** and **clCreateFromD3D10Texture3DKHR** when *resource* is not a Direct3D 10 texture object.

CL_INVALID_D3D10_RESOURCE_KHR	-1003

Returned by **clEnqueueAcquireD3D10ObjectsKHR** when any of *mem_objects* are currently acquired by OpenCL

CL_D3D10_RESOURCE_ALREADY_ACQUIRED_KHR	-1004

Returned by **clEnqueueReleaseD3D10ObjectsKHR** when any of *mem_objects* are not currently acquired by OpenCL

CL_D3D10_RESOURCE_NOT_ACQUIRED_KHR	-1005

9.10.5 Additions to Chapter 4 of the OpenCL 1.1 Specification

In *section 4.3*, replace the description of *properties* under **clCreateContext** with:

"*properties* specifies a list of context property names and their corresponding values. Each property is followed immediately by the corresponding desired value. The list is terminated with zero. If a property is not specified in *properties*, then its default value (listed in *table 4.4*) is used (it is said to be specified implicitly). If *properties* is NULL or empty (points to a list whose first value is zero), all attributes take on their default values."

Add the following to *table 4.4*:

cl_context_properties enum	Property value	Description
CL_CONTEXT_D3D10_DEVICE_KHR	ID3D10Device *	Specifies the ID3D10Device * to

		use for Direct3D 10 interoperability. The default value is NULL.

Add to the list of errors for **clCreateContext**:

- CL_INVALID_D3D10_DEVICE_KHR if the value of the property CL_CONTEXT_D3D10_DEVICE_KHR is non-NULL and does not specify a valid Direct3D 10 device with which the *cl_device_ids* against which this context is to be created may interoperate.

- CL_INVALID_OPERATION if Direct3D 10 interoperability is specified by setting CL_INVALID_D3D10_DEVICE_KHR to a non-NULL value, and interoperability with another graphics API is also specified."

Add to the list of errors for **clCreateContextFromType** the same new errors described above for **clCreateContext**.

Add the following row to *table 4.5*:

cl_context_info	**Return Type**	**Information returned in param_value**
CL_CONTEXT_D3D10_PREFER _SHARED_RESOURCES_KHR_	cl_bool	Returns CL_TRUE if Direct3D 10 resources created as shared by setting *MiscFlags* to include D3D10_RESOURCE_MISC_SHARED will perform faster when shared with OpenCL, compared with resources which have not set this flag. Otherwise returns CL_FALSE.

9.10.6 Additions to Chapter 5 of the OpenCL 1.1 Specification

Add to the list of errors for **clGetMemObjectInfo**:

- CL_INVALID_D3D10_RESOURCE_KHR if *param_name* is CL_MEM_D3D10_RESOURCE_KHR and *memobj* was not created by the function **clCreateFromD3D10BufferKHR**, **clCreateFromD3D10Texture2DKHR**, or **clCreateFromD3D10Texture3DKHR**."

Extend *table 5.9* to include the following entry.

cl_mem_info	Return type	Info. returned in *param_value*
CL_MEM_D3D10_ RESOURCE_KHR	ID3D10Resource *	If *memobj* was created using **clCreateFromD3D10BufferKHR**, **clCreateFromD3D10Texture2DKHR**, or **clCreateFromD3D10Texture3DKHR**, returns the *resource* argument specified when *memobj* was created.

Add to the list of errors for **clGetImageInfo**:

- CL_INVALID_D3D10_RESOURCE_KHR if *param_name* is CL_MEM_D3D10_SUBRESOURCE_KHR and *image* was not created by the function **clCreateFromD3D10Texture2DKHR**, or **clCreateFromD3D10Texture3DKHR**."

Extend *table 5.8* to include the following entry.

cl_image_info	Return type	Info. returned in *param_value*
CL_MEM_D3D10_ SUBRESOURCE_KHR	ID3D10Resource *	If *image* was created using **clCreateFromD3D10Texture2DKHR**, or **clCreateFromD3D10Texture3DKHR**, returns the *subresource* argument specified when *image* was created.

Add to *table 5.15* in the **Info returned in <param_value>** column for *cl_event_info* = CL_EVENT_COMMAND_TYPE:

CL_COMMAND_ACQUIRE_D3D10_OBJECTS_KHR
CL_COMMAND_RELEASE_D3D10_OBJECTS_KHR

9.10.7 Sharing Memory Objects with Direct3D 10 Resources

This section discusses OpenCL functions that allow applications to use Direct3D 10 resources as OpenCL memory objects. This allows efficient sharing of data between OpenCL and Direct3D 10. The OpenCL API may be used to execute kernels that read and/or write memory objects that are also Direct3D 10 resources. An OpenCL image object may be created from a Direct3D 10 texture resource. An OpenCL buffer object may be created from a Direct3D 10 buffer resource. OpenCL memory objects may be created from Direct3D 10 objects if and only if the OpenCL context has been created from a Direct3D 10 device.

9.10.7.1 Querying OpenCL Devices Corresponding to Direct3D 10 Devices

The OpenCL devices corresponding to a Direct3D 10 device may be queried. The OpenCL devices corresponding to a DXGI adapter may also be queried. The OpenCL devices corresponding to a Direct3D 10 device will be a subset of the OpenCL devices corresponding to the DXGI adapter against which the Direct3D 10 device was created.

The OpenCL devices corresponding to a Direct3D 10 device or a DXGI device may be queried using the function

cl_int **clGetDeviceIDsFromD3D10KHR** (cl_platform_id *platform*,
cl_d3d10_device_source_khr *d3d_device_source*,
void **d3d_object*,
cl_d3d10_device_set_khr *d3d_device_set*,
cl_uint *num_entries*,
cl_device_id **devices*,
cl_uint **num_devices*)

platform refers to the platform ID returned by **clGetPlatformIDs**.

d3d_device_source specifies the type of *d3d_object*, and must be one of the values shown in *table 9.10.1*.

d3d_object specifies the object whose corresponding OpenCL devices are being queried. The type of *d3d_object* must be as specified in *table 9.10.1*.

d3d_device_set specifies the set of devices to return, and must be one of the values shown in *table 9.10.2*.

num_entries is the number of cl_device_id entries that can be added to *devices*. If *devices* is not NULL then *num_entries* must be greater than zero.

devices returns a list of OpenCL devices found. The cl_device_id values returned in *devices* can be used to identify a specific OpenCL device. If *devices* is NULL, this argument is ignored. The number of OpenCL devices returned is the minimum of the value specified by *num_entries* and the number of OpenCL devices corresponding to *d3d_object*.

num_devices returns the number of OpenCL devices available that correspond to *d3d_object*. If *num_devices* is NULL, this argument is ignored.

clGetDeviceIDsFromD3D10KHR returns CL_SUCCESS if the function is executed successfully. Otherwise it may return

- CL_INVALID_PLATFORM if *platform* is not a valid platform.

- CL_INVALID_VALUE if *d3d_device_source* is not a valid value, *d3d_device_set* is not a valid value, *num_entries* is equal to zero and *devices* is not NULL, or if both *num_devices* and *devices* are NULL.

- CL_DEVICE_NOT_FOUND if no OpenCL devices that correspond to *d3d_object* were found.

cl_d3d_device_source_khr	**Type of *d3d_object***
CL_D3D10_DEVICE_KHR	ID3D10Device *
CL_D3D10_DXGI_ADAPTER_KHR	IDXGIAdapter *

Table 9.10.1 *Types used to specify the object whose corresponding OpenCL devices are being queried by* ***clGetDeviceIDsFromD3D10KHR***

cl_d3d_device_set_khr	**Devices returned in *devices***
CL_PREFERRED_DEVICES_FOR_D3D10_KHR	The OpenCL devices associated with the specified Direct3D object.
CL_ALL_DEVICES_FOR_D3D10_KHR	All OpenCL devices which may interoperate with the specified Direct3D object. Performance of sharing data on these devices may be considerably less than on the preferred devices.

Table 9.10.2 *Sets of devices queriable using* ***clGetDeviceIDsFromD3D10KHR***

9.10.7.2 Lifetime of Shared Objects

An OpenCL memory object created from a Direct3D 10 resource remains valid as long as the corresponding Direct3D 10 resource has not been deleted. If the Direct3D 10 resource is deleted through the Direct3D 10 API, subsequent use of the OpenCL memory object will result in undefined behavior, including but not limited to possible OpenCL errors, data corruption, and program termination.

The successful creation of a cl_context against a Direct3D 10 device specified via the context create parameter CL_CONTEXT_D3D10_DEVICE_KHR will increment the internal Direct3D reference count on the specified Direct3D 10 device. The internal Direct3D reference count on that Direct3D 10 device will be decremented when the OpenCL reference count on the returned OpenCL context drops to zero.

The OpenCL context and corresponding command-queues are dependent on the existence of the Direct3D 10 device from which the OpenCL context was created. If the Direct3D 10 device is deleted through the Direct3D 10 API, subsequent use of the OpenCL context will result in undefined behavior, including but not limited to possible OpenCL errors, data corruption, and program termination.

9.10.7.3 Sharing Direct3D 10 Buffer Resources as OpenCL Buffer Objects

The function

```
cl_mem      clCreateFromD3D10BufferKHR (cl_context context,
                                        cl_mem_flags flags,
                                        ID3D10Buffer *resource,
                                        cl_int *errcode_ret)
```

creates an OpenCL buffer object from a Direct3D 10 buffer.

context is a valid OpenCL context created from a Direct3D 10 device.

flags is a bit-field that is used to specify usage information. Refer to table 5.3 for a description of *flags*. Only CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE values specified in *table 5.3* can be used.

resource is a pointer to the Direct3D 10 buffer to share.

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

clCreateFromD3D10BufferKHR returns a valid non-zero OpenCL buffer object and *errcode_ret* is set to CL_SUCCESS if the buffer object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if values specified in *flags* are not valid.
- CL_INVALID_D3D10_RESOURCE_KHR if *resource* is not a Direct3D 10 buffer resource, if *resource* was created with the D3D10_USAGE flag D3D10_USAGE_IMMUTABLE, if a cl_mem from *resource* has already been created using **clCreateFromD3D10BufferKHR**, or if *context* was not created against the same Direct3D 10 device from which *resource* was created.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The size of the returned OpenCL buffer object is the same as the size of *resource*. This call will increment the internal Direct3D reference count on *resource*. The internal Direct3D reference count on *resource* will be decremented when the OpenCL reference count on the returned

OpenCL memory object drops to zero.

9.10.7.4 Sharing Direct3D 10 Texture and Resources as OpenCL Image Objects

The function

```
cl_mem    clCreateFromD3D10Texture2DKHR (cl_context context,
                                         cl_mem_flags flags,
                                         ID3D10Texture2D *resource,
                                         UINT subresource,
                                         cl_int *errcode_ret)
```

creates an OpenCL 2D image object from a subresource of a Direct3D 10 2D texture.

context is a valid OpenCL context created from a Direct3D 10 device.

flags is a bit-field that is used to specify usage information. Refer to table 5.3 for a description of *flags*. Only CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE values specified in *table 5.3* can be used.

resource is a pointer to the Direct3D 10 2D texture to share.

subresource is the subresource of *resource* to share.

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

clCreateFromD3D10Texture2DKHR returns a valid non-zero OpenCL image object and *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.

- CL_INVALID_VALUE if values specified in *flags* are not valid or if *subresource* is not a valid subresource index for *resource*.

- CL_INVALID_D3D10_RESOURCE_KHR if *resource* is not a Direct3D 10 texture resource, if *resource* was created with the D3D10_USAGE flag D3D10_USAGE_IMMUTABLE, if *resource* is a multisampled texture, if a cl_mem from subresource *subresource* of *resource* has already been created using **clCreateFromD3D10Texture2DKHR**, or if *context* was not created against the same Direct3D 10 device from which *resource* was created.

- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if the Direct3D 10 texture format of *resource* is not listed in *table 9.10.3*.

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

The width and height of the returned OpenCL 2D image object are determined by the width and height of subresource *subresource* of *resource*. The channel type and order of the returned OpenCL 2D image object is determined by the format of *resource* by *table 9.10.3*.

This call will increment the internal Direct3D reference count on *resource*. The internal Direct3D reference count on *resource* will be decremented when the OpenCL reference count on the returned OpenCL memory object drops to zero.

The function

cl_mem **clCreateFromD3D10Texture3DKHR** (cl_context *context*,
cl_mem_flags *flags*,
ID3D10Texture3D **resource*,
UINT *subresource*,
cl_int **errcode_ret*)

creates an OpenCL 3D image object from a subresource of a Direct3D 10 3D texture.

context is a valid OpenCL context created from a Direct3D 10 device.

flags is a bit-field that is used to specify usage information. Refer to table 5.3 for a description of *flags*. Only CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE values specified in *table 5.3* can be used.

resource is a pointer to the Direct3D 10 3D texture to share.

subresource is the subresource of *resource* to share.

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

clCreateFromD3D10Texture3DKHR returns a valid non-zero OpenCL image object and *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.

- CL_INVALID_VALUE if values specified in *flags* are not valid or if *subresource* is not a valid subresource index for *resource*.

- CL_INVALID_D3D10_RESOURCE_KHR if *resource* is not a Direct3D 10 texture resource, if *resource* was created with the D3D10_USAGE flag D3D10_USAGE_IMMUTABLE, if *resource* is a multisampled texture, if a cl_mem from subresource *subresource* of *resource* has already been created using **clCreateFromD3D10Texture3DKHR**, or if *context* was not created against the same Direct3D 10 device from which *resource* was created.

- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if the Direct3D 10 texture format of *resource* is not listed in *table 9.10.3*.

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

The width, height and depth of the returned OpenCL 3D image object are determined by the width, height and depth of subresource *subresource* of *resource*. The channel type and order of the returned OpenCL 3D image object is determined by the format of *resource* by *table 9.10.3*.

This call will increment the internal Direct3D reference count on *resource*. The internal Direct3D reference count on *resource* will be decremented when the OpenCL reference count on the returned OpenCL memory object drops to zero.

DXGI format	CL image format (channel order, channel data type)
DXGI_FORMAT_R32G32B32A32_FLOAT	CL_RGBA, CL_FLOAT
DXGI_FORMAT_R32G32B32A32_UINT	CL_RGBA, CL_UNSIGNED_INT32
DXGI_FORMAT_R32G32B32A32_SINT	CL_RGBA, CL_SIGNED_INT32
DXGI_FORMAT_R16G16B16A16_FLOAT	CL_RGBA, CL_HALF_FLOAT
DXGI_FORMAT_R16G16B16A16_UNORM	CL_RGBA, CL_UNORM_INT16
DXGI_FORMAT_R16G16B16A16_UINT	CL_RGBA, CL_UNSIGNED_INT16
DXGI_FORMAT_R16G16B16A16_SNORM	CL_RGBA, CL_SNORM_INT16
DXGI_FORMAT_R16G16B16A16_SINT	CL_RGBA, CL_SIGNED_INT16
DXGI_FORMAT_R8G8B8A8_UNORM	CL_RGBA, CL_UNORM_INT8
DXGI_FORMAT_R8G8B8A8_UINT	CL_RGBA, CL_UNSIGNED_INT8
DXGI_FORMAT_R8G8B8A8_SNORM	CL_RGBA, CL_SNORM_INT8
DXGI_FORMAT_R8G8B8A8_SINT	CL_RGBA, CL_SIGNED_INT8
DXGI_FORMAT_R32G32_FLOAT	CL_RG, CL_FLOAT
DXGI_FORMAT_R32G32_UINT	CL_RG, CL_UNSIGNED_INT32
DXGI_FORMAT_R32G32_SINT	CL_RG, CL_SIGNED_INT32
DXGI_FORMAT_R16G16_FLOAT	CL_RG, CL_HALF_FLOAT
DXGI_FORMAT_R16G16_UNORM	CL_RG, CL_UNORM_INT16
DXGI_FORMAT_R16G16_UINT	CL_RG, CL_UNSIGNED_INT16
DXGI_FORMAT_R16G16_SNORM	CL_RG, CL_SNORM_INT16

DXGI_FORMAT_R16G16_SINT	CL_RG, CL_SIGNED_INT16
DXGI_FORMAT_R8G8_UNORM	CL_RG, CL_UNORM_INT8
DXGI_FORMAT_R8G8_UINT	CL_RG, CL_UNSIGNED_INT8
DXGI_FORMAT_R8G8_SNORM	CL_RG, CL_SNORM_INT8
DXGI_FORMAT_R8G8_SINT	CL_RG, CL_SIGNED_INT8
DXGI_FORMAT_R32_FLOAT	CL_R, CL_FLOAT
DXGI_FORMAT_R32_UINT	CL_R, CL_UNSIGNED_INT32
DXGI_FORMAT_R32_SINT	CL_R, CL_SIGNED_INT32
DXGI_FORMAT_R16_FLOAT	CL_R, CL_HALF_FLOAT
DXGI_FORMAT_R16_UNORM	CL_R, CL_UNORM_INT16
DXGI_FORMAT_R16_UINT	CL_R, CL_UNSIGNED_INT16
DXGI_FORMAT_R16_SNORM	CL_R, CL_SNORM_INT16
DXGI_FORMAT_R16_SINT	CL_R, CL_SIGNED_INT16
DXGI_FORMAT_R8_UNORM	CL_R, CL_UNORM_INT8
DXGI_FORMAT_R8_UINT	CL_R, CL_UNSIGNED_INT8
DXGI_FORMAT_R8_SNORM	CL_R, CL_SNORM_INT8
DXGI_FORMAT_R8_SINT	CL_R, CL_SIGNED_INT8

Table 9.10.3 *List of Direct3D 10 and corresponding OpenCL image formats*

9.10.7.5 Querying Direct3D properties of memory objects created from Direct3D 10 resources

Properties of Direct3D 10 objects may be queried using **clGetMemObjectInfo** and **clGetImageInfo** with *param_name* CL_MEM_D3D10_RESOURCE_KHR and CL_IMAGE_D3D10_SUBRESOURCE_KHR as described in *sections 5.4.3* and *5.3.6*.

9.10.7.6 Sharing memory objects created from Direct3D 10 resources between Direct3D 10 and OpenCL contexts

The function

cl_int **clEnqueueAcquireD3D10ObjectsKHR** (cl_command_queue *command_queue*,
cl_uint *num_objects*,
const cl_mem **mem_objects*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

is used to acquire OpenCL memory objects that have been created from Direct3D 10 resources. The Direct3D 10 objects are acquired by the OpenCL context associated with *command_queue* and can therefore be used by all command-queues associated with the OpenCL context.

OpenCL memory objects created from Direct3D 10 resources must be acquired before they can be used by any OpenCL commands queued to a command-queue. If an OpenCL memory object created from a Direct3D 10 resource is used while it is not currently acquired by OpenCL, the call attempting to use that OpenCL memory object will return CL_D3D10_RESOURCE_NOT_ACQUIRED_KHR.

clEnqueueAcquireD3D10ObjectsKHR provides the synchronization guarantee that any Direct3D 10 calls made before **clEnqueueAcquireD3D10ObjectsKHR** is called will complete executing before *event* reports completion and before the execution of any subsequent OpenCL work issued in *command_queue* begins.

command_queue is a valid command-queue.

num_objects is the number of memory objects to be acquired in *mem_objects*.

mem_objects is a pointer to a list of OpenCL memory objects that were created from Direct3D 10 resources.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.

event returns an event object that identifies this particular 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.

clEnqueueAcquireD3D10ObjectsKHR returns CL_SUCCESS if the function is executed successfully. If <*num_objects*> is 0 and <*mem_objects*> is NULL then the function does nothing and returns CL_SUCCESS. Otherwise it returns one of the following errors:

- CL_INVALID_VALUE if *num_objects* is zero and *mem_objects* is not a NULL value or if *num_objects* > 0 and *mem_objects* is NULL.
- CL_INVALID_MEM_OBJECT if memory objects in *mem_objects* are not valid OpenCL memory objects or if memory objects in *mem_objects* have not been created from Direct3D 10 resources.
- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* was not created

from an Direct3D 10 context.

- CL_D3D10_RESOURCE_ALREADY_ACQUIRED_KHR if memory objects in *mem_objects* have previously been acquired using **clEnqueueAcquireD3D10ObjectsKHR** but have not been released using **clEnqueueReleaseD3D10ObjectsKHR**.

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* 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 function

cl_int **clEnqueueReleaseD3D10ObjectsKHR** (cl_command_queue *command_queue*,
cl_uint *num_objects*,
const cl_mem **mem_objects*,
cl_uint *num_events_in_wait_list*,
const cl_event **event_wait_list*,
cl_event **event*)

is used to release OpenCL memory objects that have been created from Direct3D 10 resources. The Direct3D 10 objects are released by the OpenCL context associated with *command_queue*.

OpenCL memory objects created from Direct3D 10 resources which have been acquired by OpenCL must be released by OpenCL before they may be accessed by Direct3D 10. Accessing a Direct3D 10 resource while its corresponding OpenCL memory object is acquired is in error and will result in undefined behavior, including but not limited to possible OpenCL errors, data corruption, and program termination.

clEnqueueReleaseD3D10ObjectsKHR provides the synchronization guarantee that any calls to Direct3D 10 made after the call to **clEnqueueReleaseD3D10ObjectsKHR** will not start executing until after all events in *event_wait_list* are complete and all work already submitted to *command_queue* completes execution.

num_objects is the number of memory objects to be released in *mem_objects*.

mem_objects is a pointer to a list of OpenCL memory objects that were created from Direct3D 10 resources.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event* returns an event object that identifies this particular 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.

clEnqueueReleaseExternalObjectsKHR returns CL_SUCCESS if the function is executed successfully. If *num_objects* is 0 and <*mem_objects*> is NULL the function does nothing and returns CL_SUCCESS. Otherwise it returns one of the following errors:

- CL_INVALID_VALUE if *num_objects* is zero and *mem_objects* is not a NULL value or if *num_objects* > 0 and *mem_objects* is NULL.

- CL_INVALID_MEM_OBJECT if memory objects in *mem_objects* are not valid OpenCL memory objects or if memory objects in *mem_objects* have not been created from Direct3D 10 resources.

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

- CL_INVALID_CONTEXT if context associated with *command_queue* was not created from a Direct3D 10 device.

- CL_D3D10_RESOURCE_NOT_ACQUIRED_KHR if memory objects in *mem_objects* have not previously **clEnqueueAcquireD3D10ObjectsKHR**, or have been released using **clEnqueueReleaseD3D10ObjectsKHR** since the last time that they were acquired.

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list*> is 0, or if event objects in *event_wait_list* 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.

9.10.8 Issues

1) Should this extension be KHR or EXT?

PROPOSED: KHR. If this extension is to be approved by Khronos then it should be KHR, otherwise EXT. Not all platforms can support this extension, but that is also true of OpenGL interop.

RESOLVED: KHR.

2) Requiring SharedHandle on ID3D10Resource

Requiring this can largely simplify things at the DDI level and make some implementations faster. However, the DirectX spec only defines the shared handle for a subset of the resources we would like to support:

> D3D10_RESOURCE_MISC_SHARED - Enables the sharing of resource data between two or more Direct3D devices. The only resources that can be shared are 2D non-mipmapped textures.

PROPOSED A: Add wording to the spec about some implementations needing the resource setup as shared:

"Some implementations may require the resource to be shared on the D3D10 side of the API"

If we do that, do we need another enum to describe this failure case?

PROPOSED B: Require that all implementations support both shared and non-shared resources. The restrictions prohibiting multisample textures and the flag D3D10_USAGE_IMMUTABLE guarantee software access to all shareable resources.

RESOLVED: Require that implementations support both D3D10_RESOURCE_MISC_SHARED being set and not set. Add the query for CL_CONTEXT_D3D10_PREFER_SHARED_RESOURCES_KHR to determine on a per-context basis which method will be faster.

3) Texture1D support

There is not a matching CL type, so do we want to support this and map to buffer or Texture2D? If so the command might correspond to the 2D / 3D versions:

cl_mem **clCreateFromD3D10Texture1D** (cl_context *context*,
cl_mem_flags *flags*,
ID3D10Texture2D **resource*,
UINT *subresource*,
cl_int **errcode_ret*)

RESOLVED: We will not add support for ID3D10Texture1D objects unless a corresponding OpenCL 1D Image type is created.

4) CL/D3D10 queries

The GL interop has clGetGLObjectInfo and clGetGLTextureInfo. It is unclear if these are needed on the D3D10 interop side since the D3D10 spec makes these queries trivial on the D3D10 object itself. Also, not all of the sematics of the GL call map across.

PROPOSED: Add the **clGetMemObjectInfo** and **clGetImageInfo** parameter names

CL_MEM_D3D10_RESOURCE_KHR and CL_IMAGE_D3D10_SUBRESOURCE_KHR to query the D3D10 resource from which a cl_mem was created. From this data, any D3D10 side information may be queried using the D3D10 API.

RESOLVED: We will use **clGetMemObjectInfo** and **clGetImageInfo** to access this information.

10. OpenCL Embedded Profile

The OpenCL 1.1 specification describes the feature requirements for desktop platforms. This section describes the OpenCL 1.1 embedded profile that allows us to target a subset of the OpenCL 1.1 specification for handheld and embedded platforms. The optional extensions defined in *section 9* apply to both profiles.

The OpenCL 1.1 embedded profile has the following restrictions:

1. 64 bit integers i.e. long, ulong including the appropriate vector data types and operations on 64-bit integers are optional. The **cles_khr_int64**[68] extension string will be reported if the embedded profile implementation supports 64-bit integers.

2. Support for 3D images is optional.

   If CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_MAX_HEIGHT and CL_DEVICE_IMAGE3D_MAX_DEPTH are zero, the call to **clCreateImage3D** in the embedded profile will fail to create the 3D image. The *errcode_ret* argument in **clCreateImage3D** returns CL_INVALID_OPERATION. Declaring arguments of type `image3d_t` in a kernel will result in a compilation error.

   If CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_HEIGHT and CL_DEVICE_IMAGE3D_MAX_DEPTH > 0, 3D images are supported by the OpenCL embedded profile implementation. **clCreateImage3D** will work as defined by the OpenCL specification. The `image3d_t` data type can be used in a kernel(s).

3. 2D and 3D images created with an `image_channel_data_type` value of CL_FLOAT or CL_HALF_FLOAT can only be used with samplers that use a filter mode of CL_FILTER_NEAREST. The values returned by **read_imagef** and **read_imageh**[69] for 2D and 3D images if `image_channel_data_type` value is CL_FLOAT or CL_HALF_FLOAT and sampler with `filter_mode` = CL_FILTER_LINEAR are undefined.

4. The sampler addressing modes supported for 2D and 3D images are: CLK_ADDRESS_NONE, CLK_ADDRESS_MIRRORED_REPEAT, CLK_ADDRESS_REPEAT, CLK_ADDRESS_CLAMP_TO_EDGE and CLK_ADDRESS_CLAMP.

5. The mandated minimum single precision floating-point capability given by CL_DEVICE_SINGLE_FP_CONFIG is CL_FP_ROUND_TO_ZERO or CL_FP_ROUND_TO_NEAREST. If CL_FP_ROUND_TO_NEAREST is supported, the

---

[68] Note that the performance of 64-bit integer arithmetic can vary significantly between embedded devices.

[69] If **`cl_khr_fp16`** extension is supported.

default rounding mode will be round to nearest even; otherwise the default rounding mode will be round to zero.

6. The single precision floating-point operations (addition, subtraction and multiplication) shall be correctly rounded. Zero results may always be positive 0.0. The accuracy of division and sqrt are given by *table 10.1*.

   If CL_FP_INF_NAN is not set in CL_DEVICE_SINGLE_FP_CONFIG, and one of the operands or the correctly rounded result of addition, subtraction, multiplication or division is INF or NaN, the value of the result is implementation-defined. Likewise, single precision comparison operators (<, >, <=, >=, ==, !=) return implementation-defined values when one or more operands is a NaN.

   In all cases, conversions (*section 6.2* and *6.11.7*) shall be correctly rounded as described for the FULL_PROFILE, including those that consume or produce an INF or NaN. The built-in math functions (*section 6.11.2*) shall behave as described for the FULL_PROFILE, including edge case behavior described in *section 7.5.1* but with accuracy as described by *table 10.1*.

   Note: If addition, subtraction and multiplication have default round to zero rounding mode, then **fract**, **fma** and **fdim** shall produce the correctly rounded result for round to zero rounding mode.

   This relaxation of the requirement to adhere to IEEE 754 requirements for basic floating-point operations, though extremely undesirable, is to provide flexibility for embedded devices that have lot stricter requirements on hardware area budgets.

7. Denormalized numbers for the half data type which may be generated when converting a float to a half using variants of the **vstore_half** function or when converting from a half to a float using variants of the **vload_half** function can be flushed to zero. Refer to *section 6.1.1.1*.

8. The precision of conversions from CL_UNORM_INT8, CL_SNORM_INT8, CL_UNORM_INT16 and CL_SNORM_INT16 to float is <= 2 ulp for the embedded profile instead of <= 1.5 ulp as defined in *section 8.3.1.1*. The exception cases described in *section 8.3.1.1* and given below apply to the embedded profile.

   For CL_UNORM_INT8

   ```
   0 must convert to 0.0f and
   255 must convert to 1.0f
   ```

   For CL_UNORM_INT16

   ```
   0 must convert to 0.0f and
   65535 must convert to 1.0f
   ```

For CL_SNORM_INT8

```
-128 and -127 must convert to -1.0f,
0 must convert to 0.0f and
127 must convert to 1.0f
```

For CL_SNORM_INT16

```
-32768 and -32767 must convert to -1.0f,
0 must convert to 0.0f and
32767 must convert to 1.0f
```

For CL_UNORM_INT_101010

```
0 must convert to 0.0f and
1023 must convert to 1.0f
```

9. Built-in atomic functions as defined in *section 6.11.11* are optional.

The following optional extensions defined in the OpenCL 1.1 specification (*section 9*) are available to the embedded profile:

- **cl_khr_fp64. If this extension is supported, cles_khr_int64 must also be supported.**
- **cl_khr_int64_base_atomics**
- **cl_khr_int64_extended_atomics**
- **cl_khr_fp16**
- **cles_khr_int64**

The following optional extensions defined in the OpenCL 1.0 specification (*section 9*) are also available to the embedded profile:

- **cl_khr_global_int32_base_atomics**
- **cl_khr_global_int32_extended_atomics**
- **cl_khr_local_int32_base_atomics**
- **cl_khr_local_int32_extended_atomics**

Table 10.1 describes the minimum accuracy of single precision floating-point arithmetic operations given as ULP values for the embedded profile. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

Function	**Min Accuracy - ULP values**[70]
$x + y$	Correctly rounded

---

[70] 0 ulp is used for math functions that do not require rounding.

x – y	Correctly rounded
x * y	Correctly rounded
1.0 / *x*	<= 3 ulp
x / y	<= 3 ulp
acos	<= 4 ulp
acospi	<= 5 ulp
asin	<= 4 ulp
asinpi	<= 5 ulp
atan	<= 5 ulp
atan2	<= 6 ulp
atanpi	<= 5 ulp
atan2pi	<= 6 ulp
acosh	<= 4 ulp
asinh	<= 4 ulp
atanh	<= 5 ulp
cbrt	<= 4 ulp
ceil	Correctly rounded
copysign	0 ulp
cos	<= 4 ulp
cosh	<= 4 ulp
cospi	<= 4 ulp
erfc	<= 16 ulp
erf	<= 16 ulp
exp	<= 4 ulp
exp2	<= 4 ulp
exp10	<= 4 ulp
expm1	<= 4 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp
hypot	<= 4 ulp
ilogb	0 ulp
ldexp	Correctly rounded
log	<= 4 ulp
log2	<= 4 ulp
log10	<= 4 ulp
log1p	<= 4 ulp
logb	0 ulp

mad	Any value allowed (infinite ulp)
maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	<= 16 ulp
pown(x, y)	<= 16 ulp
powr(x, y)	<= 16 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	<= 16 ulp
round	Correctly rounded
rsqrt	<= 4 ulp
sin	<= 4 ulp
sincos	<= 4 ulp for sine and cosine values
sinh	<= 4 ulp
sinpi	<= 4 ulp
sqrt	<= 4 ulp
tan	<= 5 ulp
tanh	<= 5 ulp
tanpi	<= 6 ulp
tgamma	<= 16 ulp
trunc	Correctly rounded
half_cos	<= 8192 ulp
half_divide	<= 8192 ulp
half_exp	<= 8192 ulp
half_exp2	<= 8192 ulp
half_exp10	<= 8192 ulp
half_log	<= 8192 ulp
half_log2	<= 8192 ulp
half_log10	<= 8192 ulp
half_powr	<= 8192 ulp
half_recip	<= 8192 ulp
half_rsqrt	<= 8192 ulp
half_sin	<= 8192 ulp
half_sqrt	<= 8192 ulp
half_tan	<= 8192 ulp
native_cos	Implementation-defined
native_divide	Implementation-defined
native_exp	Implementation-defined
native_exp2	Implementation-defined

native_exp10	Implementation-defined
native_log	Implementation-defined
native_log2	Implementation-defined
native_log10	Implementation-defined
native_powr	Implementation-defined
native_recip	Implementation-defined
native_rsqrt	Implementation-defined
native_sin	Implementation-defined
native_sqrt	Implementation-defined
native_tan	Implementation-defined

Table 10.1 *ULP values for built-in math functions*

The `__EMBEDDED_PROFILE__` macro is added to the language (refer to *section 6.9*). It will be the integer constant 1 for OpenCL devices that implement the embedded profile and is undefined otherwise.

CL_PLATFORM_PROFILE defined in *table 4.1* will return the string EMBEDDED_PROFILE if the OpenCL implementation supports the embedded profile only.

The minimum maximum values specified in *table 4.3* have been modified for the OpenCL embedded profile and are:

cl_device_info	**Return Type**	**Description**
CL_DEVICE_TYPE	cl_device_type	The OpenCL device type. Currently supported values are: CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_ACCELERATOR, CL_DEVICE_TYPE_DEFAULT or a combination of the above.
CL_DEVICE_VENDOR_ID	cl_uint	A unique device vendor identifier. An example of a unique device identifier could be the PCIe ID.
CL_DEVICE_MAX_COMPUTE_UNITS	cl_uint	The number of parallel compute cores on the OpenCL device. The minimum value is 1.
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	unsigned int	Maximum dimensions that specify the global and local work-item IDs. The minimum value is 3.
CL_DEVICE_MAX_WORK_ITEM_SIZES	size_t []	Maximum number of work-items that can be specified in each dimension of the work-group to **clEnqueueNDRangeKernel**.

		Returns *n* size_t entries, where *n* is the value returned by the query for CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS. The minimum value is (1, 1, 1).
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t	Maximum number of work-items in a work-group executing a kernel using the data parallel execution model. (Refer to **clEnqueueNDRangeKernel**). The minimum value is 1.
CL_DEVICE_PREFERRED_ VECTOR_WIDTH_CHAR **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_SHORT** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_INT** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_LONG** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_FLOAT** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_DOUBLE** **CL_DEVICE_PREFERRED_ VECTOR_WIDTH_HALF**	cl_uint	Preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. If the **cl_khr_fp64** extension is not supported, CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE must return 0. If the **cl_khr_fp16** extension is not supported, CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF must return 0.
CL_DEVICE_NATIVE_ VECTOR_WIDTH_CHAR **CL_DEVICE_NATIVE_ VECTOR_WIDTH_SHORT** **CL_DEVICE_NATIVE_ VECTOR_WIDTH_INT** **CL_DEVICE_NATIVE_ VECTOR_WIDTH_LONG** **CL_DEVICE_NATIVE_ VECTOR_WIDTH_FLOAT** **CL_DEVICE_NATIVE_ VECTOR_WIDTH_DOUBLE** **CL_DEVICE_NATIVE_ VECTOR_WIDTH_HALF**	cl_uint	Returns the native ISA vector width. The vector width is defined as the number of scalar elements that can be stored in the vector. If the **cl_khr_fp64** extension is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE must return 0. If the **cl_khr_fp16** extension is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF must return 0.
CL_DEVICE_MAX_CLOCK_FREQUENCY	cl_uint	Maximum configured clock frequency of the device in MHz.
CL_DEVICE_ADDRESS_BITS	cl_uint	The default compute device address space size specified as an unsigned

		integer value in bits. Currently supported values are 32 or 64 bits. If the value reported by the embedded profile is 64, then the **cles_khr_int64** extension must be supported.
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t	Maximum number of work-items in a work-group executing a kernel using the data parallel execution model. (Refer to **clEnqueueNDRangeKernel**). The minimum value is 1.
CL_DEVICE_MAX_MEM_ALLOC_SIZE	unsigned long long	Max size of memory object allocation in bytes. The minimum value is max (1/4th of **CL_DEVICE_GLOBAL_MEM_SIZE** , 1*1024*1024)
CL_DEVICE_IMAGE_SUPPORT	cl_bool	Is CL_TRUE if images are supported by the OpenCL device and CL_FALSE otherwise.
CL_DEVICE_MAX_READ_IMAGE_ARGS	unsigned int	Max number of simultaneous image objects that can be read by a kernel. The minimum value is 8 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	unsigned int	Max number of simultaneous image objects that can be written to by a kernel. The minimum value is 1 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE2D_MAX_WIDTH	size_t	Max width of 2D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE2D_MAX_HEIGHT	size_t	Max height of 2D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE3D_MAX_WIDTH	size_t	Max width of 3D image in pixels. The minimum value is 0 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE3D_MAX_HEIGHT	size_t	Max height of 3D image in pixels. The minimum value is 0.
CL_DEVICE_IMAGE3D_MAX_DEPTH	size_t	Max depth of 3D image in pixels. The minimum value is 0.

CL_DEVICE_MAX_SAMPLERS	unsigned int	Maximum number of samplers that can be used in a kernel. Refer to *section 6.11.13* for a detailed description on samplers. The minimum value is 8 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_PARAMETER_SIZE	size_t	Max size in bytes of the arguments that can be passed to a kernel. The minimum value is 256 bytes.
CL_DEVICE_MEM_BASE_ADDR_ALIGN	cl_uint	Describes the alignment in bits of the base address of any allocated memory object.
CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE	cl_uint	The smallest alignment in bytes which can be used for any data type.
CL_DEVICE_SINGLE_FP_CONFIG	cl_device_ fp_config	Describes single precision floating-point capability of the device. This is a bit-field that describes one or more of the following values: CL_FP_DENORM – denorms are supported CL_FP_INF_NAN – INF and quiet NaNs are supported. CL_FP_ROUND_TO_NEAREST– round to nearest even rounding mode supported CL_FP_ROUND_TO_ZERO – round to zero rounding mode supported CL_FP_ROUND_TO_INF – round to +ve and –ve infinity rounding modes supported CL_FP_FMA – IEEE754-2008 fused multiply-add is supported. CL_FP_SOFT_FLOAT – Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. The mandated minimum floating-point capability is: CL_FP_ROUND_TO_ZERO or CL_FP_ROUND_TO_NEAREST.

CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	cl_device_mem_ cache_type	Type of global memory cache supported. Valid values are: CL_NONE, CL_READ_ONLY_CACHE and CL_READ_WRITE_CACHE.
CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE	cl_uint	Size of global memory cache line in bytes.
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE	cl_ulong	Size of global memory cache in bytes.
CL_DEVICE_GLOBAL_MEM_SIZE	cl_ulong	Size of global device memory in bytes.
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	unsigned long long	Max size in bytes of a constant buffer allocation. The minimum value is 1 KB.
CL_DEVICE_MAX_CONSTANT_ARGS	unsigned int	Max number of arguments declared with the `__constant` qualifier in a kernel. The minimum value is 4.
CL_DEVICE_LOCAL_MEM_TYPE	cl_device_ local_mem_type	Type of local memory supported. This can be set to CL_LOCAL implying dedicated local memory storage such as SRAM, or CL_GLOBAL.
CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong	Size of local memory arena in bytes. The minimum value is 1 KB.
CL_DEVICE_ERROR_CORRECTION_SUPPORT	cl_bool	Is CL_TRUE if the device implements error correction for all accesses to compute device memory (global and constant). Is CL_FALSE if the device does not implement such error correction.
CL_DEVICE_HOST_UNIFIED_MEMORY	cl_bool	Is CL_TRUE if the device and the host have a unified memory subsystem and is CL_FALSE otherwise.
CL_DEVICE_PROFILING_TIMER_RESOLUTION	size_t	Describes the resolution of device timer. This is measured in nanoseconds. Refer to *section 5.12* for details.
CL_DEVICE_ENDIAN_LITTLE	cl_bool	Is CL_TRUE if the OpenCL device is a little endian device and CL_FALSE otherwise.
CL_DEVICE_AVAILABLE	cl_bool	Is CL_TRUE if the device is available

		and CL_FALSE if the device is not available.
CL_DEVICE_COMPILER_AVAILABLE	cl_bool	Is CL_FALSE if the implementation does not have a compiler available to compile the program source. Is CL_TRUE if the compiler is available. This can be CL_FALSE for the embedded platform profile only.
CL_DEVICE_EXECUTION_CAPABILITIES	cl_device_exec_ capabilities	Describes the execution capabilities of the device. This is a bit-field that describes one or more of the following values: CL_EXEC_KERNEL – The OpenCL device can execute OpenCL kernels. CL_EXEC_NATIVE_KERNEL – The OpenCL device can execute native kernels. The mandated minimum capability is: CL_EXEC_KERNEL.
CL_DEVICE_QUEUE_PROPERTIES	cl_command_ queue_properties	Describes the command-queue properties supported of the device. This is a bit-field that describes one or more of the following values: CL_QUEUE_OUT_OF_ORDER_EXEC_ MODE_ENABLE CL_QUEUE_PROFILING_ENABLE These properties are described in *table 5.1*. The mandated minimum capability is: CL_QUEUE_PROFILING_ENABLE.

If CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_TRUE, the values assigned to CL_DEVICE_MAX_READ_IMAGE_ARGS, CL_DEVICE_MAX_WRITE_IMAGE_ARGS, CL_DEVICE_IMAGE2D_MAX_WIDTH, CL_DEVICE_IMAGE2D_MAX_HEIGHT, CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_MAX_HEIGHT, CL_DEVICE_IMAGE3D_MAX_DEPTH and CL_DEVICE_MAX_SAMPLERS by the implementation must be greater than or equal to the minimum values specified in the embedded profile version of *table 4.3* given above. In addition, the following list of image formats must be supported by the OpenCL embedded profile implementation.

For 2D and optional 3D images, the minimum list of supported image formats (for reading and writing) is:

image_num_channels	**image_channel_order**	**image_channel_data_type**
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT

11. References

1. The ISO/IEC 9899:1999 “C” Language Specification.

2. The ISO/IEC JTC1 SC22 WG14 N1169 Specification.

3. The ANSI/IEEE Std 754-1985 and 754-2008 Specifications.

4. The AltiVec™ Technology Programming Interface Manual.

5. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugarman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan. *Brook for GPUs: Stream Computing on Graphics Hardware*

6. Ian Buck. *Brook Specification v0.2.*
http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf

7. *NVIDIA CUDA Programming Guide*.
http://developer.nvidia.com/object/cuda.html

8. *ATI CTM Guide – Technical Reference Manual*
http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

9. *OpenMP Application Program Interface*.
http://www.openmp.org/drupal/mp-documents/spec25.pdf

10. *The OpenGL Specification and the OpenGL Shading Language Specification*
http://www.opengl.org/registry/

11. *NESL – A nested data parallel language.*
http://www.cs.cmu.edu/~scandal/nesl.html

12. ***On the definition of ulp*** (x) by Jean-Michel Muller
ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5504.pdf

13. Explicit Memory Fences
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2262.html

14. Lefohn, Kniss, Strzodka, Sengupta, Owens, "Glift: Generic, Efficient, Random-Access GPU Data Structures," ACM Transactions on Graphics, Jan. 2006. pp 60--99.

15. Pharr, Lefohn, Kolb, Lalonde, Foley, Berry, "Programmable Graphics---The Future of Interactive Rendering," Neoptica Whitepaper, Mar. 2007.

16. Jens Maurer, Michael Wong. Towards support for attributes in C++ (Revision 4).
March 2008. Proposed to WG21 “Programming Language C++, Core Working Group”.

17. GCC Attribute Syntax. http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html.

Appendix A

A.1 Shared OpenCL Objects

This section describes which objects can be shared across multiple command-queues created within a host process.

OpenCL memory objects, program objects and kernel objects are created using a context and can be shared across multiple command-queues created using the same context. Event objects can be created when a command is queued to a command-queue. These event objects can be shared across multiple command-queues created using the same context.

The application needs to implement appropriate synchronization across threads on the host processor to ensure that the changes to the state of a shared object (such as a command-queue object, memory object, program or kernel object) happen in the correct order (deemed correct by the application) when multiple command-queues in multiple threads are making changes to the state of a shared object.

A command-queue can cache changes to the state of a memory object on the device associated with the command-queue. To synchronize changes to a memory object across command-queues, the application must do the following:

In the command-queue that includes commands that modify the state of a memory object, the application must do the following:

- Get appropriate event objects for commands that modify the state of the shared memory object.
- Call the **clFlush** (or **clFinish**) API to issue any outstanding commands from this command-queue.

In the command-queue that wants to synchronize to the latest state of a memory object, commands queued by the application must use the appropriate event objects that represent commands that modify the state of the shared memory object as event objects to wait on. This is to ensure that commands that use this shared memory object complete in the previous command-queue before the memory objects are used by commands executing in this command-queue.

The results of modifying a shared resource in one command-queue while it is being used by another command-queue are undefined.

A.2 Multiple Host Threads

All OpenCL API calls are thread-safe[71] except **clSetKernelArg**. **clSetKernelArg** is safe to call from any host thread, and is safe to call re-entrantly so long as concurrent calls operate on different cl_kernel objects. However, the behavior of the cl_kernel object is undefined if **clSetKernelArg** is called from multiple host threads on the same cl_kernel object at the same time[72]. Please note that there are additional limitations as to which OpenCL APIs may be called from OpenCL callback functions -- please see *section 5.9*.

The behavior of OpenCL APIs called from an interrupt or signal handler is implementation-defined

The OpenCL implementation should be able to create multiple command-queues for a given OpenCL context and multiple OpenCL contexts in an application running on the host processor.

---

[71] Please refer to the OpenCL glossary for the OpenCL definition of thread-safe. This definition may be different from usage of the term in other contexts.

[72] There is an inherent race condition in the design of OpenCL that occurs between setting a kernel argument and using the kernel with clEnqueueNDRangeKernel or clEnqueueTask. Another host thread might change the kernel arguments between when a host thread sets the kernel arguments and then enqueues the kernel, causing the wrong kernel arguments to be enqueued. Rather than attempt to share cl_kernel objects among multiple host threads, applications are strongly encouraged to make additional cl_kernel objects for kernel functions for each host thread.

Appendix B — Portability

OpenCL is designed to be portable to other architectures and hardware designs. OpenCL uses at its core a C99 based programming language. Floating-point arithmetic is based on the **IEEE-754** and **IEEE-754-2008** standards. The memory objects, pointer qualifiers and weakly ordered memory are designed to provide maximum compatibility with discrete memory architectures implemented by OpenCL devices. Command-queues and barriers allow for synchronization between the host and OpenCL devices. The design, capabilities and limitations of OpenCL are very much a reflection of the capabilities of underlying hardware.

Unfortunately, there are a number of areas where idiosyncrasies of one hardware platform may allow it to do some things that do not work on another. By virtue of the rich operating system resident on the CPU, on some implementations the kernels executing on a CPU may be able to call out to system services like `printf` whereas the same call on the GPU will likely fail for now. (Please see *section 6.8*). Since there is some advantage to having these services available for debugging purposes, implementations can use the OpenCL extension mechanism to implement these services.

Likewise, the heterogeneity of computing architectures might mean that a particular loop construct might execute at an acceptable speed on the CPU but very poorly on a GPU, for example. CPUs are designed in general to work well on latency sensitive algorithms on single threaded tasks, whereas common GPUs may encounter extremely long latencies, potentially orders of magnitude worse. A developer interested in writing portable code may find that it is necessary to test his design on a diversity of hardware designs to make sure that key algorithms are structured in a way that works well on a diversity of hardware. We suggest favoring more work-items over fewer. It is anticipated that over the coming months and years experience will produce a set of best practices that will help foster a uniformly favorable experience on a diversity of computing devices.

Of somewhat more concern is the topic of endianness. Since a majority of devices supported by the initial implementation of OpenCL are little-endian, developers need to make sure that their kernels are tested on both big-endian and little-endian devices to ensure source compatibility with OpenCL devices now and in the future. The endian attribute qualifier is supported by the OpenCL C programming language to allow developers to specify whether the data uses the endianness of the host or the OpenCL device. This allows the OpenCL compiler to do appropriate endian-conversion on load and store operations from or to this data.

We also describe how endianness can leak into an implementation causing kernels to produce unintended results:

When a big-endian vector machine (e.g. AltiVec, CELL SPE) loads a vector, the order of the data is retained. That is both the order of the bytes within each element and the order of the elements in the vector are the same as in memory. When a little-endian vector machine (e.g. SSE) loads a vector, the order of the data in register (where all the work is done) is reversed.

Both the order of the bytes within each element and the order of the elements with respect to one another in the vector are reversed.

Memory:

`uint4 a =`	**0x00010203**	0x04050607	0x08090A0B	0x0C0D0E0F

In register (big-endian):

`uint4 a =`	**0x00010203**	0x04050607	0x08090A0B	0x0C0D0E0F

In register (little-endian):

`uint4 a =`	0x0F0E0D0C	0x0B0A0908	0x07060504	**0x03020100**

This allows little-endian machines to use a single vector load to load little-endian data, regardless of how large each piece of data is in the vector. That is the transformation is equally valid whether that vector was a `uchar16` or a `ulong2`. Of course, as is well known, little-endian machines actually[73] store their data in reverse byte order to compensate for the little-endian storage format of the array elements:

Memory (big-endian):

`uint4 a =`	**0x00010203**	0x04050607	0x08090A0B	0x0C0D0E0F

Memory (little-endian):

`uint4 a =`	**0x03020100**	0x07060504	0x0B0A0908	0x0F0E0D0C

Once that data is loaded into a vector, we end up with this:

In register (big-endian):

`uint4 a =`	**0x00010203**	0x04050607	0x08090A0B	0x0C0D0E0F

---

[73] Note that we are talking about the programming model here. In reality, little endian systems might choose to simply address their bytes from "the right" or reverse the "order" of the bits in the byte. Either of these choices would mean that no big swap would need to occur in hardware.

In register (little-endian):

`uint4 a =`

0x0C0D0E0F	0x08090A0B	0x04050607	**0x00010203**

That is, in the process of correcting the endianness of the bytes within each element, the machine ends up reversing the order that the elements appear in the vector with respect to each other within the vector. `0x00010203` appears at the left of the big-endian vector and at the right of the little-endian vector.

OpenCL provides a consistent programming model across architectures by numbering elements according to their order in memory. Concepts such as `even/odd` and `high/low` follow accordingly. Once the data is loaded into registers, we find that element `0` is at the left of the big-endian vector and element `0` is at the right of the little-endian vector:

```
float x[4];
float4 v = vload4( 0, x );
```

Big-endian:

```
     v contains { x[0], x[1], x[2], x[3] }
```

Little-endian:

```
     v contains { x[3], x[2], x[1], x[0] }
```

The compiler is aware that this swap occurs and references elements accordingly. So long as we refer to them by a numeric index such as `.s0123456789abcdef` or by descriptors such as `.xyzw`, `.hi`, `.lo`, `.even` and `.odd`, everything works transparently. Any ordering reversal is undone when the data is stored back to memory. The developer should be able to work with a big endian programming model and ignore the element ordering problem in the vector ... for most problems. This mechanism relies on the fact that we can rely on a consistent element numbering. Once we change numbering system, for example by conversion-free casting (using `as_type`*n*) a vector to another vector of the same size but a different number of elements, then we get different results on different implementations depending on whether the system is big- endian, or little-endian or indeed has no vector unit at all. (Thus, the behavior of bitcasts to vectors of different numbers of elements is implementation-defined, see *section 6.2.4*)

An example follows:

```
float x[4] = { 0.0f, 1.0f, 2.0f, 3.0f };
float4 v = vload4( 0, x );
uint4 y = (uint4) v;          // legal, portable
ushort8 z = (ushort8) v;      // legal, not portable
                              // element size changed
```

Big-endian:

```
     v contains { 0.0f, 1.0f, 2.0f, 3.0f }
```

```
y contains { 0x00000000, 0x3f800000,
               0x40000000, 0x40400000 }
z contains { 0x0000, 0x0000, 0x3f80, 0x0000,
               0x4000, 0x0000, 0x4040, 0x0000 }
z.z is 0x3f80
```

Little-endian:

```
v contains { 3.0f, 2.0f, 1.0f, 0.0f }
y contains { 0x40400000, 0x40000000,
               0x3f800000, 0x00000000 }
z contains { 0x4040, 0x0000, 0x4000,
               0x0000, 0x3f80, 0x0000, 0x0000, 0x0000
          }
z.z is 0
```

Here, the value in `z.z` is not the same between big- and little-endian vector machines

OpenCL could have made it illegal to do a conversion free cast that changes the number of elements in the name of portability. However, while OpenCL provides a common set of operators drawing from the set that are typically found on vector machines, it can not provide access to everything every ISA may offer in a consistent uniform portable manner. Many vector ISAs provide special purpose instructions that greatly accelerate specific operations such as DCT, SAD, or 3D geometry. It is not intended for OpenCL to be so heavy handed that time-critical performance sensitive algorithms can not be written by knowledgeable developers to perform at near peak performance. Developers willing to throw away portability should be able to use the platform-specific instructions in their code. For this reason, OpenCL is designed to allow traditional vector C language programming extensions, such as the AltiVec C Programming Interface or the Intel C programming interfaces (such as those found in emmintrin.h) to be used directly in OpenCL with OpenCL data types as an extension to OpenCL. As these interfaces rely on the ability to do conversion-free casts that change the number of elements in the vector to function properly, OpenCL allows them too.

As a general rule, any operation that operates on vector types in segments that are not the same size as the vector element size may break on other hardware with different endianness or different vector architecture.

Examples might include:

- Combining two `uchar8's` containing high and low bytes of a `ushort`, to make a `ushort8` using `.even` and `.odd` operators (please use **upsample()** for this, see *section 6.11.3*)

- Any bitcast that changes the number of elements in the vector. (Operations on the new type are non-portable.)

- Swizzle operations that change the order of data using chunk sizes that are not the same as the element size

Examples of operations that are portable:

- Combining two `uint8's` to make a `uchar16` using `.even` and `.odd` operators. For example to interleave left and right audio streams.
- Any bitcast that does not change the number of elements (e.g. `(float4) unit4` -- we define the storage format for floating-point types)
- Swizzle operations that swizzle elements of the same size as the elements of the vector.

OpenCL has made some additions to C to make application behavior more dependable than C. Most notably in a few cases OpenCL defines the behavior of some operations that are undefined in C99:

- OpenCL provides `convert_` operators for conversion between all types. C99 does not define what happens when a floating-point type is converted to integer type and the floating-point value lies outside the representable range of the integer type after rounding. When the `_sat` variant of the conversion is used, the float shall be converted to the nearest representable integer value. Similarly, OpenCL also makes recommendations about what should happen with `NaN`. Hardware manufacturers that provide the saturated conversion in hardware may use the saturated conversion hardware for both the saturated and non-saturated versions of the OpenCL `convert_` operator. OpenCL does not define what happens for the non-saturated conversions when floating-point operands are outside the range representable integers after rounding.
- The format of `half`, `float`, and `double` types is defined to be the binary16, binary32 and binary64 formats in the draft IEEE-754 standard. (The latter two are identical to the existing IEEE-754 standard.) You may depend on the positioning and meaning of the bits in these types.
- OpenCL defines behavior for oversized shift values. Shift operations that shift greater than or equal to the number of bits in the first operand reduce the shift value modulo the number of bits in the element. For example, if we shift an `int4` left by `33` bits, OpenCL treats this as shift left by `33%32 = 1` bit.
- A number of edge cases for math library functions are more rigorously defined than in C99. Please see *section 7.5*.

Appendix C — Application Data Types

This section documents the provided host application types and constant definitions. The documented material describes the commonly defined data structures, types and constant values available to all platforms and architectures. The addition of these details demonstrates our commitment to maintaining a portable programming environment and potentially deters changes to the supplied headers.

C.1 Shared Application Scalar Data Types

The following application scalar types are provided for application convenience.

```
cl_char
cl_uchar
cl_short
cl_ushort
cl_int
cl_uint
cl_long
cl_ulong
cl_half
cl_float
cl_double
```

C.2 Supported Application Vector Data Types

Application vector types are unions used to create vectors of the above application scalar types. The following application vector types are provided for application convenience.

```
cl_charn
cl_ucharn
cl_shortn
cl_ushortn
cl_intn
cl_uintn
cl_longn
cl_ulongn
cl_halfn
cl_floatn
cl_doublen
```

n can be 2, 3, 4, 8 or 16.

The application scalar and vector data types are defined in the **cl_platform.h** header file.

C.3 Alignment of Application Data Types

The user is responsible for ensuring that data passed into and out of OpenCL buffers are natively aligned relative to the start of the buffer per requirements in *section 6.1.5*. As well, the user is responsible to ensure that data passed into and out of OpenCL images are properly aligned to the granularity of the data representing a single pixel (e.g. `image_num_channels` * `sizeof(image_channel_data_type)`) except for CL_RGB and CL_RGBx images where the data must be aligned to the granularity of a single channel in a pixel (i.e. `sizeof(image_channel_data_type)`).

OpenCL makes no requirement about the alignment of OpenCL application defined data types outside of buffers and images, except that the underlying vector primitives (e.g. `__cl_float4`) where defined shall be directly accessible as such using appropriate named fields in the `cl_type` union (see *section C.5*). Nevertheless, it is recommended that the **cl_platform.h** header should attempt to naturally align OpenCL defined application data types (e.g. `cl_float4`) according to their type.

C.4 Vector Literals

Application vector literals may be used in assignments of individual vector components. Literal usage follows the convention of the underlying application compiler.

```
cl_float2 foo = { .s[1] = 2.0f };
cl_int8 bar = {{ 2, 4, 6, 8, 10, 12, 14, 16 }};
```

C.5 Vector Components

The components of application vector types can be addressed using the `<vector_name>.s[<index>]` notation.

For example:

```
foo.s[0] = 1.0f; // Sets the 1st vector component of foo
pos.s[6] = 2; // Sets the 7th vector component of bar
```

In some cases vector components may also be accessed using the following notations. These notations are not guaranteed to be supported on all implementations, so their use should be accompanied by a check of the corresponding preprocessor symbol.

C.5.1 Named vector components notation

Vector data type components may be accessed using the `.sN`, `.sn` or `.xyzw` field naming convention, similar to how they are used within the OpenCL language. Use of the `.xyzw` field naming convention only allows accessing of the first 4 component fields. Support of these notations is identified by the CL_HAS_NAMED_VECTOR_FIELDS preprocessor symbol. For example:

```
#ifdef CL_HAS_NAMED_VECTOR_FIELDS
     cl_float4 foo;
     cl_int16  bar;
     foo.x = 1.0f;  // Set first component
     foo.s0 = 1.0f; // Same as above
     bar.z = 3;     // Set third component
     bar.se = 11;   // Same as bar.s[0xe]
     bar.sD = 12;   // Same as bar.s[0xd]
#endif
```

Unlike the OpenCL language type usage of named vector fields, only one component field may be accessed at a time. This restriction prevents the ability to swizzle or replicate components as is possible with the OpenCL language types. Attempting to access beyond the number of components for a type also results in a failure.

```
foo.xy    // illegal - illegal field name combination
bar.s1234 // illegal - illegal field name combination
foo.s7    // illegal - no component s7
```

C.5.2 High/Low vector component notation

Vector data type components may be accessed using the `.hi` and `.lo` notation similar to that supported within the language types. Support of this notation is identified by the CL_HAS_HI_LO_VECTOR_FIELDS preprocessor symbol. For example:

```
#ifdef CL_HAS_HI_LO_VECTOR_FIELDS
     cl_float4 foo;
     cl_float2 new_hi = 2.0f, new_lo = 4.0f;
     foo.hi = new_hi;
     foo.lo = new_lo;
#endif
```

C.5.3 Native vector type notation

Certain native vector types are defined for providing a mapping of vector types to architecturally builtin vector types. Unlike the above described application vector types, these native types are supported on a limited basis depending on the supporting architecture and compiler.

These types are not unions, but rather convenience mappings to the underlying architectures' builtin vector types. The native types share the name of their application counterparts but are preceded by a double underscore "__".

For example, `__cl_float4` is the native builtin vector type equivalent of the `cl_float4` application vector type. The `__cl_float4` type may provide direct access to the architectural builtin `__m128` or vector float type, whereas the `cl_float4` is treated as a union.

In addition, the above described application data types may have native vector data type members for access convenience. The native components are accessed using the `.vN` sub-vector notation, where `N` is the number of elements in the sub-vector. In cases where the native type is a subset of a larger type (more components), the notation becomes an index based array of the sub-vector type.

Support of the native vector types is identified by a __CL_TYPEN__ preprocessor symbol matching the native type name. For example:

```
#ifdef __CL_FLOAT4__ // Check for native cl_float4 type
     cl_float8 foo;
     __cl_float4 bar; // Use of native type
     bar = foo.v4[1]; // Access the second native float4
     vector
#endif
```

C.6 Implicit Conversions

Implicit conversions between application vector types are not supported.

C.7 Explicit Casts

Explicit casting of application vector types (`cl_typen`) is not supported. Explicit casting of native vector types (`__cl_typen`) is defined by the external compiler.

C.8 Other operators and functions

The behavior of standard operators and function on both application vector types (`cl_typen`) and native vector types (`__cl_typen`) is defined by the external compiler.

C.9 Application constant definitions

In addition to the above application type definitions, the following literal defintions are also available.

`CL_CHAR_BIT`	Bit width of a character
`CL_SCHAR_MAX`	Maximum value of a type `cl_char`
`CL_SCHAR_MIN`	Minimum value of a type `cl_char`
`CL_CHAR_MAX`	Maximum value of a type `cl_char`
`CL_CHAR_MIN`	Minimum value of a type `cl_char`
`CL_UCHAR_MAX`	Maximum value of a type `cl_uchar`
`CL_SHORT_MAX`	Maximum value of a type `cl_short`
`CL_SHORT_MIN`	Minimum value of a type `cl_short`
`CL_USHORT_MAX`	Maximum value of a type `cl_ushort`
`CL_INT_MAX`	Maximum value of a type `cl_int`
`CL_INT_MIN`	Minimum value of a type `cl_int`
`CL_UINT_MAX`	Maximum value of a type `cl_uint`
`CL_LONG_MAX`	Maximum value of a type `cl_long`
`CL_LONG_MIN`	Minimum value of a type `cl_long`
`CL_ULONG_MAX`	Maximum value of a type `cl_ulong`
`CL_FLT_DIAG`	Number of decimal digits of precision for the type `cl_float`
`CL_FLT_MANT_DIG`	Number of digits in the mantissa of type `cl_float`
`CL_FLT_MAX_10_EXP`	Maximum positive integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type `cl_float`
`CL_FLT_MAX_EXP`	Maximum exponent value of type `cl_float`
`CL_FLT_MIN_10_EXP`	Minimum negative integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type `cl_float`
`CL_FLT_MIN_EXP`	Minimum exponent value of type `cl_float`
`CL_FLT_RADIX`	Base value of type `cl_float`
`CL_FLT_MAX`	Maximum value of type `cl_float`
`CL_FLT_MIN`	Minimum value of type `cl_float`
`CL_FLT_EPSILON`	Minimum positive floating-point number of type `cl_float` such that `1.0 + CL_FLT_EPSILON != 1` is true.
`CL_DBL_DIG`	Number of decimal digits of precision for the type `cl_double`
`CL_DBL_MANT_DIG`	Number of digits in the mantissa of type `cl_double`
`CL_DBL_MAX_10_EXP`	Maximum positive integer such that 10 raised to this

	power minus one can be represented as a normalized floating-point number of type `cl_double`
CL_DBL_MAX_EXP	Maximum exponent value of type `cl_double`
CL_DBL_MIN_10_EXP	Minimum negative integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type `cl_double`
CL_DBL_MIN_EXP	Minimum exponent value of type `cl_double`
CL_DBL_RADIX	Base value of type `cl_double`
CL_DBL_MAX	Maximum value of type `cl_double`
CL_DBL_MIN	Minimum value of type `cl_double`
CL_DBL_EPSILON	Minimum positive floating-point number of type `cl_double` such that `1.0 + CL_DBL_EPSILON != 1` is true.
CL_NAN	Macro expanding to a value representing NaN
CL_HUGE_VALF	Largest representative value of type `cl_float`
CL_HUGE_VAL	Largest representative value of type `cl_double`
CL_MAXFLOAT	Maximum value of type `cl_float`
CL_INFINITY	Macro expanding to a value represnting infinity

These literal definitions are defined in the **cl_platform.h** header.

Appendix D — OpenCL C++ Wrapper API

The OpenCL C++ wrapper API provides a C++ interface to the platform and runtime API. The C++ wrapper is built on top of the OpenCL 1.1 C API (platform and runtime) and is not a replacement. It is **required** that any implementation of the C++ wrapper API will make calls to the underlying C API and it is assumed that the C API is a compliant implementation of the OpenCL 1.1 specification.

Refer to the OpenCL C++ Wrapper API specification for details. The OpenCL C++ Wrapper API specification can be found at http://www.khronos.org/registry/cl/.

Appendix E – Changes

E.1 Summary of changes from OpenCL 1.0

The following features are added to the OpenCL platform layer and runtime (*sections 4 and 5*):

- Following queries to *table 4.3*
  - CL_DEVICE_NATIVE_VECTOR_WIDTH_{CHAR | SHORT | INT | LONG | FLOAT | DOUBLE | HALF}
  - CL_DEVICE_HOST_UNIFIED_MEMORY
  - CL_DEVICE_OPENCL_C_VERSION

- CL_CONTEXT_NUM_DEVICES to the list of queries specified to **clGetContextInfo**.

- Optional image formats: CL_Rx, CL_RGx and CL_RGBx.

- Support for sub-buffer objects – ability to create a buffer object that refers to a specific region in another buffer object using **clCreateSubBuffer**.

- **clEnqueueReadBufferRect**, **clEnqueueWriteBufferRect** and **clEnqueueCopyBufferRect** APIs to read from, write to and copy a rectangular region of a buffer object respectively.

- **clSetMemObjectDestructorCallback** API to allow a user to register a callback function that will be called when the memory object is deleted and its resources freed.

- Options that control the OpenCL C version used when building a program executable. These are described in *section 5.6.3.5*.

- CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE to the list of queries specified to **clGetKernelWorkGroupInfo**.

- Support for user events. User events allow applications to enqueue commands that wait on a user event to finish before the command is executed by the device. Following new APIs are added **- clCreateUserEvent** and **clSetUserEventStatus**.

- **clSetEventCallback** API to register a callback function for a specific command execution status.

The following modifications are made to the OpenCL platform layer and runtime (*sections 4 and 5*):

- Following queries in *table 4.3*

  - CL_DEVICE_MAX_PARAMETER_SIZE from 256 to 1024 bytes
  - CL_DEVICE_LOCAL_MEM_SIZE from 16 KB to 32 KB.

- The *global_work_offset* argument in **clEnqueueNDRangeKernel** can be a non-NULL value.

- All API calls except **clSetKernelArg** are thread-safe.

The following features are added to the OpenCL C programming language (*section 6*):

- 3-component vector data types.

- New built-in functions
  - **get_global_offset** work-item function defined in *section 6.11.1*.
  - **minmag**, **maxmag** math functions defined in *section 6.11.2*.
  - **clamp** integer function defined in *section 6.11.3*.
  - (vector, scalar) variant of integer functions **min** and **max** in *section 6.11.3*.
  - **async_work_group_strided_copy** defined in *section 6.11.10*.
  - **vec_step**, **shuffle** and **shuffle2** defined in *section 6.11.12*.

- **cl_khr_byte_addressable_store** extension is a core feature.

- **cl_khr_global_int32_base_atomics**, **cl_khr_global_int32_extended_atomics**, **cl_khr_local_int32_base_atomics** and **cl_khr_local_int32_extended_atomics** extensions are core features. The built-in atomic function names are changed to use the **atomic_** prefix instead of **atom_**.

- Macros CL_VERSION_1_0 and CL_VERSION_1_1.

The following features in OpenCL 1.0 are deprecated:

- The **clSetCommandQueueProperty** API is no longer supported in OpenCL 1.1.
- The `__ROUNDING_MODE__` macro is no longer supported in OpenCL C 1.1.

The following new extensions are added to *section 9*:

- **cl_khr_gl_event** – Creating a CL event object from a GL sync object.
- **cl_khr_d3d10_sharing** – Sharing memory objects with Direct3D 10.

The following modifications are made to the OpenCL ES Profile described in *section 10*:

- 64-bit integer support is optional.