The **OpenCL** Specification

Version: 1.0

Document Revision: 32

Khronos OpenCL Working Group

Editor: Aaftab Munshi

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:

Andrzej Mamona, AMD
Benedict Gaster, AMD
Bill Licea Kane, AMD
David Garcia, AMD
Ed Buckingham, AMD
Jan Civlin, AMD
Laurent Morichetti, AMD
Mark Fowler, AMD
Michael Houston, AMD
Michael Mantor, AMD
Norm Rubin, AMD
Robert Simpson, AMD
Aaftab Munshi, 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
Travis Brown, Apple
Andrew Cox, ARM
Dave Shreiner, ARM
Eivind Liland, 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
Brian Murray, Freescale
Barry Minor, IBM
Brian Watt, IBM
Dan Brokenshire, 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
Ofer Rosenberg, Intel
Paul Lalonde, Intel
Stephen Junkins, Intel
Tim Foley, Intel
Timothy Mattson, Intel
Bill Bush, Kestrel Institute
Lindsay Errington, Kestrel Institute
Jon Leech, Khronos
Cormac Brick, Movidia
David Donohoe, Movidia
Jyrki Leskelä, Nokia
Kari Pulli, Nokia
Amit Rao, NVIDIA
Chris Cameron, NVIDIA
Christopher Lamb, NVIDIA
Ian Buck, NVIDIA
Jason Sanders, NVIDIA
Mark Harris, NVIDIA
Neil Trevett, NVIDIA
Alex Bourd, Qualcomm
Michael McCool, RapidMind

Stefanus Du Toit, RapidMind
Jonathan Grant, Renesas
Robert Schulman, Seaweed Systems
John Bates, Sony
Ajit Kamat, Symbian
Madhukar Budagavi, Texas Instruments
Tom Olson, 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 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.

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 Objects* and *Image Objects*.

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*.

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.

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*.

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).

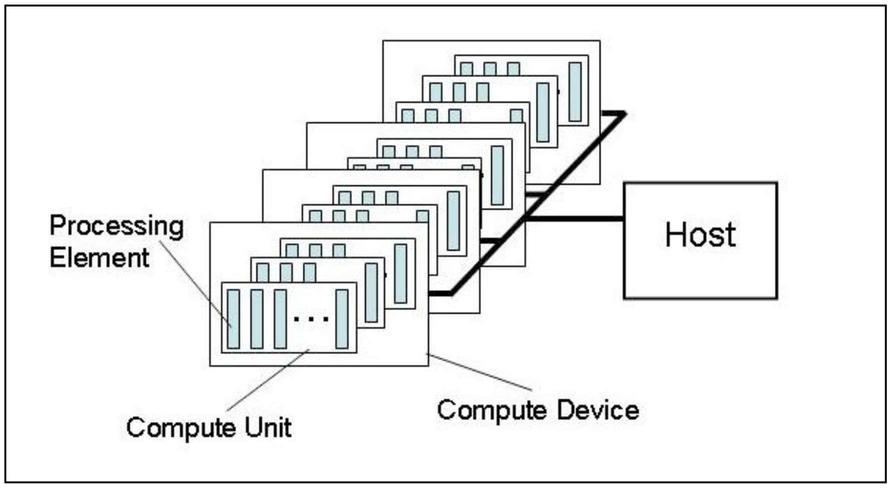


Figure 3.1: *Platform model … one host plus one or more compute devices each with one or more compute units each with one or more processing elements.*

3.2 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 1.0 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. Each work-item's global ID and local ID are N-dimensional tuples. The global ID components are values in the range from zero to 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 (Gx, Gy) and the size of each work-group (Sx, Sy). 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

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

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:

$$(w_x, w_y) = ((g_x – s_x) / S_x, (g_y – s_y) / S_y)$$

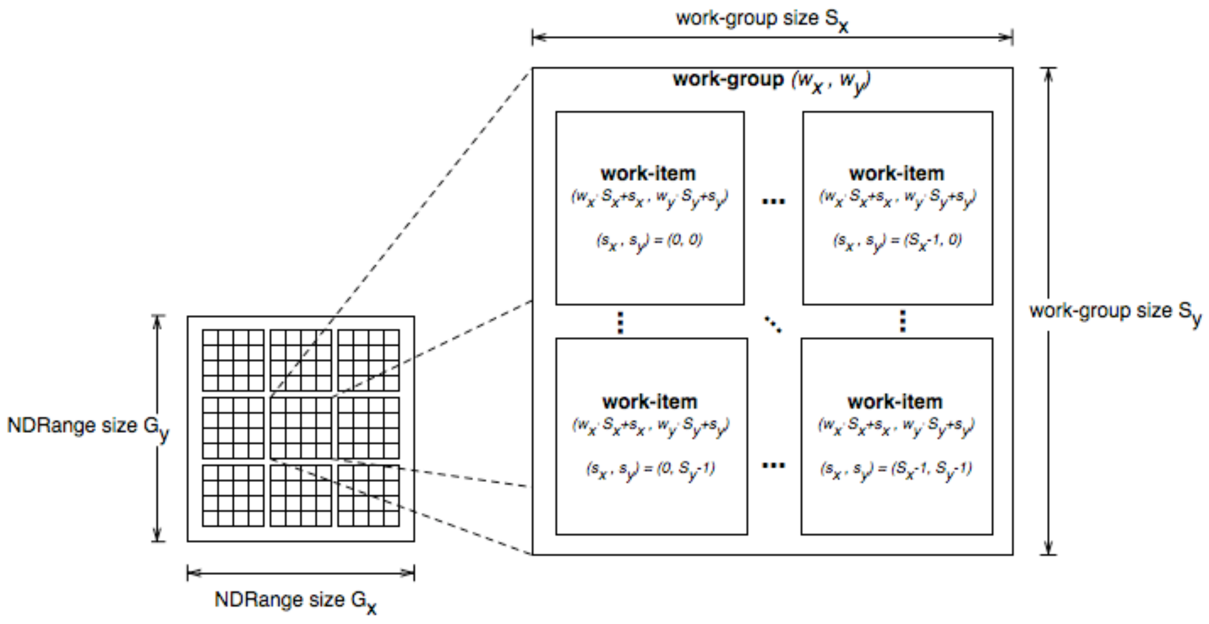

work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

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*.

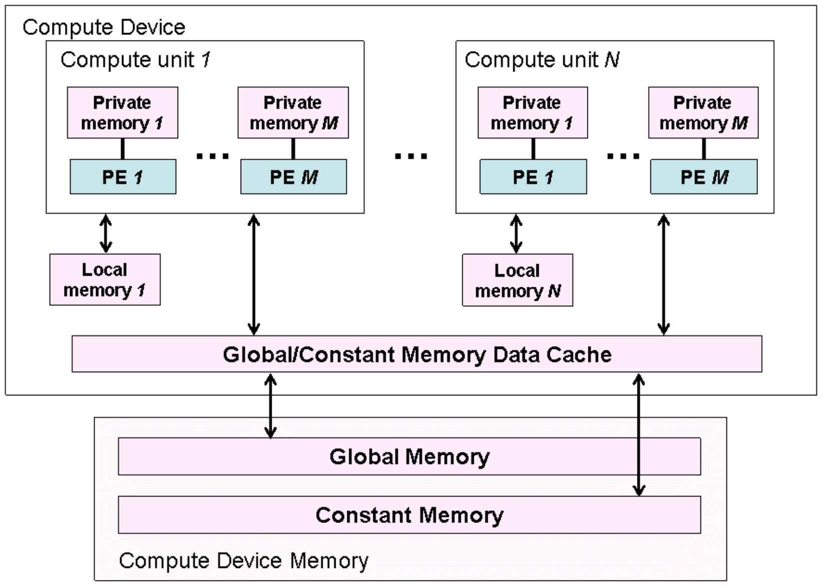


Figure 3.3: *Conceptual OpenCL device architecture with processing elements (PE), compute units and devices. The host is not shown.*

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 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 function

```
cl_int   clGetPlatformInfo (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*.

param_name is an enum 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[][1]	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

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

		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 1.0 is described in *section 10*.
CL_PLATFORM_VERSION	char[]	OpenCL version string. Returns the OpenCL version supported by the implementation. This version string has the following format: *OpenCL<space><major_version.minor_version><space><platform-specific information>* The *major_version.minor_version* value returned will be 1.0.

Table 4.1. *OpenCL Platform Queries*

clGetPlatformInfo returns CL_SUCCESS if the function is executed successfully. It returns 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.

4.2 Querying Devices

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

```
cl_int    clGetDeviceIDs (cl_device_type device_type,
                          cl_uint num_entries,
                          cl_device_id *devices,
                          cl_uint *num_devices)
```

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 mininum 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_INVALID_DEVICE_TYPE if *device_type* is not a valid value, returns CL_INVALID_VALUE if *num_entries* is equal to zero and *devices* is not NULL or if both *num_devices* and *devices* are NULL, returns CL_DEVICE_ NOT_FOUND if no OpenCL devices that matched *device_type* were found, and returns CL_SUCCESS if the function is executed successfully.

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 enum 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 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_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.
CL_DEVICE_MAX_CLOCK_FREQUENCY	cl_uint	Maximum configured clock frequency of the device in MHz.
CL_DEVICE_ADDRESS_BITS	cl_bitfield	Describes the address spaces supported by the device. This is a bitfield and can be set to a combination of the following values: CL_DEVICE_ADDRESS_32_BITS – device supports a 32-bit address space. CL_DEVICE_ADDRESS_64_BITS – device supports a 64-bit address space.
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 6.11.8* 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 256.
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. 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 qualifer 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 16 KB.

CL_DEVICE_ERROR_CORRECTION_SUPPORT	cl_bool	Is CL_TRUE if the device implements error correction for the memories, caches, registers etc. in the device. Is CL_FALSE if the device does not implement error correction. This can be a requirement for certain clients of OpenCL.
CL_DEVICE_PROFILING_TIMER_RESOLUTION	size_t	Describes the resolution of device timer. This is measured in nanoseconds. Refer to *section 5.9* 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 OpenCL ES 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_	Describes the command-queue

	queue_properties	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_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	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.
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 1.0.
CL_DEVICE_EXTENSIONS	char[]	Returns a space separated list of extension names (the extension names

		themselves do not contain any spaces). The list of extension names returned currently can include one or more of the following approved extension names: **cl_khr_fp64** **cl_khr_select_fprounding_mode** **cl_khr_global_int32_base_atomics** **cl_khr_global_int32_extended_atomics** **cl_khr_local_int32_base_atomics** **cl_khr_local_int32_extended_atomics** **cl_khr_int64_base_atomics** **cl_khr_int64_extended_atomics** **cl_khr_3d_image_writes** **cl_khr_byte_addressable_store** **cl_khr_fp16** 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. It returns CL_INVALID_DEVICE if *device* is not valid, returns 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.

4.3 Contexts

The function

```
cl_context    clCreateContext (cl_context_properties properties,
                               cl_uint num_devices,
                               const cl_device_id *devices,
                               void (*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 is reserved and must be zero.

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

devices is a pointer to a list of unique devices returned by **clGetDeviceIDs**. If more than one device is specified in *devices*, an implementation-defined[2] selection criteria may be applied to determine if the list of devices specified can be used together to create a context.

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

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

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

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

[2] Some implementations may require that the list of devices specified to **clCreateContext** support the same OpenCL profile (CL_DEVICE_PROFILE) and/or version (CL_DEVICE_VERSION).

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. It returns a NULL value with the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_VALUE if *properties* is not zero.
- *errcode_ret* returns CL_INVALID_VALUE if *devices* is NULL.
- *errcode_ret* returns CL_INVALID_VALUE if *num_devices* is equal to zero.
- *errcode_ret* returns CL_INVALID_DEVICE if *devices* contains an invalid device.
- *e*rrcode_ret returns CL_INVALID_DEVICE_LIST if more than one device is specified in *devices* and the list of devices specified cannot be used together to create a context.
- *errcode_ret* returns CL_DEVICE_NOT_AVAILABLE if a device in *devices* is currently not available even though the device was returned by **clGetDeviceIDs**.
- *errcode_ret* returns 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 (cl_context_properties properties,
                                 cl_device_type device_type,
                                 void (*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 is reserved and must be zero.

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. It returns a NULL value with the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_VALUE if *properties* is not zero.
- *errcode_ret* returns CL_INVALID_DEVICE_TYPE if *device_type* is not a valid value.
- *errcode_ret* returns CL_DEVICE_NOT_AVAILABLE if no devices that match *device_type* are currently available.
- errcode_ret returns CL_DEVICE_NOT_FOUND if no devices that match *device_type* were found.
- *errcode_ret* returns 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. It returns CL_INVALID_CONTEXT if *context* is not a valid OpenCL context.

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. It returns CL_INVALID_CONTEXT if *context* is not a valid OpenCL context.

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 enum 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 4.4*.

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 4.4*.

cl_context_info	**Return Type**	**Information returned in param_value**
CL_CONTEXT_REFERENCE_COUNT[3]	cl_uint	Return the *context* reference count.
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**.

Table 4.4 *List of supported param_names by* **clGetContextInfo**

clGetContextInfo returns CL_SUCCESS if the function is executed successfully. It returns CL_INVALID_CONTEXT if *context* is not a valid context, returns CL_INVALID_VALUE if

[3] 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 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 4.4* and *param_value* is not a NULL value.

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, writing a memory object.

5.1 Command Queues

OpenCL objects such as memory objects, 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 5.8*.
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 5.9*.

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 the device type specified when the *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*.

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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context.
- *errcode_ret* returns CL_INVALID_DEVICE if *device* is not a valid device or is not associated with *context*.
- *errcode_ret* returns CL_INVALID_VALUE if values specified in *properties* are not valid.
- errcode_ret returns CL_INVALID_QUEUE_PROPERTIES if values specified in *properties* are valid but are not supported by the device.
- *errcode_ret* returns 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. It returns CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.

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

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.

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[4]	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**, and can be changed by **clSetCommandQueueProperty**.

Table 5.2 *List of supported param_names by clGetCommandQueueInfo*

clGetCommandQueueInfo returns CL_SUCCESS if the function is executed successfully. It returns CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue, returns 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.

The function

cl_int **clSetCommandQueueProperty** (cl_command_queue *command_queue*,
cl_command_queue_properties *properties*,
cl_bool *enable*,
cl_command_queue_properties **old_properties*)

can be used to enable or disable the properties of a command-queue.

command_queue specifies the command-queue being queried.

properties specifies the new command-queue properties to be applied to *command_queue* .

enable determines whether the values specified by *properties* are enabled (if *enable* is CL_TRUE) or disabled (if *enable* is CL_FALSE) for the command-queue . The property values are described in *table 5.1*.

old_properties returns the command-queue properties before they were changed by **clSetCommandQueueProperty**. If *old_properties* is NULL, it is ignored.

As specified in table 5.1, the **CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE** command-queue property determines whether the commands in a command-queue are executed in-order or out-of-order. Changing this command-queue property will cause the OpenCL implementation to block until all previously queued commands in *command_queue* have completed. This can be an expensive operation and therefore changes to the

[4] 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_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property should be only done when absolutely necessary.

clSetCommandQueueProperty returns CL_SUCCESS if the command-queue properties are successfully updated. It returns CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue, returns CL_INVALID_VALUE if the values specified in *properties* are not valid and returns CL_INVALID_QUEUE_PROPERTIES if values specified in *properties* are not supported by the device.

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 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.

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 flags 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. A request by the application to read or write to the memory

	pointed by *host_ptr* (refer to **clEnqueueMap{Buffer|Image}** functions) will be guaranteed to contain the latest bits. 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_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. Passing in a pointer to an already allocated buffer on the host and using it as a buffer object allows applications to share data efficiently with kernels and the host.

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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context.

- *errcode_ret* returns CL_INVALID_VALUE if values specified in *flags* are not valid.

- *errcode_ret* returns CL_INVALID_BUFFER_SIZE if *size* is 0 or is greater than CL_DEVICE_MAX_MEM_ALLOC_SIZE value specified in *table 4.3*.

- *errcode_ret* returns 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*.

- *errorcode_ret* returns CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for buffer object.

- *errcode_ret* returns CL_INVALID_OPERATION if the buffer object cannot be created for all devices in *context*.

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

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.

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

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.

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.
- 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 buffer memory objects.
- 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_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.2.3 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. It returns CL_INVALID_MEM_OBJECT if *memobj* is not a valid memory object. **clCreateBuffer** and **clCreateImage{2D|3D}** perform an implicit retain.

The function

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

decrements the *memobj* reference count. 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. **clReleaseMemObject** returns CL_SUCCESS if the function is executed successfully. It returns CL_INVALID_MEM_OBJECT if *memobj* is not a valid memory object.

5.2.4 Creating Image Objects

An **image** (1D, or 2D) 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 5.2.4.1* 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. Passing in a pointer to an already allocated buffer on the host and using it as a memory object allows applications to share data efficiently with kernels and the host. 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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context.
- *errcode_ret* returns CL_INVALID_VALUE if values specified in *flags* are not valid.
- *errcode_ret* returns CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in *image_format* are not valid or if *image_format* is NULL.
- *errcode_ret* returns 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 or if values specified by *image_row_pitch* do not follow rules described in the argument description above.
- *errcode_ret* returns 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*.
- *errcode_ret* returns CL_IMAGE_FORMAT_NOT_SUPPORTED if the *image_format* is not supported.
- *errorcode_ret* returns CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for image object.
- *errcode_ret* returns CL_INVALID_OPERATION if the image object as specified by the *image_format*, *flags* and dimensions cannot be created for all devices in *context* that support images or if there are no devices in *context* that support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- *errcode_ret* returns 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 5.2.4.1* 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. Passing in a pointer to an already allocated buffer on the host and using it as a memory object allows applications to share data efficiently with kernels and the host. 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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context.
- *errcode_ret* returns CL_INVALID_VALUE if values specified in *flags* are not valid.
- *errcode_ret* returns CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in *image_format* are not valid or if *image_format* is NULL.
- *errcode_ret* returns CL_INVALID_IMAGE_SIZE if *image_width*, *image_height* or *image_depth* are 0 or if they exceed values specified in CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_MAX_HEIGHT or CL_DEVICE_IMAGE3D_MAX_DEPTH respectively or if values specified by *image_row_pitch* and *image_slice_pitch* do not follow rules described in the argument description above.
- *errcode_ret* returns 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*.
- *errcode_ret* returns CL_IMAGE_FORMAT_NOT_SUPPORTED if the *image_format* is not supported.
- *errorcode_ret* returns CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for image object.
- *errcode_ret* returns CL_INVALID_OPERATION if the image object as specified by the *image_format*, *flags* and dimensions cannot be created for all devices in *context* that support images, or if there are no devices in *context* that support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- *errcode_ret* returns CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.2.4.1 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 5.4.*

Enum values that can be specified in channel_order
CL_R, or CL_A
CL_RG, or CL_RA
CL_RGB. This format can only be used if channel data type = CL_UNORM_SHORT_565, CL_UNORM_SHORT_555 or CL_UNORM_INT101010.
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 5.4 *List of supported Image Channel Order Values*

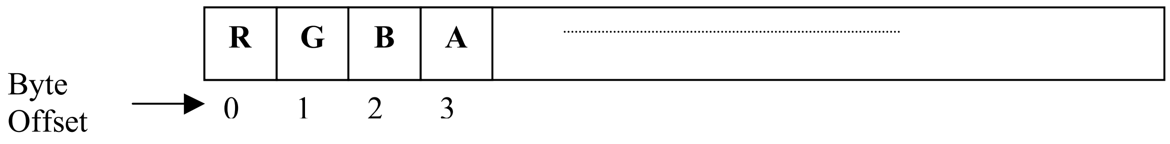
`image_channel_data_type` describes the size of the channel data type. The list of supported values is described in *table 5.5*. The number of bits per element determined by the `image_channel_data_type` and `image_channel_order` must be a power of two.

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.
CL_UNORM_SHORT_555	Represents a normalized x-5-5-5 4-channel xRGB image. The channel order must be CL_RGB.

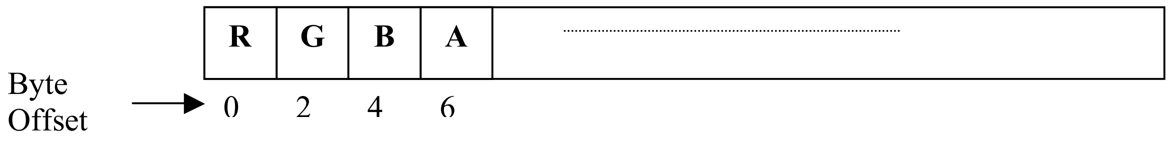
CL_UNORM_INT_101010	Represents a normalized x-10-10-10 4-channel xRGB image. The channel order must be CL_RGB.
CL_SIGNED_INT8	Each channel component is an unnormalized signed 8-bit integer value
CL_SIGNED_INT16	Each channel component is an unnormalized signed 16-bit integer value
CL_SIGNED_INT32	Each 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 5.5 *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.

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

5.2.5.1 Minimum List of Supported Image Formats

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*. In addition, the following list of image formats must be supported by the OpenCL implementation.

For read-only 2D and 3D images, the minimum list of supported image formats is described in *table 5.6* below:

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 5.6 *Min. list of supported image formats for read-only images*

For read-write or write-only 2D images, the minimum list of supported image formats is described in *table 5.7* below:

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 for read-write images*

5.2.6 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.

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

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.

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

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

5.2.7 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.

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.
- 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_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.

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

5.2.8 Mapping and Unmapping Memory 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.

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 if *buffer* is created with a *host_ptr* and CL_MEM_USE_HOST_PTR is set in *flags* arguments to **clCreateBuffer** or if *buffer* is created with CL_MEM_ALLOC_HOST_PTR set in *flags* argument to **clCreateBuffer** and the region specified by (*offset*, *cb*) is a valid region in the buffer object. 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*:

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

- *errcode_ret* returns CL_INVALID_CONTEXT if context associated with *command_queue* and *buffer* are not the same.

- *errcode_ret* returns CL_INVALID_MEM_OBJECT if *buffer* is not a valid buffer object.

- *errcode_ret* returns 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.

- *errcode_ret* returns 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.

- *errcode_ret* returns 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 atleast *cb* bytes in size. The result of a memory access outside this region is undefined.

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.

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 if *image* is created with a *host_ptr* and CL_MEM_USE_HOST_PTR is set in *flags* arguments to **clCreateImage{2D|3D}** or if *image* is created with CL_MEM_ALLOC_HOST_PTR set in *flags* argument to **clCreateImage{2D|3D}** and the 2D or 3D rectangle specified by *origin* and *region* is a valid region in the image object. 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*:

- *errcode_ret* returns CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- *errcode_ret* returns CL_INVALID_CONTEXT if context associated with *command_queue* and *image* are not the same.
- *errcode_ret* returns CL_INVALID_MEM_OBJECT if *image* is not a valid image object.
- *errcode_ret* returns 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.
- *errcode_ret* returns 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.
- *errcode_ret* returns CL_INVALID_VALUE if *image_row_pitch* is NULL.
- *errcode_ret* returns CL_INVALID_VALUE if *image* is a 3D image object and *image_slice_pitch* is NULL.
- *errcode_ret* returns 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.
- *errcode_ret* returns 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 atleast (*image_row_pitch * region[1] + region[0])* pixels in size for a 2D image, and is atleast (*image_slice_pitch * region[2] + image_row_pitch * region[1] + region[0])* pixels in size for a 3D image. The result of a memory access outside this region is undefined.

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.

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_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.

clEnqueueMapBuffer and **clEnqueueMapImage** increments the mapped count of the memory object. 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 memory object being mapped.

5.2.8.1 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 buffer or image 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.

5.2.9 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 5.8*.

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.

clGetMemObjectInfo returns CL_SUCCESS if the function is executed successfully, returns 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, and returns CL_INVALID_MEM_OBJECT if *memobj* is a not a valid memory object.

cl_mem_info	Return type	Info. returned in *param_value*
CL_MEM_TYPE	cl_mem_object_type	Returns CL_MEM_OBJECT_BUFFER if *memobj* is created with **clCreateBuffer**, CL_MEM_OBJECT_IMAGE2D if *memobj* is created with **clCreateImage2D** or CL_MEM_OBJECT_IMAGE3D if *memobj* is created with **clCreateImage3D**.
CL_MEM_FLAGS	cl_mem_flags	Return the *flags* argument value specified when *memobj* is created with **clCreateBuffer** or **clCreateImage{2D|3D}**.
CL_MEM_SIZE	size_t	Return actual size of *memobj* in bytes.
CL_MEM_HOST_PTR	void *	Return the *host_ptr* argument value specified when *memobj* is created.
CL_MEM_MAP_COUNT[5]	cl_uint	Map count.
CL_MEM_REFERENCE_COUNT[6]	cl_uint	Return *memobj* reference count.
CL_MEM_CONTEXT	cl_context	Return context specified when memory object is created

Table 5.8 *List of supported param_names by clGetMemObjectInfo*

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

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

[6] 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_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 5.9*.

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.

clGetImageInfo returns CL_SUCCESS if the function is executed successfully, returns 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, and returns CL_INVALID_MEM_OBJECT if *image* is a not a valid image object.

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, depth = 0.

Table 5.9 *List of supported param_names by clGetImageInfo*

5.3 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.

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 6.11.8.1* 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_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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context.
- *errcode_ret* returns CL_INVALID_VALUE if *addressing_mode, filter_mode* or *normalized_coords* or combination of these argument values are not valid.
- *errcode_ret* returns 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).

- *errcode_ret* returns 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** does an implicit retain. **clRetainSampler** returns CL_SUCCESS if the function is executed successfully. It returns CL_INVALID_SAMPLER if *sampler* is not a valid sampler object.

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. It returns CL_INVALID_SAMPLER if *sampler* is not a valid sampler object.

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[7]	cl_uint	Return the *sampler* reference count.
CL_SAMPLER_CONTEXT	cl_context	Return the context specified when the sampler is created.
CL_SAMPLER_ADDRESSING_ MODE	cl_addressing_ mode	Return the value specified by *addressing_mode* argument to **clCreateSampler**.
CL_SAMPLER_FILTER_MODE	cl_filter_mode	Return the value specified by *filter_mode* argument to **clCreateSampler**.
CL_SAMPLER_NORMALIZED_ COORDS	cl_bool	Return the value specified by *normalized_coords* argument to **clCreateSampler**.

Table 5.10 *clGetSamplerInfo parameter queries.*

clGetSamplerInfo returns CL_SUCCESS if the function is executed successfully, returns 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, and returns CL_INVALID_SAMPLER if *sampler* is a not a valid sampler object.

[7] 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.

5.4 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.

5.4.1 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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context.
- *errcode_ret* returns CL_INVALID_VALUE if *count* is zero or if *strings* or any entry in *strings* is NULL.
- *errcode_ret* returns CL_COMPILER_NOT_AVAILABLE if a compiler is not available i.e. CL_DEVICE_COMPILER_AVAILABLE specified in *table 4.3* is set to CL_FALSE.
- *errcode_ret* returns 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 void **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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context.
- *errcode_ret* returns CL_INVALID_VALUE if *device_list* is NULL or *num_devices* is zero.
- errcode_ret returns CL_INVALID_DEVICE if OpenCL devices listed in *device_list* are not in the list of devices associated with *context*.
- *errcode_ret* returns CL_INVALID_VALUE if *lengths* or *binaries* are NULL or if any entry in *lengths*[i] is zero or *binaries*[i] is NULL.
- *errcode_ret* returns CL_INVALID_BINARY if an invalid program binary was encountered for any device. *binary_status* will return specific status for each device.
- *errcode_ret* returns 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. It returns CL_INVALID_PROGRAM if *program* is not a valid program object.

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. It returns CL_INVALID_PROGRAM if *program* is not a valid program object.

5.4.2 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 (*pfn_notify)(cl_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.

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 string that describes the build options to be used for building the program executable. The list of supported options is described *in section 5.4.3*.

pfn_notify is a function pointer to a notification routine. The notification routine allows an application to register a callback function 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_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_BUILD_ERROR 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_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.4.3 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.

5.4.3.1 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.

5.4.3.2 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 floating-point constant as single precision constant instead of implicitly converting it to double 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 and if the optional extension for double precision is supported, double precision denormalized numbers may also be flushed to zero. 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 precison but 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.

5.4.3.3 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 behavior of distinct `+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 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.

5.4.3.4 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.

5.4.4 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.

5.4.5 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[8]	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 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**.

[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.

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	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 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.

Table 5.11 *clGetProgramInfo parameter queries.*

clGetProgramInfo returns CL_SUCCESS if the function is executed successfully, returns 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, and returns CL_INVALID_PROGRAM if *program* is a not a valid program object.

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, returns CL_INVALID_DEVICE if *device* is not in the list of devices associated with *program*, returns 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, and returns CL_INVALID_PROGRAM if *program* is a not a valid program object.

5.5 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.

5.5.1 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` qualifer.

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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_PROGRAM if *program* is not a valid program object.
- errcode_ret returns CL_INVALID_PROGRAM_EXECUTABLE if there is no successfully built executable for *program*.
- *errcode_ret* returns CL_INVALID_KERNEL_NAME if *kernel_name* is not found in *program*.
- *errcode_ret* returns 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.
- *errcode_ret* returns CL_INVALID_VALUE if *kernel_name* is NULL.
- *errcode_ret* returns 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, returns CL_INVALID_PROGRAM if *program* is not a valid program object, returns CL_INVALID_PROGRAM_EXECUTABLE if there is no successfully built executable for any device in *program*, returns CL_INVALID_VALUE if *kernels* is not NULL and *num_kernels* is less than the number of kernels in *program* and returns 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. It returns CL_INVALID_KERNEL if *kernel* is not a valid kernel object. **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. It returns CL_INVALID_KERNEL if *kernel* is not a valid kernel object. The kernel object is deleted once the number of instances that are retained to *kernel* become zero and after all queued execution instances of *kernel* have finished.

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

The memory object specified as argument value must be a buffer object (or NULL) if the argument is declared to be a pointer of a built-in or user defined type with the __*global* or __*constant* qualifier. 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*.

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.

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.

- 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).`

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[9].

5.5.3 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*.

[9] 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.

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[10]	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, returns 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, and returns CL_INVALID_KERNEL if *kernel* is a not a valid kernel object.

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.

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.

[10] 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_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 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 6.7.2*. If the work-group size is not specified using the above attribute qualifier (0, 0, 0) is returned.

Table 5.14 *clGetKernelWorkGroupInfo parameter queries.*

clGetKernelWorkGroupInfo returns CL_SUCCESS if the function is executed successfully, returns 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*, returns 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, and returns CL_INVALID_KERNEL if *kernel* is a not a valid kernel object.

5.6 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 three.

global_work_offset must currently be a NULL value. In a future revision of OpenCL, *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 instead of having the global IDs always start at offset (0, 0, … 0).

global_work_size 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 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 divisable 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 6.7.2*). 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 OpenCL 1.0, the starting global ID is always (0, 0, … 0). 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.

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_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_WORK_GROUP_SIZE if *local_work_size* is specified and number of work-items specified by *global_work_size* is not evenly divisable 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 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_INVALID_GLOBAL_OFFSET if *global_work_offset* is not NULL.

- 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 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_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.

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_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_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 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_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_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 *device* 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 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_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.7 Event Objects

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

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_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 function was executed successfully. It returns CL_INVALID_VALUE if num_events is zero, returns CL_INVALID_CONTEXT if events specified in *event_list* do not belong to the same context, and returns CL_INVALID_EVENT if event objects specified in *event_list* are not valid event objects.

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*.
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_EVENT_COMMAND_	cl_int	Return the execution status of the command

EXECUTION_STATUS		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.).
CL_EVENT_REFERENCE_ COUNT[11]	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.

clGetEventInfo returns CL_SUCCESS if the function is executed successfully, returns 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, and returns CL_INVALID_EVENT if *event* is a not a valid event object.

The function

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

increments the *event* reference count. **clRetainEvent** returns CL_SUCCESS if the function is executed successfully. It returns CL_INVALID_EVENT if *event* is not a valid event object. The OpenCL commands that return an event perform an implicit retain.

[11] 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.

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. It returns CL_INVALID_EVENT if *event* is not a valid event object. 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.

5.8 Out-of-order Execution of Kernels and Memory Object Commands

The OpenCL functions that are submitted to a command-queue are queued 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 queued 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 queued 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 or can be changed dynamically using **clSetCommandQueueProperty**. In out-of-order execution mode there is no guarantee that the queued 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 or a barrier function can be queued to the command-queue. The wait for events command ensures that previously queued commands identified by the list of events to wait for have finished before the next batch of commands is executed. The barrier ensures that all previously queued 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 queued 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 queue a wait for event or a barrier command can be queued that must complete before reads or writes to the memory object(s) occur.

The function

cl_int **clEnqueueMarker** (cl_command_queue *command_queue*,
cl_event **event*)

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

clEnqueueMarker rteturns CL_SUCCESS if the function is successfully executed. It returns CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue, returns CL_INVALID_VALUE if *event* is a NULL value and returns 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}**, **clEnqueueCopy{Buffer|Image}**, **clEnqueueCopyBufferToImage, clEnqueueCopyImageToBuffer** or **clEnqueueMarker**.

The events specified in *event_list* act as synchronization points.

clEnqueueWaitForEvents returns CL_SUCCESS if the function was successfully executed. It returns CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue, returns CL_INVALID_VALUE if *num_events* is zero or *event_list* is NULL, returns CL_INVALID_EVENT if event objects specified in *event_list* are not valid events and returns 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. **clEnqueueBarrier** is a synchronization point.

clEnqueueBarrier returns CL_SUCCESS if the function was executed successfully. It returns CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue and returns CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.9 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[12] being referred to are: **clEnqueue{Read|Write|Map}Buffer, clEnqueue{Read|Write|Map}Image, clEnqueueCopy{Buffer|Image}, clEnqueueCopyImageToBuffer, clEnqueueCopyBufferToImage, clEnqueueNDRangeKernel , clEnqueueTask, clEnqueueNativeKernel and clEnqueueMarker**. 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** or by setting the CL_QUEUE_PROFILING_ENABLE flag in *properties* argument to **clSetCommandQueueProperty**.

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.

[12] **clEnqueueAcquireGLObjects** and **clEnqueueReleaseGLObjects** defined in *section B.1.5* 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, returns CL_PROFILING_INFO_NOT_AVAILABLE if the CL_QUEUE_PROFILING_ENABLE flag is not set for the command-queue and if the profiling information is currently not available (because the command identified by *event* has not completed), returns 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, and returns CL_INVALID_EVENT if *event* is a not a valid event object.

5.10 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. It returns CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue and returns 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 such as **clEnqueueRead{Image|Buffer}** with *blocking_read* set to CL_TRUE, **clEnqueueWrite{Image|Buffer}** with *blocking_write* set to CL_TRUE, **clEnqueueMap{Buffer|Image}** with *blocking_map* set to CL_TRUE or **clWaitForEvents** perform an implicit flush of the command-queue.

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. It returns CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue and returns 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	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_SPACE defined in *table 3.2* is 32-bits and is a 64-bit unsigned integer if CL_DEVICE_ADDRESS_SPACE 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_SPACE defined in *table 3.2* is 32-bits and is a 64-bit signed integer if CL_DEVICE_ADDRESS_SPACE 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[13]

The char, unsigned char, short, unsigned short, integer, unsigned integer, long, unsigned long and 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 and ulong followed by a literal value *n* that defines the number of elements in the vector. Supported values of *n* are 2, 4, 8, and 16.

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***

[13] *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.*

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 6.11.8* for a detailed description of this type.
image3d_t	A 3D image. Refer to *section 6.11.8* for a detailed description of this type.
sampler_t	A sampler type. Refer to *section 6.11.8* for a detailed description of this type.
event_t	An event handle. This can be used to identify async copies from global to local memory and vice-versa. Refer to *section 6.11.11*.

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, 4, 8 and 16, are also reserved.

Type	Description
bool*n*	A boolean vector.
double, double*n*	A double precision floating-point number, and double precision 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,	A complex 128-bit floating-point number, and

complex quad*n*, imaginary quad, imaginary quad*n*	complex and imaginary 128-bit floating-point 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*

Structures of built-in scalar and vector data types, and structures of structs are supported. Arrays of built-in scalar and vector data types and/or arrays of structs are also supported. typedefs 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.

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. The behavior of a direct unaligned load/store is considered to be undefined, except for the vector data load and store functions defined in *section 6.11.7*. These vector load and store functions allow you to read and write vectors types from addresses aligned to the size of the vector type or the size of a scalar element of the vector type.

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 constant 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 `char4, uchar4, short4, ushort4, int4, uint4, long4, ulong4, and 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
```

The component selection syntax allows multiple components to be selected by appending their names after the period (**.**).

```
float4 c, a, b;

c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;                      // is a float
c.xy = (float2)(3.0f, 4.0f); // is a float2
```

The order of the components can be different to swizzle them, 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.

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

pos.xw = (float2)(5.0, 6.0);  // 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.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);
```

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`
`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`

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 10th element of the `float16` variable `x` and `x.sf (or x.sF)` refers to the 15th 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 `.odd`) and `.hi` (or `.even`) suffixes to get smaller vector types or to combine smaller vector types to a larger vector type. Multiple levels of `.lo` (or `.odd`) and `.hi` (or `.even`) 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.

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
```

The `.odd` suffix refers to the odd elements of a vector. The `.even` suffix refers to the even elements of a vector.

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.hi;
        t.odd = x.lo;
        x.even = t.hi;
        x.odd = t.lo;
        //write back
        m[0] = x.hi.hi;    // { m[0][0], m[1][0], m[2][0], m[3][0] }
        m[1] = x.hi.lo;    // { m[0][1], m[1][1], m[2][1], m[3][1] }
        m[2] = x.lo.hi;    // { m[0][2], m[1][2], m[2][2], m[3][2] }
        m[3] = x.lo.lo;    // { m[0][3], m[1][3], m[2][3], m[3][3] }
}
```

6.2 Conversions and Type Casting

6.2.1 Implicit Conversions

Implicit conversions between scalar built-in types defined in *table 6.1* are supported.

Implicit conversions are not allowed for built-in vector data types. There are no implicit array or structure conversions. For example, an array of `int` cannot be implicitly converted to an array of `float`. 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`.

6.2.2 Explicit Casts

Standard typecasts for built-in scalar data types will perform appropriate conversion. 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 example below will generate a compilation error.

```
float4    f;
int4      i = (int4) 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 rounding mode will be used for conversions to floating-point vector types.

In the examples below:

```
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).
```

6.2.3 Explicit Conversions

Explicit conversions may performed using the

```
convert_<dest type name>(srctype)
```

suite of functions. These provide a full set of type conversions between supported types (see *section 6.1.1*) except for the following types: `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 a `int4` vector `c`.

```
float   f;
int     i = convert_int(f);
```

`convert_int` converts a `float` scalar `f` to a `int` scalar `i`.

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:

destType*n* **convert_destType*n*<_sat><_roundingMode>** (sourceType*n*)

6.2.3.1 Data Types

Conversions are available for the following scalar types: `bool`, `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 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 towards 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 6.5 *Rounding Modes.*

By default, conversions to vectors of integer type use the `_rtz` (round toward zero) rounding mode and conversions to floating-point type[14] 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. When converting between integer types, the resulting value for out of range inputs will be equal to the set of least significant bits in the source operand element that fit in the corresponding destination element. When converting from a floating-point type to integer type, the behavior is implementation-defined.

[14] 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.

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;

// -ve 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 );

// 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[15]. 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 the representation of the object, then the value of the additional bytes is undefined.

Examples:

```
union{ float f; uint u; double d;} 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)
```

[15] In addition, some other C language extensions to C 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.

6.2.4.2 Reinterpreting Types Using as_typen()

All data types described in tables 6.1 and 6.2 (except `bool`, `half` and `void`) may be also reinterpreted as another data type of the same size using the **as_typen()** operator[16]. 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 a IEEE-754 single precision value.

When the operand and result type contain a different number of elements, the result shall be implementation-defined. That is, a conforming implementation shall explicitly define a 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_typen`() 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;
```

[16] While the union is intended to reflect the organization of data in memory, the **as_typen**() construct is intended to reflect the organization of data in register. The as_typen() construct is 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.

```
// 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 size
double4 g = as_double4(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 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.

6.3 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 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 is promoted and/or up-converted to the type used by the vector operand (down-conversion of the scalar type is illegal and will result in a compile time error). 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 size. In this case, the operation is done component-wise resulting in the same size vector.

   All other cases 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[17]. 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.

[17] 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. If the source operands are a vector float, the result is a vector signed integer. Either the operands' types must match, or the conversions from *section 6.2.1* **Implicit Conversions** will be applied to the integer operand, after which the types must match.

   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 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. If the operand types do not match, then there must be a conversion from *section 6.2.1* **Implicit Conversions** applied to one operand that can make them match, in which case this conversion is done. 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*, `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.

g. The logical operators and (**&&**), or (**||**) operate on all scalar and vector built-in types except the built-in scalar and vector float types. And (**&&**) will only evaluate the right hand operand if the left hand operand compares unequal to 0. Or (**||**) will only evaluate the right hand operand if the left hand operand compares equal to 0. For built-in vector types, the operators are applied component-wise.

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 is *true*, otherwise it selects to evaluate the third expression. If the result is a vector value, then this is equivalent to calling **select**(*exp2*, *exp3*, *exp1*). The **select** function is described in *table 6.13*. 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. 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 scalar data type or each component of a vector data type; 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 scalar data type or each component of a vector data type. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the empty bits are cleared. If `E1` has a signed type and a negative value, the empty bits are set.

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[18] 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 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 a function, the result is a function designator; if it 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[19].

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,

[18] Variable length arrays are not supported in OpenCL 1.0. Refer to *section 6.8.d*.

[19] 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.

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 the **[]** operator were changed to a + operator. Otherwise, the result is a pointer to the object or function designated by its operand[20].

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 6.5* in *section 6.2.1* **Implicit Conversions** that converts to the type of *lvalue*, 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 promoted and/or up-converted to the type used by the vector operand (down-conversion of the scalar type is illegal and will result in a compile time error). 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.

Expressions on the left of an assignment are evaluated before expressions on the right of the assignment. 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

[20] 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 a function designator or an lvalue that is a valid operand of the unary **&** operator, ***&E** is a function designator or an lvalue equal to **E**. 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. 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.

```
    lvalue = lvalue op expression
```

and the l-value and expression must satisfy the semantic requirements of both *op* and equals (=).

Note: Except for the **`sizeof`** operator, the **`half`** data type cannot be used with any of the operators described in this section.

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`. The address space name for arguments to a `__kernel` function declared to be a pointer of a type can be one of the following names only: `__global`, `__local` or `__constant`. A pointer into address space A can only be assigned to a pointer that is in the same address space A. Casting a pointer in address space A to a pointer into address space B is illegal.

Arguments passed to the `__kernel` function declared to be of type `image2d_t` or `image3d_t` can only be allocated from the `__global` space. The `__global` address space is therefore assumed to be the address space name for any arguments of type `image2d_t` or `image3d_t`.

The `__constant` address space name must be used for any global variables declared in the program source.

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 arry 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 memory object is attached to an argument declared with this qualifier, the argument must be declared as type `image2d_t` for a 2D image memory object or as type `image3d_t` for a 3D image memory object. The elements of an image memory object cannot be directly accessed. Built-in functions to read from and write to an image memory 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. This qualifier can be used with arguments to functions (including `__kernel` functions) declared as pointers, or with variables defined inside a `__kernel` function.

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. This qualifier can be used with arguments to functions (including `__kernel` functions) that are declared as pointers, or with local variables inside a function declared as pointers, or with global variables. Global variables declared in the program source with the **`__constant`** qualifer 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.

6.5.4 __private (or private)

Variables inside a function (including `__kernel` functions) that are not pointers are in the **`__private`** or **`private`** address space. Variables declared as pointers that are defined inside a function or passed as arguments to a function are considered to be in the **`__private`** address space if an address space qualifier is not specified.

6.6 Image Access Qualifiers

Image memory 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 memory object. The **`__read_only (or read_only)`** and **`__write_only (or write_only)`** qualifiers must be used with image memory object arguments to declare if the image memory 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 image memory object and `imageB` is a write-only image memory object.

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 qualifer:

- 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.

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>)))`[21] 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`.

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
```

[21] 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.

```
    // 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

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 6.11.8* for a list of image read and write functions. Pointers to `image2d_t` or `image3d_t` are not allowed. `image2d_t` or `image3d_t` data types cannot be declared in a struct. `image2d_t` and `image3d_t` 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 as local variables insides functions 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 supported.

g. The `extern`, `static`, `auto` and `register` storage-class specifiers are not supported.

h. Predefined identifiers such as `__func__` 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 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.

        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.

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 100.

`__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*.

`__ROUNDING_MODE__` is used to determine the current rounding mode and is set to `rte`. The `__ROUNDING_MODE__` only affects the rounding mode of conversions to a float type.

`__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 `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 6.7.1* for the function attribute qualifier 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.

6.11.1 Work-Item Functions

Table 6.6 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.

Table 6.6 *Work-Item Functions Table*

6.11.2 Math Functions

The list of built-in math functions is described in *table 6.7*. 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 6.7 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, 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 *x*, gentype *y*)	Compute **atan2** (*x*, *y*) / 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**[22] (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*, gentype **iptr*)[23]	Returns **fmin**(*x* – **floor** (*x*), 0x1.fffffep-1f). **floor**(x) is returned in *iptr*.
gentype **frexp** (gentype *x*, 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*, 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.

[22] **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.
[23] 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 **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[24].
gentype **modf** (gentype *x*, 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*.
gentype **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*, gentype *y*, 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 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

[24] 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.

	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*, 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*

Table 6.8 describes the following functions:

- A subset of functions from *table 6.7* 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 6.7* 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 6.7. The accuracy (and in some cases the input range(s)) of these functions is implementation-defined.

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 6.8 *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 5.4.3.2*) 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 values given in the following list shall be replaced by implementation-defined constant expressions that are greater or equal in magnitude (absolute value) to those shown, with the same sign. 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 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`}.

6.11.3 Integer Functions

Table 6.9 describes the built-in integer functions that take scalar or vector arguments. We use the generic type name `gentype` to indicate that the function can take `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,` or `ulong{2|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`.

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*)[25]	Returns (*x* + *y* + 1) >> 1. The intermediate sum does not modulo overflow.
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*.
gentype **mad_sat** (gentype *a*, gentype *b*, gentype *c*)	Returns *a* * *b* + *c* and saturates the result.
gentype **max** (gentype *x*, gentype *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*.
gentype **min** (gentype *x*, gentype *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*n* **upsample** (char*n* *hi*, uchar*n* *lo*) ushort*n* **upsample** (uchar*n* *hi*, uchar*n* *lo*)	*result*[i] = ((short)*hi*[i] << 8) | *lo*[i] *result*[i] = ((ushort)*hi*[i] << 8) | *lo*[i]

[25] 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.

int*n* **upsample** (short*n* *hi*, ushort*n* *lo*) uint*n* **upsample** (ushort*n* *hi*, ushort*n* *lo*) long*n* **upsample** (int*n* *hi*, uint*n* *lo*) ulongn*n* **upsample** (uint*n* *hi*, uint*n* *lo*)	*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 6.9 *Scalar and Vector Integer Argument Built-in Functions*

Table 6.10 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, 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 6.10 *Fast Integer Built-in Functions*

The minimum magnitudes shown below shall be replaced by implementation-defined magnitudes with the same sign. 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
```

6.11.4 Common Functions[26]

Table 6.11 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, 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*, genType *edge1*,	Returns 0.0 if *x* <= *edge0* and 1.0 if *x* >= *edge1* and performs smooth Hermite interpolation between 0

[26] The **mix** and **smoothstep** functions can be implemented using contractions such as **mad** or **fma**.

genType *x*)
 	
genType **smoothstep** (float *edge0*, float *edge1*, genType *x*)
	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 6.11 *Scalar and Vector Argument Built-in Common Function Table*

6.11.5 Geometric Functions[27]

Table 6.12 describes the list of built-in geometric functions. These all operate component-wise. The description is per-component. The generic type name `gentype` indicates that the function can take `float, float2,` or `float4` as the type for the arguments.

The built-in geometric functions are implemented using the round to nearest even rounding mode.

Function	**Description**
float4 **cross** (float4 *p0*, float4 *p1*)	Returns the cross product of *p0.xyz* and *p1.xyz*. The *w* component of float4 result returned will be 0.0.
float **dot** (gentype *p0*, gentype *p1*)	Compute dot product.
float **distance** (gentype *p0*, gentype *p1*)	Returns the distance between *p0* and *p1*. This is calculated as **length**(*p0* – *p1*).
float **length** (gentype *p*)	Return the length of vector *p*, 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.
float **fast_distance** (gentype *p0*, gentype *p1*)	Returns **fast_length**(*p0* – *p1*).
float **fast_length** (gentype *p*)	Returns the length of vector *p* computed as: **half_sqrt**($p.x^2 + p.y^2 + \ldots$.)
gentype **fast_normalize** (gentype *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.

[27] 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 6.12 *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 described in *table 6.13* can be used with built-in scalar or vector types as arguments and return a scalar vector integer result. The argment type `gentype` can be `char`, `char`*`n`*, `uchar`, `uchar`*`n`*, `short`, `short`*`n`*, `ushort`, `ushort`*`n`*, `int`, `int`*`n`*, `uint`, `uint`*`n`*, `long`, `long`*`n`*, `ulong`, `ulong`*`n`*, `float` and `float`*`n`*. The argument type `igentype` refers to signed integer vector types i.e. `char`, `char`*`n`*, `short`, `short`*`n`*, `int`, `int`*`n`*, `long` and `long`*`n`*. The argument type `ugentype` refers to unsigned integer vector types i.e. `uchar`, `uchar`*`n`*, `ushort`, `ushort`*`n`*, `uint`, `uint`*`n`*, `ulong` and `ulong`*`n`*.

The functions **isequal**, **isnotequal**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, **isfinite**, **isinf**, **isnan**, **isnormal**, **isordered**, **isunordered** and **signbit** described in *table 6.13* 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*.
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 6.13 *Scalar and Vector Relational Functions*

6.11.7 Vector Data Load and Store Functions

Table 6.14 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, float`. We use the generic type name `gentype`*n* 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}, int{2|4|8|16}, uint{2|4|8|16}, long{2|4|8|16}, ulong{2|4|8|16}` or `float{2|4|8|16}` as the type for the arguments unless otherwise stated. The suffix used in gentype*n* or the function name (i.e. **vload*n***, **vstore*n*** etc.) represents the number of elements in the built-in vector type (*n* = 2, 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 location (*p* + (*offset* * *n*)). The read address computed as (*p* + (*offset* * *n*)) must be 8-bit aligned if gentype is char*n*, uchar*n*; 16-bit aligned if gentype is short*n*, ushort*n*; 32-bit aligned if gentype is int*n*, uint*n*, float*n*; 64-bit aligned if gentype is long*n*, ulong*n*.
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 8-bit aligned if gentype is char*n*, uchar*n*; 16-bit aligned if gentype is short*n*, ushort*n*; 32-bit aligned if gentype is int*n*, uint*n*, float*n*; 64-bit aligned if gentype is long*n*, ulong*n*.
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*)	**vload_half** returns sizeof (float) bytes of data read from location (*p* + *offset*). The half value read is converted to single precision floating-point. The read address computed as (*p* + *offset*) must be 16-bit aligned.

float **vload_half** (size_t *offset*, 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*)	**vload_half*n*** returns sizeof (float*n*) bytes of data read from location (*p* + (*offset* * *n*)). The half*n* values read are converted to single precision floating-point. 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*,	The single-precision floating-point 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 write address computed as (*p* + *offset*) must be 16-bit aligned. **vstore_half** uses the current rounding mode. The default current rounding mode is round to nearest even.

size_t *offset*, __private half **p*)	
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 single-precision floating-point vector value given by *data* is converted to a half vector value using the appropriate rounding mode. The half*n* value is then written to address computed as (*p* + (*offset* * *n*)). The write 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*)	Return sizeof (float*n*) bytes of data read from location (*p* + (*offset* * *n*)). The half*n* values are read from location (*p* + (*offset* * *n*)) and then converted to single precision floating-point. The read address computed as (*p* + (*offset* * *n*)) must be aligned to sizeof (half) * number of half elements being read (given by *n*) bytes.

float*n* **vloada_half*n*** (size_t *offset*, const __private half **p*)	
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 single-precision floating-point vector value is converted to a half vector value using the appropriate rounding mode. The half*n* value is then written to address computed as (*p* + (*offset* * *n*)). The write address computed as (*p* + (*offset* * *n*)) must be aligned to sizeof (half) * number of half elements being read (given by *n*) bytes. **vstorea_half*n*** uses the current rounding mode. The default current rounding mode is round to nearest even.

Table 6.14 *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 6.14*. The pointer argument p can be a pointer to __global, __local or __private memory for store functions described in *table 6.14*. The pointer argument p can be a pointer to __global, __local, __constant or __private memory for load functions described in *table 6.14*.

6.11.8 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.

6.11.8.1 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 a 2D or 3D 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 6.15*.

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. In OpenCL 1.0, the samplers specified with an image in multiple **read_image{f|i|ui}** calls 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_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[28] – out of range image coordinates will return a border color. The border color is (`0.0f, 0.0f, 0.0f, 0.0f`) if image channel order is CL_A, CL_RA, CL_ARGB, CL_BGRA or CL_RGBA and is (`0.0f, 0.0f, 0.0f, 1.0f`) if image channel order is CL_R, CL_RG, or CL_RGB. 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.

[28] This is similar to the GL_ADDRESS_CLAMP_TO_BORDER addressing mode.

`<filter mode>`	Specifies the filtering mode to use. This must be a literal 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 6.15 *Sampler Descriptor*

Samplers cannot be declared as arrays, pointers or as local variables inside functions 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**.

6.11.8.2 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 1D or 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*) unsigned int4 **read_imageui** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) unsigned int4 **read_imageui** (image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate *(x, y)* to do an element lookup in the 1D or 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 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*, unsigned int4 *color*)	Write *color* value to location specified by coordinate *(x, y)* in the 1D or 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*)	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.

unsigned int4 **read_imageui** (image3d_t *image*, sampler_t *sampler*, int4 *coord*) unsigned int4 **read_imageui** (image3d_t *image*, sampler_t *sampler*, float4 *coord*)	**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 2D or 3D image width in pixels.
int **get_image_height** (image2d_t *image*) int **get_image_height** (image3d_t *image*)	Return the 2D or 3D image height in pixels.
int **get_image_depth** (image3d_t *image*)	Return the 3D 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_RG CLK_RA CLK_RGB CLK_RGBA CLK_ARGB CLK_BGRA
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 6.16 *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 6.16* 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 unsigned int4 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 unsigned int4 components of channel data**
`CL_R`	`(r, 0.0, 0.0, 1.0)`
`CL_A`	`(0.0, 0.0, 0.0, a)`
`CL_RG`	`(r, g, 0.0, 1.0)`
`CL_RA`	`(r, 0.0, 0.0, a)`
`CL_RGB`	`(r, g, b, 1.0)`
`CL_RGBA, CL_BGRA, CL_ARGB`	`(r, g, b, a)`

6.11.9 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 6.17 *Built-in Synchronization Functions*

6.11.10 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 6.18 *Built-in Explicit Memory Fence Functions*

6.11.11 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_elements*, event_t *event*) event_t **async_work_group_copy** (__global gentype **dst*, const __local gentype **src*, size_t *num_elements*, event_t *event*)	Perform an async copy of *num_elements* `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; otherwise the results are undefined. Returns an event handle 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 handle returned will be the event handle supplied in *event* argument.
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 events 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; otherwise the results are undefined.
void **prefetch** (const __global gentype **p*, size_t *num_elements*)	Prefetch *num_elements* * 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*

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 not a requirement.

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 6.9.2* (math functions), *6.9.4* (common functions) and *6.9.5* (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 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[29] describes the minimum accuracy of single precision floating-point arithmetic operations given as ULP values.

Function	**Min Accuracy - ULP values**[30]
x* + *y	Correctly rounded
x* – *y	Correctly rounded
x* * *y	Correctly rounded
1.0 / *x*	<= 1 ulp
x* / *y	<= 2 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 – e^x	<= 3 ulp
exp2 – 2^x	<= 3 ulp
exp10 – 10^x	<= 3 ulp
expm1	<= 3 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp

[29] The ULP values for built-in math functions **lgamma** and **lgamma_r** is currently undefined.
[30] 0 ulp is used for math functions that do not require rounding.

fmin	0 ulp
fmod	0 ulp
fract	<= 1 ulp
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)
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 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.

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 (±0, ±0) returns NaN.
powr (+∞, ±0) returns NaN.
powr (+1, ±∞) 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 ±∞, or if *y* is 0 and the other argument is non-NaN or if either argument is a NaN.

rootn (±0, *n*) is ±∞ for odd *n* < 0.
rootn (±0, *n*) is +∞ 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 (±∞) returns a NaN.

tanpi (±0) returns ±0.
tanpi (±∞) 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 *x* is +∞ where *n* + 0.5 is representable.
tanpi (*n* + 0.5) for odd integer *x* is -∞ 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`).[31]

[31] 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, 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.

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.

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

$$\begin{aligned}T = &(1 - a) * (1 - b) * (1 - c) * T_{i0j0k0}\\ &+ a * (1 - b) * (1 - c) * T_{i1j0k0}\\ &+ (1 - a) * b * (1 - c) * T_{i0j1k0}\\ &+ a * b * (1 - c) * T_{i1j1k0}\\ &+ (1 - a) * (1 - b) * c * T_{i0j0k1}\\ &+ a * (1 - b) * c * T_{i1j0k1}\\ &+ (1 - a) * b * c * T_{i0j1k1}\end{aligned}$$

$$+ a * b * c * T_{i1j1k1}$$

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

$$\begin{aligned} T = & (1 - a) * (1 - b) * T_{i0j0} \\ & + a * (1 - b) * T_{i1j0} \\ & + (1 - a) * b * T_{i0j1} \\ & + a * b * T_{i1j1} \end{aligned}$$

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.

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 **read_image{f|i|ui}** 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

$$\begin{aligned}T = &(1 - a) * (1 - b) * (1 - c) * T_{i0j0k0}\\ &+ a * (1 - b) * (1 - c) * T_{i1j0k0}\\ &+ (1 - a) * b * (1 - c) * T_{i0j1k0}\\ &+ a * b * (1 - c) * T_{i1j1k0}\\ &+ (1 - a) * (1 - b) * c * T_{i0j0k1}\end{aligned}$$

```
        + a * (1 – b) * c * Ti1j0k1
        + (1 – a) * b * c * Ti0j1k1
        + a * b * c * Ti1j1k1
```

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) * Ti0j0
        + a * (1 – b) * Ti1j0
        + (1 – a) * b * Ti0j1
        + a * b * Ti1j1
```

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_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_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_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 relative 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)

```
convert_uchar_sat_<impl-rounding-mode>(f * 255.0f)
```

Let $f_{preferred}$ = (f * 255.0f)
Let f_{approx} = impl-rounding-mode(f * 255.0f)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

float → CL_UNORM_INT16 (16-bit unsigned integer)

```
convert_ushort_sat_<impl-rounding-mode>(f * 65535.0f)
```

Let $f_{preferred}$ = (f * 65535.0f)
Let f_{approx} = impl-rounding-mode(f * 65535.0f)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

float → CL_SNORM_INT8 (8-bit signed integer)

```
convert_char_sat_<impl_rounding_mode>(f * 127.0f)
```

Let $f_{preferred}$ = (f * 127.0f)
Let f_{approx} = impl-rounding-mode(f * 127.0f)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

float → CL_SNORM_INT16 (16-bit signed integer)

```
convert_short_sat_<impl-rounding-mode>(f * 32767.0f)
```

Let $f_{preferred}$ = (f * 32767.0f)
Let f_{approx} = 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_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_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_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
require	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 Extension Function Pointers

The macro

clGetExtensionFunctionAddress (*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 **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.

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 1.0 adds support for 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.
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**
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 6.7* (also listed below) are extended to include appropriate versions of functions that take `double,` and `double{2|4|8|16}` as arguments and return values. `gentype` now also includes `double, double2, 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 *x*, gentype *y*)	Compute **atan2** (*x*, *y*) / 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, 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*, gentype **iptr*)	Returns **fmin**(*x* – **floor** (*x*), 0x1.fffffep-1f). **floor**(x) is returned in *iptr*.
gentype **frexp** (gentype *x*, 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*, 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[32].
gentype **modf** (gentype *x*, 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*, 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 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

[32] 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.

	current rounding direction.
gentype **rsqrt** (gentype)	Compute inverse square root.
gentype **sin** (gentype)	Compute sine.
gentype **sincos** (gentype *x*, 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** (πx).
gentype **sqrt** (gentype)	Compute square root.
gentype **tan** (gentype)	Compute tangent.
gentype **tanh** (gentype)	Compute hyperbolic tangent.
gentype **tanpi** (gentype *x*)	Compute **tan** (πx).
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 values given in the following list shall be replaced by implementation-defined constant expressions that are greater or equal in magnitude (absolute value) to those shown, with the same sign. 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 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[33]

The built-in common functions defined in *table 6.11* (also listed below) are extended to include appropriate versions of functions that take `double`, and `double{2|4|8|16}` as arguments and return values. `gentype` now also includes `double, double2, 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. $(180 / \pi)$ * *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*,	Returns the linear blend of *x* & *y* implemented as:

[33] The **mix** and **smoothstep** functions can be implemented using contractions such as **mad** or **fma**.

gentype *y*, gentype *a*) gentype **mix** (gentype *x*, gentype *y*, double *a*)	*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*,
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 6.11 *Scalar and Vector Argument Built-in Common Function Table*

9.3.4 Geometric Functions[34]

The built-in geometric functions defined in *table 6.12* (also listed below) are extended to include appropriate versions of functions that take `double`, and `double{2|4}` as arguments and return values. `gentype` now also includes `double, double2,` and `double4`. These are described below.

Function	**Description**
double4 **cross** (double4 *p0*, double4 *p1*)	Returns the cross product of *p0.xyz* and *p1.xyz*. The *w* compoent of double result will be 0.0.
double **dot** (gentype *p0*, gentype *p1*)	Compute dot product.

[34] The geometric functions can be implemented using contractions such as **mad** or **fma**.

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 6.12 *Scalar and Vector Argument Built-in Geometric Function Table*

9.3.5 Relational Functions

The scalar and vector relational functions described in *table 6.13* are extended to include versions that take `double`, `double2`, `double4`, `double8` and `double16` as arguments.

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 6.13* 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)	Test for a NaN.

long*n* **isnan** (double*n*)	
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*, long*n* *c*) double*n* **select** (double*n* *a*, double*n* *b*, ulong*n* *c*)	For each component, *result[i]* = if MSB of *c[i]* is set ? *b[i]* : *a[i]*.

Table 6.13 *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 6.14* (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`, `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*,	Return sizeof (gentype*n*) bytes of data read from location (*p* + (*offset* * *n*)). The read address computed as (*p* + (*offset* * *n*)) must be 8-bit aligned if gentype is char*n*, uchar*n*; 16-bit aligned if gentype is short*n*, ushort*n*; 32-bit aligned if gentype is int*n*, uint*n*, float*n*; 64-bit aligned if gentype is long*n*, ulong*n* or double*n*.

const __private gentype **p*)	
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 8-bit aligned if gentype is char*n*, uchar*n*; 16-bit aligned if gentype is short*n*, ushort*n*; 32-bit aligned if gentype is int*n*, uint*n*, float*n*; 64-bit aligned if gentype is long*n*, ulong*n* or double*n*.
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*) 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*)	The double-precision floating-point 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 write address computed as (*p* + *offset*) must be16-bit aligned. **vstore_half** uses 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*) 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*)	The double-precision floating-point vector value given by *data* is converted to a half vector value using the appropriate rounding mode. The half*n* value is then written to address computed as (*p* + (*offset* * *n*)). The write 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 **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*)	The double-precision floating-point vector value is converted to a half vector value using the appropriate rounding mode. The half*n* value is then written to address computed as (*p* + (*offset* * *n*)). The write address computed as (*p* + (*offset* * *n*)) must be aligned to sizeof (half) * number of half elements being read (given by *n*) bytes. **vstorea_half*n*** uses the current rounding

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*, size_t *offset*, __private half **p*)	mode. The default current rounding mode is round to nearest even.

Table 6.14 *Vector Data Load and Store Functions*

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_elements*, event_t *event*) event_t **async_work_group_copy** (__global gentype **dst*, const __local gentype **src*,	Perform an async copy of *num_elements* `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; otherwise the results are undefined. Returns an event handle that can be used by

size_t *num_elements*, event_t *event*)	**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 handle returned will be the event handle supplied in *event* argument.
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 events 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; otherwise the results are undefined.
void **prefetch** (const __global gentype **p*, size_t *num_elements*)	Prefetch *num_elements* * 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. 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.

Function	**Min Accuracy - ULP values**[35]
x + y	Correctly rounded
x – y	Correctly rounded
x * y	Correctly rounded

[35] 0 ulp is used for math functions that do not require rounding.

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
copysign	0 ulp
cos	<= 4 ulp
cosh	<= 4 ulp
cospi	<= 4 ulp
erfc	<= 16 ulp
erf	<= 16 ulp
exp – e^x	<= 3 ulp
exp2 – 2^x	<= 3 ulp
exp10 – 10^x	<= 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	<= 1 ulp
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)
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 Selecting Rounding Mode

OpenCL 1.0 adds support for specifying the rounding mode for an instruction or group of instructions in the program source as an optional extension. An application that wants to use this feature will need to include the **`#pragma OPENCL EXTENSION cl_khr_select_fprounding_mode : enable`** directive.

If the **cl_khr_select_fprounding_mode** extension is supported, the OpenCL implementation must support all four rounding modes for single precision floating-point i.e. the CL_DEVICE_SINGLE_FP_CONFIG described in *table 4.3* must include CL_FP_ROUND_TO_ZERO and CL_FP_ROUND_TO_INF. This is already the case for double precision floating-point.

The appropriate rounding mode can be specified using the following pragma in the program source.

```
#pragma OPENCL SELECT_ROUNDING_MODE rounding-mode
```

The *`rounding-mode-value`* can be one of the following values:

- **`rte     - round to nearest even`**
- **`rtz     - round to zero`**
- **`rtp     - round to positive infinity`**
- **`rtn     - round to negative infinity`**

The **`#pragma OPENCL SELECT_ROUNDING_MODE`** sets the rounding mode for all instructions that operate on floating-point types (scalar or vector types) or produce floating-point values that follow this pragma in the program source until the next **`#pragma OPENCL SELECT_ROUNDING_MODE`** is encountered. Note that the rounding mode specified for a block of code is known at compile time. Except where otherwise documented, the callee functions do not inherit the rounding mode of the caller function.

If this extension is enabled, the **`__ROUNDING_MODE__`** preprocessor symbol shall be defined to be one of the following according to the current rounding mode.

```
#define __ROUNDING_MODE__     rte
#define __ROUNDING_MODE__     rtz
#define __ROUNDING_MODE__     rtp
#define __ROUNDING_MODE__     rtz
```

This is intended to let us remap `foo()` to `foo_rte()` in the preprocessor using

```
#define foo          foo ## __ROUNDING_MODE__
```

The default rounding mode is round to nearest even. The built-in math functions described in *section 6.11.2*, the common functions described in *section 6.11.4* and the geometric functions described in *section 6.11.5* are implemented with the round to nearest even rounding mode. Various builtin conversions and the **vstore_half** and **vstorea_half** builtin functions that do not specify a rounding mode inherit the current rounding mode. Conversions from floating-point to integer type always use `rtz` mode, except where the user specifically asks for another rounding mode.

Some examples that describe how the rounding mode selection works are given below:

```
#pragma OPENCL SELECT_ROUNDING_MODE rtz
      float4      a = b * c;  ← uses round to zero rounding mode.

#pragma OPENCL SELECT_ROUNDING_MODE rtp
      float4  d = foo(a);     ← function foo uses rounding mode
                                specified where source for foo() is
                                implemented.
```

9.5 Atomic Functions for 32-bit integers

OpenCL 1.0 adds support for the following optional extensions that implement atomic operations on 32-bit signed and unsigned integers to locations in `__global` memory:

> **cl_khr_global_int32_base_atomics**, and
> **cl_khr_global_int32_extended_atomics**.

An application that wants to use any of these extensions will need to include the **`#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable`** or **`#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics : enable`** directive in the OpenCL program source.

The atomic functions supported by the **cl_khr_global_int32_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_global_int32_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**
int **atom_add** (__global int **p*, int *val*) unsigned int **atom_add** (__global 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 **atom_sub** (__global int **p*, int *val*) unsigned int **atom_sub** (__global 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 **atom_xchg** (__global int **p*, int *val*) unsigned int **atom_xchg** (__global unsigned int **p*, unsigned int *val*)	Swaps the *old* value stored at location *p* with new value given by *val*. Returns *old* value.
int **atom_inc** (__global int **p*) unsigned int **atom_inc** (__global 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 **atom_dec** (__global int **p*) unsigned int **atom_dec** (__global 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 **atom_cmpxchg** (__global int **p*, int *cmp*, int *val*) unsigned int **atom_cmpxchg** (__global 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*.

Table 9.1 *Built-in Atomic Functions implemented by* **cl_khr_global_int32_base_atomics** *extension*

Function	**Description**
int **atom_min** (__global int **p*, int *val*) unsigned int **atom_min** (__global 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 **atom_max** (__global int **p*, int *val*) unsigned int **atom_max** (__global unsigned int **p*, unsigned 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*.
int **atom_and** (__global int **p*, int val) unsigned int **atom_and** (__global 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 **atom_or** (__global int **p*, int *val*) unsigned int **atom_or** (__global 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 **atom_xor** (__global int **p*, int val) unsigned int **atom_xor** (__global 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 9.2 *Built-in Atomic Functions implemented by* **cl_khr_global_int32_extended_atomics** *extension*

9.6 Local Atomics for 32-bit integers

OpenCL 1.0 adds support for the following optional extensions that implement atomic operations on 32-bit signed and unsigned integers to locations in `__local` memory:

> **cl_khr_local_int32_base_atomics**, and
> **cl_khr_local_int32_extended_atomics**.

An application that wants to use any of these extensions will need to include the **`#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable`** or **`#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics : enable`** directive in the OpenCL program source.

The atomic functions supported by the **cl_khr_local_int32_base_atomics** extension are described in *table 9.3*. All of the functions listed in *table 9.3* are performed in one atomic transaction.

The atomic functions supported by the **cl_khr_local_int32_extended_atomics** extension are described in *table 9.4*. All of the functions listed in *table 9.4* are performed in one atomic transaction.

Function	**Description**
int **atom_add** (__local int **p*, int *val*) unsigned int **atom_add** (__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 **atom_sub** (__local int **p*, int *val*) unsigned int **atom_sub** (__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 **atom_xchg** (__local int **p*, int *val*) unsigned int **atom_xchg** (__local unsigned int **p*, unsigned int *val*)	Swaps the *old* value stored at location *p* with new value given by *val*. Returns *old* value.
int **atom_inc** (__local int **p*) unsigned int **atom_inc** (__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 **atom_dec** (__local int **p*) unsigned int **atom_dec** (__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 **atom_cmpxchg** (__local int **p*, int *cmp*, int *val*) unsigned int **atom_cmpxchg** (	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute

__local unsigned int *p*, unsigned int *cmp*. unsigned int *val*)	(*old* == *cmp*) ? *val* : *old* and store result at location pointed by *p*. The function returns *old*.

Table 9.3 *Built-in Local Atomic Functions implemented by* **cl_khr_local_int32_base_atomics** *extension*

Function	**Description**
int **atom_min** (__local int **p*, int *val*) unsigned int **atom_min** (__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 **atom_max** (__local int **p*, int *val*) unsigned int **atom_max** (__local unsigned int **p*, unsigned 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*.
int **atom_and** (__local int **p*, int val) unsigned int **atom_and** (__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 **atom_or** (__local int **p*, int *val*) unsigned int **atom_or** (__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 **atom_xor** (__local int **p*, int val) unsigned int **atom_xor** (__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 9.4 *Built-in Local Atomic Functions implemented by* **cl_khr_local_int32_extended_atomics** *extension*

9.7 64-bit Atomics

OpenCL 1.0 adds support for the following optional extensions that implement atomic operations on 64-bit signed and unsigned integers to locations in `__global` and `__local` memory:

cl_khr_int64_base_atomics and **cl_khr_int64_extended_atomics**.

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 9.5*. All of the functions listed in *table 9.5* are performed in one atomic transaction.

The atomic functions supported by the **cl_khr_int64_extended_atomics** extension are described in *table 9.6*. All of the functions listed in *table 9.6* 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** (__global long **p*, long *val*) long **atom_add** (__local long **p*, long *val*) ulong **atom_add** (__global ulong **p*, ulong *val*) ulong **atom_add** (__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** (__global long **p*, long *val*) long **atom_sub** (__local long **p*, long *val*) ulong **atom_sub** (__global ulong **p*, ulong *val*) ulong **atom_sub** (__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** (__global long **p*, long *val*) long **atom_xchg** (__local long **p*, long *val*) ulong **atom_xchg** (__global ulong **p*, ulong *val*) ulong **atom_xchg** (__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** (__global long **p*) long **atom_inc** (__local long **p*) ulong **atom_inc** (__global ulong **p*) ulong **atom_inc** (__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** (__global long **p*)	Read the 64-bit value (referred to as *old*)

long **atom_dec** (__local long **p*) ulong **atom_dec** (__global ulong **p*) ulong **atom_dec** (__local ulong **p*)	stored at location pointed by *p*. Compute (*old* - *1*) and store result at location pointed by *p*. The function returns *old.*
long **atom_cmpxchg** (__global long **p*, long *cmp*, long *val*) long **atom_cmpxchg** (__local long **p*, long *cmp*, long *val*) ulong **atom_cmpxchg** (__global ulong **p*, ulong *cmp*, ulong *val*) ulong **atom_cmpxchg** (__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 9.5 *Built-in Atomic Functions implemented by* **cl_khr_int64_base_atomics** *extension*

Function	**Description**
long **atom_min** (__global long **p*, long *val*) long **atom_min** (__local long **p*, long *val*) ulong **atom_min** (__global ulong **p*, ulong *val*) ulong **atom_min** (__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** (__global long **p*, long *val*) long **atom_max** (__local long **p*, long *val*) ulong **atom_max** (__global ulong **p*, ulong *val*) ulong **atom_max** (__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** (__global long **p*, long *val*) long **atom_and** (__local long **p*, long *val*) ulong **atom_and** (__global ulong **p*, ulong *val*) ulong **atom_and** (__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** (__global long **p*, long *val*) long **atom_or** (__local long **p*, long *val*) ulong **atom_or** (__global ulong **p*, ulong *val*) ulong **atom_or** (__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** (__global long **p*, long *val*) long **atom_xor** (__local long **p*, long *val*) ulong **atom_xor** (__global ulong **p*, ulong *val*) ulong **atom_xor** (__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 9.6 *Built-in Atomic Functions implemented by* **cl_khr_int64_extended_atomics** *extension*

9.8 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*, unsigned int4 *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.

For read-write or write-only 3D images, the minimum list of supported image formats is given by the table below:

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

9.9 Byte Addressable Stores

Section 6.8.m describes restrictions on built-in types `char`, `uchar`, `char2`, `uchar2`, `short`, and `half`. The OpenCL extension **`cl_khr_byte_addressable_store`** removes these restrictions. An application that wants to be able to write to elements of a pointer (or struct) that are of type `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort` and `half` will need to include the **`#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable`** directive before any code that performs writes that may not be supported as per *section 6.8.m*.

In the example given in section 6.8.m and copied below the lines in **bold** will compile and work correctly if the **`cl_khr_byte_addressable_store`** extension is supported.

```
__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;

    px[1] = pA[1];  ← no longer an error.

    pB[id] = b;     ← no longer an error.
}
```

9.10 Half Floating-Point

OpenCL 1.0 supports half as a storage format only in the core specification. 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.
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 4	**cl_half4**
half 8	**cl_half8**
half16	**cl_half16**

9.10.1 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.

9.10.2 Math Functions

The built-in math functions defined in *table 6.7* (also listed below) are extended to include appropriate versions of functions that take `half,` and `half{2|4|8|16}` as arguments and return values. `gentype` now also includes `half, half2, 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 *x*, gentype *y*)	Compute **atan2** (*x*, *y*) / 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*, gentype **iptr*)	Returns **fmin**(*x* – **floor** (*x*), 0x1.fffffep-1f). **floor**(x) is returned in *iptr*.
gentype **frexp** (gentype *x*, 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*, 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 **lgamma** (gentype *x*)	Log gamma function.
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[36].
gentype **modf** (gentype *x*, 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

[36] 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.

	the argument. It stores the integral part in the object pointed to by *iptr*.
half*n* **nan** (ushort*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*, 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 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*, 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*

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

9.10.3 Common Functions[37]

The built-in common functions defined in *table 6.11* (also listed below) are extended to include appropriate versions of functions that take `half`, and `half{2|4|8|16}` as arguments and return values. `gentype` now also includes `half, half2, half4, half8` and `half16`. These are described below.

Function	**Description**
gentype **clamp** (gentype *x*, gentype *minval*, gentype *maxval*) gentype **clamp** (gentype *x*, half *minval*, half *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*, 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.

[37] The **mix** and **smoothstep** functions can be implemented using contractions such as **mad** or **fma**.

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 6.11 *Scalar and Vector Argument Built-in Common Function Table*

9.10.4 Geometric Functions[38]

The built-in geometric functions defined in *table 6.12* (also listed below) are extended to include appropriate versions of functions that take `half`, and `half{2|4}` as arguments and return values. `gentype` now also includes `half, half2,` and `half4`. These are described below.

Function	Description
half4 **cross** (half4 *p0*, half4 *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., $\sqrt{p.x^2 + p.y^2 + \ldots}$
gentype **normalize** (gentype *p*)	Returns a vector in the same direction as *p* but with a length of 1.

Table 6.12 *Scalar and Vector Argument Built-in Geometric Function Table*

[38] The geometric functions can be implemented using contractions such as **mad** or **fma**.

9.10.5 Relational Functions

The scalar and vector relational functions described in *table 6.13* are extended to include versions that take `half`, `half2`, `half4`, `half8` and `half16` as arguments.

The relational and equality operators (<, <=, >, >=, !=, ==) can be used with `half`*`n`* vector types and shall produce a vector `short`*`n`* result as described in *section 6.3*.

The functions described in *table 6.13* are extended to include the `half`*`n`* 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*, half *y*) short*n* **isgreaterequal** (half*n x*, half*n y*)	Returns the component-wise compare of *x* >= *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 6.13 *Vector Relational Functions*

9.10.6 Image Read and Write Functions

The image read and write functions defined in *table 6.16* are extended to support image coordinates and 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 1D or 2D memory 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 or 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 1D or 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_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. Appropriate data format conversion will be done to convert channel data from a half floating-point value to actual data format in which the channels are stored. 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 … image width – 1, 0 … image height – 1) respectively, is undefined.
half4 **read_imageh** (image3d_t *image*, sampler_t *sampler*, int4 *coord*) half4 **read_imageh** (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_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 or 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 is undefined.

Table 6.16 *Built-in Image Read and Write Functions*

If the **`cl_khr_3d_image_writes`** extension is supported, then the following function will also be supported by the **`cl_khr_fp16`** extension.

void **write_imageh** (image3d_t *image*, int4 *coord*, half4 *color*)	Write *color* value to location specified by coordinate *(x, y, z)* in the 3D image memory 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_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. Appropriate data format conversion will be done to convert channel data from a half floating-point value to actual data format in which the channels are stored. The behavior of **write_imageh** 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.

9.10.7 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 `gentype`*`n`* is extended to include `half2`, `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 location (*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 6.14 *Vector Data Load and Store Functions*

9.10.8 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 `half`, `half2`, `half4`, `half8` and `half16`.

Function	Description
event_t **async_work_group_copy** (__local gentype **dst*, const __global gentype **src*, size_t *num_elements*, event_t *event*) event_t **async_work_group_copy** (__global gentype **dst*, const __local gentype **src*, size_t *num_elements*, event_t *event*)	Perform an async copy of *num_elements* `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; otherwise the results are undefined. Returns an event handle 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 handle returned will be the event handle supplied in *event* argument.
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 events 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; otherwise the results are undefined.
void **prefetch** (const __global gentype **p*, size_t *num_elements*)	Prefetch *num_elements* * 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*

9.10.9 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. The required minimum half precision floating-point capability as implemented by this extension is CL_FP_ROUND_TO_ZERO | CL_FP_ROUND_TO_INF | CL_FP_INF_NAN.

9.10.10 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 operations on half types are not required to be correctly rounded using a round to nearest even rounding mode. These operations can be correctly rounded using a round-to-zero rounding mode instead. Conversion between integer and a half floating-point format must be correctly rounded using a round to nearest even rounding mode.

The following table describes the minimum accuracy of half precision floating-point arithmetic operations given as ULP values.

Function	Min Accuracy - ULP values[39]
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
erf	<= 4 ulp
exp – e^x	<= 2 ulp
exp2 – 2^x	<= 2 ulp
exp10 – 10^x	<= 2 ulp
expm1	<= 2 ulp

[39] 0 ulp is used for math functions that do not require rounding.

fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	<= 1 ulp
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)
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.*

10. OpenCL Embedded Profile

The OpenCL 1.0 specification describes the feature requirements for desktop platforms. This section describes the OpenCL 1.0 embedded profile that allows us to target a subset of the OpenCL 1.0 specification for handheld and embedded platforms. The optional extensions defined in *section 8* apply to both profiles.

The OpenCL 1.0 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 not supported.

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. 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**[40] 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_REPEAT and CLK_ADDRESS_CLAMP_TO_EDGE.

5. Addition, subtraction, multiplication, fused multiply-add operations are not required to be correctly rounded using round to nearest even rounding mode as required by IEEE 754. The OpenCL 1.0 embedded profile relaxes this requirement and allow these basic operations for single precision floating-point types to be implemented as correctly rounded with round to zero rounding mode that results in an error of <= 1 ulp. 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.

---

[40] If **`cl_khr_fp16`** extension is supported.

6. 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*.

7. 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
   ```

Table 10.1 describes the minimum accuracy of single precision floating-point arithmetic operations given as ULP values for the embedded profile.

Function	**Min Accuracy - ULP values**[41]
x* + *y	Correctly rounded
x* – *y	Correctly rounded
x* * *y	Correctly rounded
1.0 / *x*	<= 1 ulp
x* / *y	<= 3 ulp

[41] 0 ulp is used for math functions that do not require rounding.

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 – e^x	<= 4 ulp
exp2 – 2^x	<= 4 ulp
exp10 – 10^x	<= 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	<= 1 ulp
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)
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_MAX_COMPUTE_UNITS	unsigned int	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_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_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 6.11.8* 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_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.

		The mandated minimum floating-point capability is: CL_FP_ROUND_TO_ZERO.
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` qualifer in a kernel. The minimum value is 4.
CL_DEVICE_LOCAL_MEM_SIZE	unsigned long long	Size of local memory arena in bytes. The minimum value is 1 KB.
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 read-only 2D and optional 3D images, the minimum list of supported image formats 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

For read-write or write-only 2D images, the minimum list of supported image formats 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

The OpenCL implementation is *thread-safe* for API calls that create, retain and release objects such as a context, command-queue, program, kernel and memory objects. OpenCL API calls that queue commands to a command-queue or change the state of OpenCL objects such as command-queue objects, memory objects, program and kernel objects are not *thread-safe*.

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.

Appendix B

B.1 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/render-buffer objects as OpenCL memory objects. This allows efficient sharing of data between these OpenCL and OpenGL. The OpenCL API can be used to execute kernels that read and/or write memory objects that are also an OpenGL buffer object or a texture.

An OpenCL image object can be created from an OpenGL texture or renderbuffer object. An OpenCL buffer object can be created from an OpenGL buffer object.

An OpenCL memory object can be created from an OpenGL texture/buffer/render-buffer object or the default system provided framebuffer if and only if the OpenCL context has been created from a GL context. OpenGL contexts are created using platform specific APIs (EGL, CGL, WGL, GLX are some of the platform specific APIs that allow applications to create GL contexts). The appropriate platform API (such as EGL, CGL, WGL, GLX) will be extended to allow a CL context to be created from a GL context. Creating an OpenCL memory object from the default system provided framebuffer will also require an appropriate extension to the platform API. Refer to the appropriate platform API documentation to understand how to create a CL context from a GL context and creating a CL memory object from the default system provided framebuffer.

B.1.1 CL Buffer Objects → GL Buffer Objects

The function

```
cl_mem      clCreateFromGLBuffer (cl_context context,
                                  cl_mem_flags flags,
                                  GLuint bufobj,
                                  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 a GL buffer object name. The GL buffer object must have a data store created though it does not need to 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. 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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context or was not created from a GL context.
- *errcode_ret* returns CL_INVALID_VALUE if values specified in *flags* are not valid.
- *errcode_ret* returns CL_INVALID_GL_OBJECT if *bufobj* is not a GL buffer object or is a GL buffer object but does not have a data store created.
- *errcode_ret* returns 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**. Changes to the size of the GL buffer object data store using appropriate GL API calls (i.e. **glBufferData**) while there is a CL buffer object created from this GL buffer object will result in undefined behavior when this buffer object is used in OpenCL.

The **clRetainMemObject** and **clReleaseMemObject** functions can be used to retain and release the buffer object.

B.1.2 CL Image Objects → GL Textures

The function

cl_mem **clCreateFromGLTexture2D** (cl_context *context*,
cl_mem_flags *flags*,
GLenum *target*,
GLint *miplevel*,
GLuint *texture*,
int **errcode_ret*)

creates an OpenCL 2D image object from an OpenGL 2D 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.

target must be 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_ARB[42].

miplevel is the mipmap level to be used.

texture is a GL 2D texture, cubemap or texture rectangle object name. The texture object must be a complete texture as per OpenGL rules on texture completeness. The *texture* format and dimensions specified using appropriate **glTexImage2D** call for *miplevel* will be used to create the 2D image object. Only GL texture formats that map to appropriate image channel order and data type specified in *tables 5.4* and *5.5* can be used to create the 2D image object.

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

clCreateFromGLTexture2D returns a valid non-zero OpenCL image object and *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context or was not created from a GL context.

- *errcode_ret* returns CL_INVALID_VALUE if values specified in *flags* are not valid.

- *errcode_ret* returns CL_INVALID_MIPLEVEL if *miplevel* is not a valid mip-level for *texture*.

- *errcode_ret* returns CL_INVALID_GL_OBJECT if *texture* is not an appropriate GL 2D texture, cubemap or texture rectangle.

- *errcode_ret* returns CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if the OpenGL texture format does not map to an appropriate OpenCL image format.

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

[42] Can be used only if the GL_ARB_texture_rectangle OpenGL extension is implemented.

The function

```
cl_mem      clCreateFromGLTexture3D[43] (cl_context context,
                                         cl_mem_flags flags,
                                         GLenum target,
                                         GLint miplevel,
                                         GLuint texture,
                                         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.

target must be GL_TEXTURE_3D.

miplevel is the mipmap level to be used.

texture is a GL 3D texture object name. The texture object must be a complete texture as per OpenGL rules on texture completeness. The *texture* format and dimensions specified using appropriate **glTexImage3D** call for *miplevel* will be used to create the 3D image jobject. Only GL texture formats that map to appropriate image channel order and data type specified in *tables 5.4* and *5.5* can be used to create the 3D image object.

errcode_ret will return an appropriate error code. 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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context or was not created from a GL context.
- *errcode_ret* returns CL_INVALID_VALUE if values specified in *flags* are not valid.
- *errcode_ret* returns CL_INVALID_MIPLEVEL if *miplevel* is not a valid mip-level for *texture*.
- *errcode_ret* returns CL_INVALID_GL_OBJECT if *texture* is not a GL 3D texture object.

[43] OpenCL 1.0 supports read-only 3D image memory objects. Writes to 3D image objects that are GL 3D textures are supported if the OpenCL implementation supports the **`cl_khr_3d_image_writes`** extension.

- *errcode_ret* returns CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if the OpenGL texture format does not map to an appropriate OpenCL image format.

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

Changes to the dimensions or format/type used to represent texels of the GL texture using appropriate GL API calls (i.e. **glTexImage2D**, **glTexImage3D**) while there is a CL image object created from this GL texture object will result in undefined behavior when this image object is used in OpenCL.

The **clRetainMemObject** and **clReleaseMemObject** functions can be used to retain and release the image memory objects.

B.1.3 CL Image Objects → GL Renderbuffers

The function

```
cl_mem      clCreateFromGLRenderbuffer (cl_context context,
                                        cl_mem_flags flags,
                                        GLuint renderbuffer,
                                        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 a GL renderbuffer object name. The renderbuffer storage must be specified before the image object can be created. Only GL renderbuffer formats that map to appropriate image channel order and data type specified in *tables 5.4* and *5.5* can be used to create the 2D image object.

errcode_ret will return an appropriate error code. 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. It returns a NULL value with one of the following error values returned in *errcode_ret*:

- *errcode_ret* returns CL_INVALID_CONTEXT if *context* is not a valid context or was not created from a GL context.

- *errcode_ret* returns CL_INVALID_VALUE if values specified in *flags* are not valid.
- *errcode_ret* returns CL_INVALID_GL_OBJECT if *renderbuffer* is not a GL renderbuffer object.
- *errcode_ret* returns CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if the OpenGL renderbuffer format does not map to an appropriate OpenCL image format.
- *errcode_ret* returns CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Changes to the dimensions or format used to represent pixels of the GL renderbuffer using appropriate GL API calls (i.e. **glRenderbufferStorage**) while there is a CL image object created from this GL renderbuffer object will result in undefined behavior when this image object is used in OpenCL.

The **clRetainMemObject** and **clReleaseMemObject** functions can be used to retain and release the image memory objects.

B.1.4 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_TEXTURE_RECTANGLE, 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. It returns CL_INVALID_MEM_OBJECT if *memobj* is not a valid OpenCL memory object, returns CL_INVALID_GL_OBJECT if there is no GL object associated with *memobj*.

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 the table below.

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 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 *target* argument specified in **clCreateGLTexture2D**, or **clCreateGLTexture3D**.
CL_GL_MIPMAP_LEVEL	GLint	The *miplevel* argument specified in **clCreateGLTexture2D**, or **clCreateGLTexture3D**.

clGetGLTextureInfo returns CL_SUCCESS if the function is executed successfully. It returns CL_INVALID_MEM_OBJECT if *memobj* is not a valid OpenCL memory object, returns CL_INVALID_GL_OBJECT if there is no GL texture object (2D or 3D texture) associated with *memobj*, and returns 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 above and *param_value* is not NULL, or if *param_value* and *param_value_size_ret* are NULL.

B.1.5 Sharing memory objects that map to GL objects between GL and CL contexts

Creating CL memory objects from GL objects using **clCreateFromGLBuffer**, **clCreateFromGLTexture2D**, **clCreateFromGLTexture3D,** or **clCreateFromGLRenderbuffer** ensures that the underlying storage of that GL object will not be deleted while the corresponding CL memory object still exists.

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.

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 refer to a GL object (buffer/texture/renderbuffer objects or the framebuffer).

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 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.

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_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 refer to a GL object (buffer/texture/renderbuffer objects or the framebuffer).

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 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.

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

If the CL and GL context are in the same host thread, then **clEnqueueAcquireGLObjects** and **clEnqueueReleaseGLObjects** allows you to do finer grained synchronization of OpenGL and OpenCL commands. The **clEnqueueAcquireGLObjects** will ensure that all OpenGL commands that use the GL objects specified in *mem_objects* argument to **clEnqueueAcquireGLObjects** are completed before any OpenCL commands that use these GL objects are executed. The GL context associated with the CL context specified in **clEnqueueAcquireGLObjects** must be currently bound; otherwise the behavior of **clEnqueueAcquireGLObjects** is considered undefined. Similarly, the **clEnqueueReleaseGLObjects** ensures that all OpenCL commands that use the GL objects specified in *mem_objects* argument to **clEnqueueReleaseGLObjects** are completed before any OpenGL commands that use these GL objects are executed. If the CL and GL context are being used by separate host threads, the **glFlush** command must be called by the GL context before a **clEnqueueAcquireGLObjects** is called by the CL context. This would be no different from synchronizing GL objects across multiple GL contexts. Note that the application still has to implement appropriate synchronization to make sure CL and GL calls are made in the correct order across threads.

The results of reading or writing to an OpenCL memory object created from an OpenGL object that has not been acquired using **clEnqueueAcquireGLObjects** or has been released using **clEnqueueReleaseGLObjects** are undefined. The results of modifying an OpenCL memory object created from an OpenGL object in a GL context while it is being used in a CL context or vice-versa are undefined.

Appendix C — 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[44] 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

[44] 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 `.0123456789abcdef` 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 `uchart16` 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 D - Examples

The examples described in appendix D are not considered as part of the official OpenCL specification. They are included to illustrate how to use the OpenCL APIs to execute a kernel on a device, and common algorithms such as matrix tranpose and reduction written as OpenCL kernels. These examples have been written with the goal of illustrating how to use OpenCL and should not be considered as examples of how to write performant OpenCL kernels.

D.1 A Simple OpenCL Kernel

This example demonstrates how a kernel can be written that operates on individual data elements of a memory object which is very similar to what is allowed in GLSL / HLSL. In this example we show a function that computes a dot product of two float4 arrays and writes result to a float array.

The OpenCL kernel source is:

```
__kernel void
dot_product (__global const float4 *a,
             __global const float4 *b, __global float *c)
{
    int gid = get_global_id(0);

    c[gid] = dot(a[gid], b[gid]);
}
```

The following code describes the OpenCL runtime calls the application must make to create appropriate memory objects, create program object and load the program source for the kernel described above, build the program executable, create the kernel object, load the appropriate argument values and execute the dot product kernel on the GPU device.

```
void
delete_memobjs(cl_mem *memobjs, int n)
{
    int i;
    for (i=0; i<n; i++)
        clReleaseMemObject(memobjs[i]);
}

int
exec_dot_product_kernel(const char *program_source,
                        int n, void *srcA, void *srcB, void *dst)
{
    cl_context         context;
```

```
cl_command_queue  cmd_queue;
cl_device_id      *devices;
cl_program        program;
cl_kernel         kernel;
cl_mem            memobjs[3];
size_t            global_work_size[1];
size_t            local_work_size[1];
size_t            cb;
cl_int            err;

// create the OpenCL context on a GPU device
context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                                  NULL, NULL, NULL);
if (context == (cl_context)0)
    return -1;

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);
if (cmd_queue == (cl_command_queue)0)
{
    clReleaseContext(context);
    return -1;
}
free(devices);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
                          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                          sizeof(cl_float4) * n, srcA, NULL);
if (memobjs[0] == (cl_mem)0)
{
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

memobjs[1] = clCreateBuffer(context,
                          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                          sizeof(cl_float4) * n, srcB, NULL);
if (memobjs[1] == (cl_mem)0)
{
    delete_memobjs(memobjs, 1);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;

}
```

```
memobjs[2] = clCreateBuffer(context,
                            CL_MEM_READ_WRITE,
                            sizeof(cl_float) * n, NULL, NULL);
if (memobjs[2] == (cl_mem)0)
{
    delete_memobjs(memobjs, 2);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// create the program
program = clCreateProgramWithSource(context,
                   1, (const char**)&program_source, NULL, NULL);
if (program == (cl_program)0)
{
    delete_memobjs(memobjs, 3);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
    delete_memobjs(memobjs, 3);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// create the kernel
kernel = clCreateKernel(program, "dot_product", NULL);
if (kernel == (cl_kernel)0)
{
    delete_memobjs(memobjs, 3);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// set the args values
err = clSetKernelArg(kernel,  0,
                  sizeof(cl_mem), (void *) &memobjs[0]);
err |= clSetKernelArg(kernel, 1,
                  sizeof(cl_mem), (void *) &memobjs[1]);
err |= clSetKernelArg(kernel, 2,
                  sizeof(cl_mem), (void *) &memobjs[2]);
    if (err != CL_SUCCESS)
    {
        delete_memobjs(memobjs, 3);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(cmd_queue);
        clReleaseContext(context);
        return -1;
    }

    // set work-item dimensions
    global_work_size[0] = n;
    local_work_size[0]= 1;

    // execute kernel
    err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
                                 global_work_size, local_work_size,
                                 0, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        delete_memobjs(memobjs, 3);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(cmd_queue);
        clReleaseContext(context);
        return -1;
    }

    // read output image
    err = clEnqueueReadBuffer(cmd_queue, memobjs[2], CL_TRUE,
                              0, n * sizeof(cl_float), dst,
                              0, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        delete_memobjs(memobjs, 3);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(cmd_queue);
        clReleaseContext(context);
        return -1;
    }

    // release kernel, program, and memory objects
    delete_memobjs(memobjs, 3);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return 0; // success...
}
```

D.2 Matrix Transpose

In this section we describe how to efficiently perform a transpose of a matrix composed of M x N power-of-two elements on OpenCL devices (typically GPUs) that support dedicated local memory.

In this example 64 work-items are issued per work-group which individually operate small 32x2 sections to fill a 32x32 sub-matrix (over 8 iterations). The final 32 x 32 sub-matrix is transposed in local memory with one column padding to avoid bank conflicts. Performing the transpose in local memory allows the reads and writes to global memory to be coalesced. The extra column padding is used to offset the write addresses, so that they don't conflict with the read requests.

Using a padding of 32 (or any odd multiple of GROUP_DIMX = 32) ensures that the reads and writes for each element in global memory on some OpenCL devices will be offset and not operate on the same memory bank/channel. This is important for the global memory write operations, since the column major indices are non-sequential and can cause global memory bank conflicts.

Global memory read requests will operate on sequential indices for the row-major elements, and will not conflict.

The OpenCL kernel code to transpose the matrix is given below.

```
#define PADDING         (32)
#define GROUP_DIMX      (32)
#define LOG_GROUP_DIMX  (5)
#define GROUP_DIMY      (2)
#define WIDTH           (256)
#define HEIGHT          (4096)

__kernel void matrix_transpose(
    __global float *output,
    __global float *input,
    __local float *tile)
{

     int block_x = get_group_id(0);
     int block_y = get_group_id(1);

     int local_x = get_local_id(0) & (GROUP_DIMX - 1);
     int local_y = get_local_id(0) >> LOG_GROUP_DIMX;

     int local_input  = mad24(local_y, GROUP_DIMX + 1, local_x);
     int local_output = mad24(local_x, GROUP_DIMX + 1, local_y);

     int in_x = mad24(block_x, GROUP_DIMX, local_x);
     int in_y = mad24(block_y, GROUP_DIMX, local_y);
     int input_index = mad24(in_y, WIDTH, in_x);

     int out_x = mad24(block_y, GROUP_DIMX, local_x);
     int out_y = mad24(block_x, GROUP_DIMX, local_y);

     int output_index = mad24(out_y, HEIGHT + PADDING, out_x);

     int global_input_stride  = WIDTH * GROUP_DIMY;
     int global_output_stride = (HEIGHT + PADDING) * GROUP_DIMY;

     int local_input_stride  = GROUP_DIMY * (GROUP_DIMX + 1);
     int local_output_stride = GROUP_DIMY;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;

     tile[local_input] = input[input_index];
     local_input += local_input_stride;
     input_index += global_input_stride;
```

```
tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];

barrier(CLK_LOCAL_MEM_FENCE);

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;
```

```
        output[output_index] = tile[local_output];
        local_output += local_output_stride;
        output_index += global_output_stride;

        output[output_index] = tile[local_output];
        local_output += local_output_stride;
        output_index += global_output_stride;

        output[output_index] = tile[local_output];
        local_output += local_output_stride;
        output_index += global_output_stride;

        output[output_index] = tile[local_output];
        local_output += local_output_stride;
        output_index += global_output_stride;

        output[output_index] = tile[local_output];
        local_output += local_output_stride;
        output_index += global_output_stride;

        output[output_index] = tile[local_output];
        local_output += local_output_stride;
        output_index += global_output_stride;

        output[output_index] = tile[local_output];
        local_output += local_output_stride;
        output_index += global_output_stride;

        output[output_index] = tile[local_output];
        local_output += local_output_stride;
        output_index += global_output_stride;

        output[output_index] = tile[local_output];

}
```

D.3 A Simple Reduction Example

Here is a simple reduction example for summing up the elements of a float array.

The OpenCL kernel code is given below.

```
#ifndef GROUP_SIZE
#define GROUP_SIZE (64)
#endif

__kernel void
reduce(__global float *output, __global const float *input,
                     __local float *shared, unsigned int n)
{
    const unsigned int lid = get_local_id(0));
    const unsigned int lsize = GROUP_SIZE;//get_local_size(0);
    // NOTE:  get_local_size(0) must equal GROUP_SIZE

    const unsigned int gid = get_group_id(0);
    const unsigned int gsize = get_num_groups(0);

    const unsigned int gs2 = GROUP_SIZE * 2;
    const size_t stride = gs2 * gsize;

    shared[lid] = 0.0f;

    size_t i = gid * gs2 + lid;
    while (i < n)
    {
        shared[lid] += input[i] + input[(i+GROUP_SIZE)];
        i += stride;
    }
    barrier(CLK_LOCAL_MEM_FENCE);

#if (GROUP_SIZE >= 512)
    if (lid < 256)
        shared[lid] += shared[lid + 256];
    barrrier(CLK_LOCAL_MEM_FENCE);
#endif

#if (GROUP_SIZE >= 256)
    if (lid < 128)
        shared[lid] += shared[lid + 128];
    barrier(CLK_LOCAL_MEM_FENCE);
#endif

#if (GROUP_SIZE >= 128)
    if (lid <  64)
        shared[lid] += shared[lid +  64];
    barrier(CLK_LOCAL_MEM_FENCE);
#endif

    if (lid < 32)
    {
#if (GROUP_SIZE >=  64)
        shared[lid] += shared[lid + 32];
        barrier(CLK_LOCAL_MEM_FENCE);
#endif

#if (GROUP_SIZE >=  32)
        shared[lid] += shared[lid + 16];
        barrier(CLK_LOCAL_MEM_FENCE);
#endif
#if (GROUP_SIZE >=  16)
        shared[lid] += shared[lid +  8];
        barrier(CLK_LOCAL_MEM_FENCE);
#endif
#if (GROUP_SIZE >=   8)
        shared[lid] += shared[lid +  4];
        barrier(CLK_LOCAL_MEM_FENCE);
#endif
#if (GROUP_SIZE >=   4)
        shared[lid] += shared[lid +  2];
        barrier(CLK_LOCAL_MEM_FENCE);
#endif
#if (GROUP_SIZE >=   2)
        shared[lid] += shared[lid +  1];
        barrier(CLK_LOCAL_MEM_FENCE);
#endif
    }

    if (lid == 0)
        output[gid] = shared[0];
}
```

Note that the above reduction kernel example requires `n` to be a mutiple of `GROUP_SIZE`.

The following code describes the OpenCL runtime calls the application must make to create appropriate memory objects, create program object and load the program source for the kernel described above in this section, build the program executable, create the kernel object, load the appropriate argument values and execute the reduction kernel on the GPU device.

```
#include <libc.h>
#include <stdbool.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <OpenCL/opencl.h>
#include <math.h>
```

```
//////////////////////////////////////////////////////////////////////

#define MAX_GROUPS      (64)
#define MAX_WORK_ITEMS  (64)

static int count      = 1024 * 1024;

//////////////////////////////////////////////////////////////////////

static char *
load_program_source(const char *filename)
{
    struct stat statbuf;
    FILE        *fh;
    char        *source;

    fh = fopen(filename, "r");
    if (fh == 0)
        return 0;

    stat(filename, &statbuf);
    source = (char *) malloc(statbuf.st_size + 1);
    fread(source, statbuf.st_size, 1, fh);
    source[statbuf.st_size] = '\0';

    return source;
}

//////////////////////////////////////////////////////////////////////

float reduce_float(float *data, int size)
{
    int i;
    float sum = data[0];
    float c = (float)0.0f;
    for (i = 1; i < size; i++)
    {
        float y = data[i] - c;
        float t = sum + y;
        c = (t - sum) - y;
        sum = t;
    }
    return sum;
}

//////////////////////////////////////////////////////////////////////

void create_reduction_pass_counts(
    int count, int max_groups, int max_work_items,
    int *level_count, size_t **group_counts,
    size_t **work_item_counts, int **entry_counts)
```

```
{
    int work_items = (count < max_work_items * 2) ? count / 2
                                                   : max_work_items;
    if (count < 1)
        work_items = 1;

    int groups = count / (work_items * 2);
    groups = max_groups < groups ? max_groups : groups;

    int max_levels = 1;
    int s = groups;

    while(s > 1)
    {
        int work_items = (s < max_work_items * 2) ? s / 2
                                                  : max_work_items;
        s = s / (work_items*2);
        max_levels++;
    }

    *group_counts = (size_t*)malloc(max_levels * sizeof(size_t));
    *work_item_counts = (size_t*)malloc(max_levels * sizeof(size_t));
    *entry_counts = (int*)malloc(max_levels * sizeof(int));

    (*level_count) = max_levels;
    (*group_counts)[0] = groups;
    (*work_item_counts)[0] = work_items;
    (*entry_counts)[0] = count;

    s = groups;
    int level = 1;

    while(s > 1)
    {
        int work_items = (s < max_work_items * 2) ? s / 2
                                                  : max_work_items;
        int groups = s / (work_items * 2);
        groups = (max_groups < groups) ? max_groups : groups;

        (*group_counts)[level] = groups;
        (*work_item_counts)[level] = work_items;
        (*entry_counts)[level] = s;

        s = s / (work_items*2);
        level++;
    }
}

////////////////////////////////////////////////////////////////////////
```

```
int main(int argc, char **argv)
{
    cl_int           err;
    cl_device_id     device_id;
    cl_command_queue commands;
    cl_context       context;
    cl_mem           output;
    cl_mem           input;
    cl_mem           partials;
    int              level_count = 0;
    size_t*          group_counts = 0;
    size_t*          work_item_counts = 0;
    int*             entry_counts = 0;
    int i;

    // Create some random input data on the host
    //
    float *float_data = (float*)malloc(count * sizeof(float));
    for (i = 0; i < count; i++)
    {
        float_data[i] = ((float) rand() / (float) RAND_MAX);
    }

    // Connect to a GPU compute device
    //
    err = clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
    }

    // Create a compute context
    //
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    if (!context)
    {
        printf("Error: Failed to create a compute context!\n");
        return EXIT_FAILURE;
    }

    // Create a command commands
    //
    commands = clCreateCommandQueue(context, device_id, 0, &err);
    if (!commands)
    {
        printf("Error: Failed to create a command commands!\n");
        return EXIT_FAILURE;
    }

    // Load the compute program from disk into a cstring buffer
    //
    const char* filename = "reduce_kernel.cl";
    printf("Loading program '%s'...\n", filename);
    char *source = load_program_source(filename);
    if(!source)
    {
        printf("Error: Failed to load compute program from file!\n");
        return EXIT_FAILURE;
    }

    // Create the input buffer on the device
    //
    input = clCreateBuffer(context, CL_MEM_READ_WRITE,
                           sizeof(float) * count, NULL, NULL);
    if (!input)
    {
        printf("Error: Failed to allocate input data
                                buffer on device!\n");
        return EXIT_FAILURE;
    }

    // Fill the input buffer with the host allocated random data
    //
   err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0,
                  sizeof(float) * count, (void *)float_data,
                  0, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to write to input data
                                buffer on device!\n");
        return EXIT_FAILURE;
    }

    // Create an intermediate data buffer for intra-level results
    //
    partials = clCreateBuffer(context, CL_MEM_READ_WRITE,
                           sizeof(float) * count, NULL, NULL);
    if (!partials)
    {
        printf("Error: Failed to allocate partial sum
                                buffer on device!\n");
        return EXIT_FAILURE;
    }

    // Create the output buffer on the device
    //
    output = clCreateBuffer(context, CL_MEM_READ_WRITE,
                           sizeof(float) * count, NULL, NULL);
    if (!output)
    {
        printf("Error: Failed to allocate result
                                buffer on device!\n");
        return EXIT_FAILURE;
```

```
}

// Determine the global and local dimensions for the execution
//
create_reduction_pass_counts(count, MAX_GROUPS, MAX_WORK_ITEMS,
                             &level_count, &group_counts,
                             &work_item_counts, &entry_counts);

// Create programs and kernels for each level of the reduction
//
cl_program *programs =
  (cl_program*)malloc(level_count * sizeof(cl_program));

memset(programs, 0, level_count * sizeof(cl_program));

cl_kernel *kernels = (cl_kernel*)malloc(level_count *
                                         sizeof(cl_kernel));
memset(kernels, 0, level_count * sizeof(cl_kernel));

for(i = 0; i < level_count; i++)
{
    char *block_source = malloc(strlen(source) + 1024);
    size_t length = strlen(source) + 1024;
    memset(block_source, 0, length);

    // Insert macro definitions to specialize the kernel to a
    // particular group size
    //
    const char define[] = "#define GROUP_SIZE";
    sprintf(block_source, "%s (%d) \n%s\n", define,
                  (int)group_counts[i], source);


    // Create the compute program from the source buffer
    //
    programs[i] = clCreateProgramWithSource(context, 1,
                        (const char **) & block_source,
                        NULL, &err);
    if (!programs[i] || err != CL_SUCCESS)
    {
        printf("Error: Failed to create compute program!\n");
        return EXIT_FAILURE;
    }

    // Build the program executable
    //
    err = clBuildProgram(programs[i], 0, NULL, NULL, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];
```

```
            printf("Error: Failed to build program executable!\n");
            clGetProgramBuildInfo(programs[i], device_id,
                CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
            printf("%s\n", buffer);
            return EXIT_FAILURE;
        }

        // Create the compute kernel from within the program
        //
        kernels[i] = clCreateKernel(programs[i], "reduce", &err);
        if (!kernels[i] || err != CL_SUCCESS)
        {
            printf("Error: Failed to create compute kernel!\n");
            return EXIT_FAILURE;
        }

        free(block_source);
    }

    // Do the reduction for each level
    //
    printf("Performing Reduction [%d]...\n", count);

    cl_mem pass_swap = output;
    cl_mem pass_input = output;
    cl_mem pass_output = input;

    for(i = 0; i < level_count; i++)
    {
        size_t global = group_counts[i] * work_item_counts[i];
        size_t local = work_item_counts[i];
        unsigned int entries = entry_counts[i];

        printf("Pass[%4d] Global[%4d] Local[%4d]
               Groups[%4d] WorkItems[%4d] Entries[%d]\n",  i,
                (int)global, (int)local, (int)group_counts[i],
                (int)work_item_counts[i], entries);

        // Swap the inputs and outputs for each pass
        //
        pass_swap = pass_input;
        pass_input = pass_output;
        pass_output = pass_swap;

        err = CL_SUCCESS;
        err |= clSetKernelArg(kernels[i],  0,
                                  sizeof(cl_mem), &pass_output);
        err |= clSetKernelArg(kernels[i],  1,
                                  sizeof(cl_mem), &pass_input);
        err |= clSetKernelArg(kernels[i],  2,
                             sizeof(float) * work_item_counts[i], NULL);
        err |= clSetKernelArg(kernels[i],  3,
                               sizeof(int),    &entries);
        if (err != CL_SUCCESS)
        {
            printf("Error: Failed to set kernel arguments!\n");
            return EXIT_FAILURE;
        }

        // After the first pass, use the partial sums for the
       // next input values
        //
        if(pass_input == input)
            pass_input = partials;

        err = CL_SUCCESS;
        err |= clEnqueueNDRangeKernel(commands, kernels[i], 1, NULL,
                                &global, &local, 0, NULL, NULL);
        if (err != CL_SUCCESS)
        {
            printf("Error: Failed to execute kernel!\n");
            return EXIT_FAILURE;
        }

    }

    // Read back the final sum that was computed on the device
    //
    float computed_result = 0.0f;
    err = clEnqueueReadBuffer(commands, pass_output, CL_TRUE, 0,
                      sizeof(float), &computed_result, 0, NULL, NULL);
    if (err)
    {
        printf("Error: Failed to read back results
                                from the device!\n");
        return EXIT_FAILURE;
    }

    // Do our own reduction to compare the results
    //
    float reference = reduce_float(float_data, count);

    // Verify the results are correct
    //
    float error = fabs(reference - computed_result);

    // Report any incorrect results
    //
    if (error > 1e-5)
    {
        printf("Reference %f != Device Result %f\n",
                           reference, computed_result);
        printf("Error:  Incorrect results obtained!
                     Max error = %f\n", error);
        return EXIT_FAILURE;
    }
    else
    {
        printf("Results Validated!\n");
    }

    // Shutdown and cleanup
    //
    for(i = 0; i < level_count; i++)
    {
        clReleaseKernel(kernels[i]);
        clReleaseProgram(programs[i]);
    }

    clReleaseMemObject(input);
    clReleaseMemObject(output);
    clReleaseMemObject(partials);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);

    free(group_counts);
    free(work_item_counts);
    free(entry_counts);
    free(kernels);

    return 0;
}
```