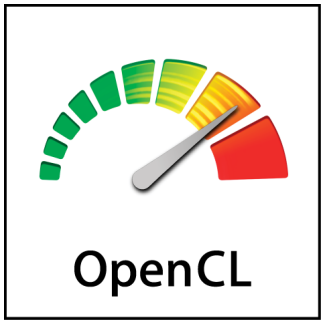

The OpenCL Specification

Version: 2.2

Khronos OpenCL Working Group

Revision: v2.2-3

Editor: Alex Bourd

May 12, 2017

KHRONOS™ GROUP

Contents

List of Tables

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:

Chuck Rose, Adobe
Eric Berdahl, Adobe
Shivani Gupta, Adobe
Bill Licea Kane, AMD
Ed Buckingham, AMD
Jan Civlin, AMD
Laurent Morichetti, AMD
Mark Fowler, AMD
Marty Johnson, AMD
Michael Mantor, AMD
Norm Rubin, AMD
Ofer Rosenberg, AMD
Brian Sumner, AMD
Victor Odintsov, AMD
Aaftab Munshi, Apple
Abe Stephens, Apple
Alexandre Namaan, Apple
Anna Tikhonova, Apple
Chendi Zhang, Apple
Eric Bainville, Apple
David Hayward, Apple
Giridhar Murthy, Apple
Ian Ollmann, Apple
Inam Rahman, Apple
James Shearer, Apple
MonPing Wang, Apple
Tanya Lattner, Apple
Mikael Bourges-Sevenier, Aptina
Anton Lokhmotov, ARM
Dave Shreiner, ARM
Hedley Francis, ARM
Robert Elliott, ARM
Scott Moyers, ARM
Tom Olson, ARM
Anastasia Stulova, ARM
Christopher Thompson-Walsh, Broadcom
Holger Waechtler, Broadcom
Norman Rink, Broadcom
Andrew Richards, Codeplay
Maria Rovatsou, Codeplay
Alistair Donaldson, Codeplay
Alastair Murray, Codeplay
Stephen Frye, Electronic Arts
Eric Schenk, Electronic Arts
Daniel Laroche, Freescale
David Neto, Google
Robin Grosman, Huawei
Craig Davies, Huawei
Brian Horton, IBM
Brian Watt, IBM

Gordon Fossum, IBM
Greg Bellows, IBM
Joaquin Madruga, IBM
Mark Nutter, IBM
Mike Perks, IBM
Sean Wagner, IBM
Jon Parr, Imagination Technologies
Robert Quill, Imagination Technologies
James McCarthy, Imagination Technologie
Aaron Kunze, Intel
Aaron Lefohn, Intel
Adam Lake, Intel
Alexey Bader, Intel
Allen Hux, Intel
Andrew Brownsword, Intel
Andrew Lauritzen, Intel
Bartosz Sochacki, Intel
Ben Ashbaugh, Intel
Brian Lewis, Intel
Geoff Berry, Intel
Hong Jiang, Intel
Jayanth Rao, Intel
Josh Fryman, Intel
Larry Seiler, Intel
Mike MacPherson, Intel
Murali Sundaresan, Intel
Paul Lalonde, Intel
Raun Krisch, Intel
Stephen Junkins, Intel
Tim Foley, Intel
Timothy Mattson, Intel
Yariv Aridor, Intel
Michael Kinsner, Intel
Kevin Stevens, Intel
Jon Leech, Khronos
Benjamin Bergen, Los Alamos National Laboratory
Roy Ju, Mediatek
Bor-Sung Liang, Mediatek
Rahul Agarwal, Mediatek
Michal Witaszek, Mobica
JenqKuen Lee, NTHU
Amit Rao, NVIDIA
Ashish Srivastava, NVIDIA
Bastiaan Aarts, NVIDIA
Chris Cameron, NVIDIA
Christopher Lamb, NVIDIA
Dibyapran Sanyal, NVIDIA
Guatam Chakrabarti, NVIDIA
Ian Buck, NVIDIA
Jaydeep Marathe, NVIDIA
Jian-Zhong Wang, NVIDIA
Karthik Raghavan Ravi, NVIDIA
Kedar Patil, NVIDIA
Manjunath Kudlur, NVIDIA
Mark Harris, NVIDIA

Michael Gold, NVIDIA
Neil Trevett, NVIDIA
Richard Johnson, NVIDIA
Sean Lee, NVIDIA
Tushar Kashalikar, NVIDIA
Vinod Grover, NVIDIA
Xiangyun Kong, NVIDIA
Yogesh Kini, NVIDIA
Yuan Lin, NVIDIA
Mayuresh Pise, NVIDIA
Allan Tzeng, QUALCOMM
Alex Bourd, QUALCOMM
Anirudh Acharya, QUALCOMM
Andrew Gruber, QUALCOMM
Andrzej Mamona, QUALCOMM
Benedict Gaster, QUALCOMM
Bill Torzewski, QUALCOMM
Bob Rychlik, QUALCOMM
Chihong Zhang, QUALCOMM
Chris Mei, QUALCOMM
Colin Sharp, QUALCOMM
David Garcia, QUALCOMM
David Ligon, QUALCOMM
Jay Yun, QUALCOMM
Lee Howes, QUALCOMM
Richard Ruigrok, QUALCOMM
Robert J. Simpson, QUALCOMM
Sumesh Udayakumaran, QUALCOMM
Vineet Goel, QUALCOMM
Lihan Bin, QUALCOMM
Vlad Shimanskiy, QUALCOMM
Jian Liu, QUALCOMM
Tasneem Brutch, Samsung
Yoonseo Choi, Samsung
Dennis Adams, Sony
Pr-Anders Aronsson, Sony
Jim Rasmusson, Sony
Thierry Lepley, STMicroelectronics
Anton Gorenko, StreamComputing
Jakub Szuppe, StreamComputing
Vincent Hindriksen, StreamComputing
Alan Ward, Texas Instruments
Yuan Zhao, Texas Instruments
Pete Curry, Texas Instruments
Simon McIntosh-Smith, University of Bristol
James Price, University of Bristol
Paul Preney, University of Windsor
Shane Peelar, University of Windsor
Brian Hutsell, Vivante
Mike Cai, Vivante
Sumeet Kumar, Vivante
Wei-Lun Kao, Vivante
Xing Wang, Vivante
Jeff Fifield, Xilinx
Hem C. Neema, Xilinx

Henry Styles, Xilinx
Ralph Wittig, Xilinx
Ronan Keryell, Xilinx
AJ Guillon, YetiWare Inc

Chapter 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 todays 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 intermediate language with a well-specified computation environment. The OpenCL standard:

- Supports both data- and task-based parallel programming models
- Utilizes a portable and self-contained intermediate representation with support for parallel execution
- 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. 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.

Chapter 2

Glossary

Application: The combination of the program running on the host and OpenCL devices.

Acquire semantics: One of the memory order semantics defined for synchronization operations. Acquire semantics apply to atomic operations that load from memory. Given two units of execution, **A** and **B**, acting on a shared atomic object **M**, if **A** uses an atomic load of **M** with acquire semantics to synchronize-with an atomic store to **M** by **B** that used release semantics, then **A**'s atomic load will occur before any subsequent operations by **A**. Note that the memory orders *release*, *sequentially consistent*, and *acquire_release* all include *release semantics* and effectively pair with a load using acquire semantics.

Acquire release semantics: A memory order semantics for synchronization operations (such as atomic operations) that has the properties of both acquire and release memory orders. It is used with read-modify-write operations.

Atomic operations: Operations that at any point, and from any perspective, have either occurred completely, or not at all. Memory orders associated with atomic operations may constrain the visibility of loads and stores with respect to the atomic operations (see *relaxed semantics*, *acquire semantics*, *release semantics* or *acquire release semantics*).

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 three types of *barriers* a command-queue barrier, a work-group barrier and a sub-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 kernel execution model provides built-in *work-group barrier* functionality. 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*.
- The OpenCL kernel execution model provides built-in *sub-group barrier* functionality. This *barrier* built-in function can be used by a *kernel* executing on a *device* to perform synchronization between *work-items* in a *sub-group* executing the *kernel*. All the *work-items* of a *sub-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.

Built-in Kernel: A *built-in kernel* is a *kernel* that is executed on an OpenCL *device* or *custom device* by fixed-function hardware or in firmware. *Applications* can query the *built-in kernels* supported by a *device* or *custom device*. A *program object* can only contain *kernels* written in OpenCL C or *built-in kernels* but not both. See also *Kernel* and *Program*.

Child kernel: see *device-side enqueue.*

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*.

Command synchronization: Constraints on the order that commands are launched for execution on a device defined in terms of the synchronization points that occur between commands in host command-queues and between commands in device-side command-queues. See *synchronization points*.

Complete: The final state in the six state model for the execution of a command. The transition into this state occurs is signaled through event objects or callback functions associated with a command.

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

Compute Unit: An OpenCL *device* has one or more *compute units*. A *work-group* executes on a single *compute unit*. A *compute unit* is composed of one or more *processing elements* and *local memory*. A *compute unit* may also include dedicated texture filter 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*.

Control flow: The flow of instructions executed by a work-item. Multiple logically related work items may or may not execute the same control flow. The control flow is said to be *converged* if all the work-items in the set execution the same stream of instructions. In a *diverged* control flow, the work-items in the set execute different instructions. At a later point, if a diverged control flow becomes converged, it is said to be a re-converged control flow.

Converged control flow: see **control flow**.

Custom Device: An OpenCL *device* that fully implements the OpenCL Runtime but does not support *programs* written in OpenCL C. A custom device may be specialized non-programmable hardware that is very power efficient and performant for directed tasks or hardware with limited programmable capabilities such as specialized DSPs. Custom devices are not OpenCL conformant. Custom devices may support an online compiler. Programs for custom devices can be created using

the OpenCL runtime APIs that allow OpenCL programs to be created from source (if an online compiler is supported) and/or binary, or from *built-in kernels_supported by the _device*. See also *Device*.

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.

Data race: The execution of a program contains a data race if it contains two actions in different work items or host threads where (1) one action modifies a memory location and the other action reads or modifies the same memory location, and (2) at least one of these actions is not atomic, or the corresponding memory scopes are not inclusive, and (3) the actions are global actions unordered by the global-happens-before relation or are local actions unordered by the local-happens before relation.

Deprecation: existing features are marked as deprecated if their usage is not recommended as that feature is being de-emphasized, superseded and may be removed from a future version of the specification.[BA2]

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.

Device-side enqueue: A mechanism whereby a kernel-instance is enqueued by a kernel-instance running on a device without direct involvement by the host program. This produces *nested parallelism*; i.e. additional levels of concurrency are nested inside a running kernel-instance. The kernel-instance executing on a device (the *parent kernel*) enqueues a kernel-instance (the *child kernel*) to a device-side command queue. Child and parent kernels execute asynchronously though a parent kernel does not complete until all of its child-kernels have completed.

Diverged control flow: see *control flow*.

Ended: The fifth state in the six state model for the execution of a command. The transition into this state occurs when execution of a command has ended. When a Kernel-enqueue command ends, all of the work-groups associated with that command have finished their execution.

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.

Fence: A memory ordering operation without an associated atomic object. A fence can use the *acquire semantics, release semantics*, or *acquire release semantics*.

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.

Generic address space: An address space that include the *private*, *local*, and *global* address spaces available to a device. The generic address space supports conversion of pointers to and from private, local and global address spaces, and hence lets a programmer write a single function that at compile time can take arguments from any of the three named address spaces.

Global Happens before: see *happens before*.

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. *Global memory* is included within the *generic address space* that includes the private and local address spaces.

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

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

Happens before: An ordering relationship between operations that execute on multiple units of execution. If an operation A happens-before operation B then A must occur before B; in particular, any value written by A will be visible to B.We define two separate happens before relations: *global-happens-before* and *local-happens-before*. These are defined in section 3.3.6.

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

Host-thread: the unit of execution that executes the statements in the Host program.

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.

Independent Forward Progress: If an entity supports independent forward progress, then if it is otherwise not dependent on any actions due to be performed by any other entity (for example it does not wait on a lock held by, and thus that must be released by, any other entity), then its execution cannot be blocked by the execution of any other entity in the system (it will not be starved). Work items in a subgroup, for example, typically do not support independent forward progress, so one work item in a subgroup may be completely blocked (starved) if a different work item in the same subgroup enters a spin loop.

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.

Intermediate Language: A lower-level language that may be used to create programs. SPIR-V is a required IL for OpenCL 2.2 runtimes. Additional ILs may be accepted on an implementation-defined basis.

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

Kernel-instance: The work carried out by an OpenCL program occurs through the execution of kernel-instances on devices. The kernel instance is the *kernel object*, the values associated with the arguments to the kernel, and the parameters that define the *NDRange* index space.

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.

Kernel Language: A language that is used to create source code for kernel. Supported kernel languages include OpenCL C, OpenCL C++, and OpenCL dialect of SPIR-V.

Launch: The transition of a command from the *submitted* state to the *ready* state. See *Ready*.

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*. *Local memory* is included within the *generic address space* that includes the private and global address spaces.

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 Consistency Model: Rules that define which values are observed when multiple units of execution load data from any shared memory plus the synchronization operations that constrain the order of memory operations and define synchronization relationships. The memory consistency model in OpenCL is based on the memory model from the ISO C11 programming language.

Memory Objects: A *memory object* is a handle to a reference counted region of *global memory*. Also see_Buffer Object_and_Image Object_.

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

Memory Scopes: These memory scopes define a hierarchy of visibilities when analyzing the ordering constraints of memory operations. They are defined by the values of the memory_scope enumeration constant. Current values are **memory_scope_work_item**(memory constraints only apply to a single work-item and in practice apply only to image operations)**, memory_scope_sub_group** (memory-ordering constraints only apply to work-items executing in a sub-group), **memory_scope_work_group** (memory-ordering constraints only apply to work-items executing in a work-group), **memory_scope_device** (memory-ordering constraints only apply to work-items executing on a single device) and **memory_scope_all_svm_devices** (memory-ordering constraints only apply to work-items executing across multiple devices and when using shared virtual memory).

Modification Order:All modifications to a particular atomic object M occur in some particular **total order**, called the **modification order** of M. If A and B are modifications of an atomic object M, and A happens-before B, then A shall precede B in the modification order of M. Note that the modification order of an atomic object M is independent of whether M is in local or global memory.

Nested Parallelism: See *device-side enqueue*.

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*.

Parent device: The OpenCL *device* which is partitioned to create *sub-devices*. Not all *parent devices_are _root devices*. A *root device* might be partitioned and the *sub-devices* partitioned again. In this case, the first set of *sub-devices* would be *parent devices* of the second set, but not the *root devices*. Also see *device*, *parent device* and *root device*.

Parent kernel: see *device-side enqueue*.

Pipe: The *pipe* memory object conceptually is an ordered sequence of data items. A pipe has two endpoints: a write endpoint into which data items are inserted, and a read endpoint from which data items are removed. At any one time, only one kernel instance may write into a pipe, and only one kernel instance may read from a pipe. To support the producer consumer design pattern, one kernel instance connects to the write endpoint (the producer) while another kernel instance connects to the reading endpoint (the consumer).

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

Private Memory: A region of memory private to a *work-item*. Variables defined in one *work-items 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.

Queued: The first state in the six state model for the execution of a command. The transition into this state occurs when the command is enqueued into a command-queue.

Ready: The third state in the six state model for the execution of a command. The transition into this state occurs when pre-requisites constraining execution of a command have been met; i.e. the command has been launched. When a Kernel-enqueue command is launched, work-groups associated with the command are placed in a devices work-pool from which they are scheduled for execution.

Re-converged Control Flow: see *control flow*.

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*. Implementations may also modify the *reference count*, e.g. to track attached objects or to ensure correct operation of in-progress or scheduled activities. The object becomes inaccessible to host code when the number of *release* operations performed matches the number of *retain* operations plus the allocation of the object. At this point the reference count may be zero but this is not guaranteed.

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.

Relaxed Semantics: A memory order semantics for atomic operations that implies no order constraints. The operation is *atomic* but it has no impact on the order of memory operations.

Release Semantics: One of the memory order semantics defined for synchronization operations. Release semantics apply to atomic operations that store to memory. Given two units of execution, **A** and **B**, acting on a shared atomic object **M**, if **A** uses an atomic store of **M** with release semantics to synchronize-with an atomic load to **M** by **B*that used acquire semantics, then *A*s atomic store will occur *after* any prior operations by *A**. Note that the memory orders *acquire*, *sequentialy consistent*, and *acquire_release* all include *acquire semantics* and effectively pair with a store using release semantics.

Remainder work-groups: When the work-groups associated with a kernel-instance are defined, the sizes of a work-group in each dimension may not evenly divide the size of the NDRange in the corresponding dimensions. The result is a collection of work-groups on the boundaries of the NDRange that are smaller than the base work-group size. These are known as *remainder work-groups*.

Running: The fourth state in the six state model for the execution of a command. The transition into this state occurs when the execution of the command starts. When a Kernel-enqueue command starts, one or more work-groups associated with the command start to execute.

Root device: A *root device* is an OpenCL *device* that has not been partitioned. Also see *device*, *parent device* and *root device*.

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 doesnt 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 filter mode, and whether the input image coordinate is a normalized or unnormalized value.

Scope inclusion: Two actions **A** and **B** are defined to have an inclusive scope if they have the same scope **P** such that: (1) if **P** is memory_scope_sub_group, and **A** and **B** are executed by work-items within the same sub-group, or (2) if **P** is memory_scope_work_group, and **A** and **B** are executed by work-items within the same work-group, or (3) if **P** is memory_scope_device, and **A** and **B** are executed by work-items on the same device, or (4) if **P** is memory_scope_all_svm_devices, if **A** and **B** are executed by host threads or by work-items on one or more devices that can share SVM memory with each other and the host process.

Sequenced before: A relation between evaluations executed by a single unit of execution. Sequenced-before is an asymmetric, transitive, pair-wise relation that induces a partial order between evaluations. Given any two evaluations A and B, if A is sequenced-before B, then the execution of A shall precede the execution of B.

Sequential consistency: Sequential consistency interleaves the steps executed by each unit of execution. Each access to a memory location sees the last assignment to that location in that interleaving.

Sequentially consistent semantics: One of the memory order semantics defined for synchronization operations. When using sequentially-consistent synchronization operations, the loads and stores within one unit of execution appear to execute in program order (i.e., the sequenced-before order), and loads and stores from different units of execution appear

to be simply interleaved.

Shared Virtual Memory (SVM): An address space exposed to both the host and the devices within a context. SVM causes addresses to be meaningful between the host and all of the devices within a context and therefore supports the use of pointer based data structures in OpenCL kernels. It logically extends a portion of the global memory into the host address space therefore giving work-items access to the host address space. There are three types of SVM in OpenCL **Coarse-Grained buffer SVM**: Sharing occurs at the granularity of regions of OpenCL buffer memory objects. **Fine-Grained buffer SVM**: Sharing occurs at the granularity of individual loads/stores into bytes within OpenCL buffer memory objects. **Fine-Grained system SVM**: Sharing occurs at the granularity of individual loads/stores into bytes occurring anywhere within the host memory.

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.

Specialization constants: Specialization is intended for constant objects that will not have known constant values until after initial generation of a SPIR-V module. Such objects are called specialization constants. Application might provide values for the specialization constants that will be used when SPIR-V program is built. Specialization constants that do not receive a value from an application shall use default value as defined in SPIR-V specification.

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*.

Sub-device: An OpenCL *device* can be partitioned into multiple *sub-devices*. The new *sub-devices* alias specific collections of compute units within the parent *device*, according to a partition scheme. The *sub-devices* may be used in any situation that their parent *device* may be used. Partitioning a *device* does not destroy the parent *device*, which may continue to be used along side and intermingled with its child *sub-devices*. Also see *device*, *parent device* and *root device*.

Sub-group: Sub-groups are an implementation-dependent grouping of work-items within a work-group. The size and number of sub-groups is implementation-defined.

Sub-group Barrier. See *Barrier*.

Submitted: The second state in the six state model for the execution of a command. The transition into this state occurs when the command is flushed from the command-queue and submitted for execution on the device. Once submitted, a programmer can assume a command will execute once its prerequisites have been met.

SVM Buffer: A memory allocation enabled to work with Shared Virtual Memory (SVM). Depending on how the SVM buffer is created, it can be a coarse-grained or fine-grained SVM buffer. Optionally it may be wrapped by a Buffer Object. See *Shared Virtual Memory (SVM)*.

Synchronization: Synchronization refers to mechanisms that constrain the order of execution and the visibility of memory operations between two or more units of execution.

Synchronization operations: Operations that define memory order constraints in a program. They play a special role in controlling how memory operations in one unit of execution (such as work-items or, when using SVM a host thread) are made visible to another. Synchronization operations in OpenCL include *atomic operations* and *fences*.

Synchronization point: A synchronization point between a pair of commands (A and B) assures that results of command A happens-before command B is launched (i.e. enters the ready state) .

Synchronizes with: A relation between operations in two different units of execution that defines a memory order constraint in global memory (*global-synchronizes-with*) or local memory (*local-synchronizes-with*).

Task Parallel Programming Model: A programming model in which computations are expressed in terms of multiple concurrent tasks executing in one or more *command-queues*. The concurrent tasks can be running different *kernels*.

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

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

Unit of execution: a generic term for a process, OS managed thread running on the host (a host-thread), kernel-instance, host program, work-item or any other executable agent that advances the work associated with a program.

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-instance* and share *local memory* and *work-group functions*.

Work-group Barrier. See *Barrier*.

Work-group Function: A function that carries out collective operations across all the work-items in a work-group. Available collective operations are a barrier, reduction, broadcast, prefix sum, and evaluation of a predicate. A work-group function must occur within a *converged control flow*; i.e. all work-items in the work-group must encounter precisely the same work-group function.

Work-group Synchronization: Constraints on the order of execution for work-items in a single work-group.

Work-pool: A logical pool associated with a device that holds commands and work-groups from kernel-instances that are ready to execute. OpenCL does not constrain the order that commands and work-groups are scheduled for execution from the work-pool; i.e. a programmer must assume that they could be interleaved. There is one work-pool per device used by all command-queues associated with that device. The work-pool may be implemented in any manner as long as it assures that work-groups placed in the pool will eventually execute.

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 work-items by its *global ID* or the combination of its *work-group* ID and its *local ID* within a *work-group*.

Chapter 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 is implemented as both host code and device kernel code. The host code portion of an OpenCL application runs on a host processor according to the models native to the host platform. The OpenCL application host code submits the kernel code as commands from the host to OpenCL devices. An OpenCL device executes the commands computation on the processing elements within the device.

An OpenCL device has considerable latitude on how computations are mapped onto the devices processing elements. When processing elements within a compute unit execute the same sequence of statements across the processing elements, the control flow is said to be *converged.* Hardware optimized for executing a single stream of instructions over multiple processing elements is well suited to converged control flows. When the control flow varies from one processing element to another, it is said to be *diverged.* While a kernel always begins execution with a converged control flow, due to branching statements within a kernel, converged and diverged control flows may occur within a single kernel. This provides a great deal of flexibility in the algorithms that can be implemented with OpenCL.

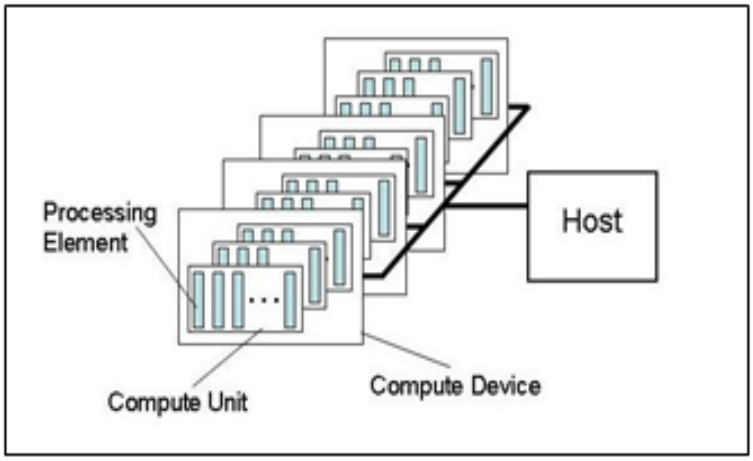

Figure 3.1: *Platform model . . . one host plus one or more compute devices each with one or more compute units composed of one or more processing elements.*

Programmers provide programs in the form of SPIR-V source binaries, OpenCL C or OpenCL C++ source strings or implementation-defined binary objects. The OpenCL platform provides a compiler to translate program input of either form into executable program objects. The device code compiler may be *online* or *offline*. An *online compiler* is available during host program execution using standard APIs. An *offline compiler* is invoked outside of host program control, using platform-specific methods. The OpenCL runtime allows developers to get a previously compiled device program executable and be able to load and execute a previously compiled device program executable.

OpenCL defines two kinds of platform profiles: a *full profile* and a reduced-functionality *embedded profile*. A full profile platform must provide an online compiler for all its devices. An embedded platform may provide an online compiler, but is not required to do so.

A device may expose special purpose functionality as a *built-in function*. The platform provides APIs for enumerating and invoking the built-in functions offered by a device, but otherwise does not define their construction or semantics. A *custom device* supports only built-in functions, and cannot be programmed via a kernel language.

All device types support the OpenCL execution model, the OpenCL memory model, and the APIs used in OpenCL to manage devices.

The platform model is an abstraction describing how OpenCL views the hardware. The relationship between the elements of the platform model and the hardware in a system may be a fixed property of a device or it may be a dynamic feature of a program dependent on how a compiler optimizes code to best utilize physical hardware.

3.2 Execution Model

The OpenCL execution model is defined in terms of two distinct units of execution: **kernels** that execute on one or more OpenCL devices and a **host program** that executes on the host. With regard to OpenCL, the kernels are where the "work" associated with a computation occurs. This work occurs through **work-items** that execute in groups (**work-groups**).

A kernel executes within a well-defined context managed by the host. The context defines the environment within which kernels execute. It includes the following resources:

- **Devices**: One or more devices exposed by the OpenCL platform.

- **Kernel Objects**:The OpenCL functions with their associated argument values that run on OpenCL devices.
- **Program Objects**:The program source and executable that implement the kernels.
- **Memory Objects**:Variables visible to the host and the OpenCL devices. Instances of kernels operate on these objects as they execute.

The host program uses the OpenCL API to create and manage the context. Functions from the OpenCL API enable the host to interact with a device through a *command-queue*. Each command-queue is associated with a single device. The commands placed into the command-queue fall into one of three types:

- **Kernel-enqueue commands**: Enqueue a kernel for execution on a device.
- **Memory commands**: Transfer data between the host and device memory, between memory objects, or map and unmap memory objects from the host address space.
- **Synchronization commands**: Explicit synchronization points that define order constraints between commands.

In addition to commands submitted from the host command-queue, a kernel running on a device can enqueue commands to a device-side command queue. This results in *child kernels* enqueued by a kernel executing on a device (the *parent kernel*). Regardless of whether the command-queue resides on the host or a device, each command passes through six states.

1. **Queued**: The command is enqueued to a command-queue. A command may reside in the queue until it is flushed either explicitly (a call to clFlush) or implicitly by some other command.
2. **Submitted**: The command is flushed from the command-queue and submitted for execution on the device. Once flushed from the command-queue, a command will execute after any prerequisites for execution are met.
3. **Ready**: All prerequisites constraining execution of a command have been met. The command, or for a kernel-enqueue command the collection of work groups associated with a command, is placed in a device work-pool from which it is scheduled for execution.
4. **Running**: Execution of the command starts. For the case of a kernel-enqueue command, one or more work-groups associated with the command start to execute.
5. **Ended**: Execution of a command ends. When a Kernel-enqueue command ends, all of the work-groups associated with that command have finished their execution. *Immediate side effects*, i.e. those associated with the kernel but not necessarily with its child kernels, are visible to other units of execution. These side effects include updates to values in global memory.
6. **Complete**: The command and its child commands have finished execution and the status of the event object, if any, associated with the command is set to CL_COMPLETE.

The execution states and the transitions between them are summarized in Figure 3-2. These states and the concept of a device work-pool are conceptual elements of the execution model. An implementation of OpenCL has considerable freedom in how these are exposed to a program. Five of the transitions, however, are directly observable through a profiling interface. These profiled states are shown in Figure 3-2.

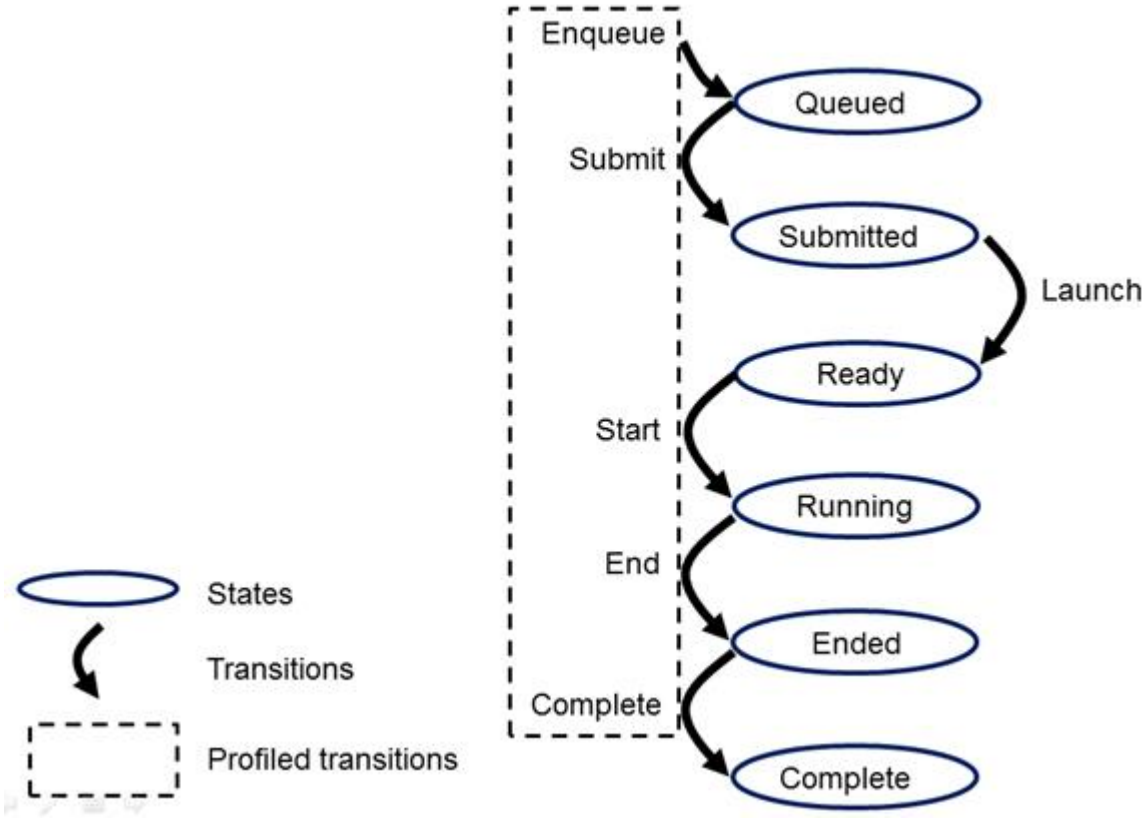

Figure 3-2: The states and transitions between states defined in the OpenCL execution model. A subset of these transitions is exposed through the profiling interface (see section 5.14).

Commands communicate their status through *Event objects*. Successful completion is indicated by setting the event status associated with a command to CL_COMPLETE. Unsuccessful completion results in abnormal termination of the command which is indicated by setting the event status to a negative value. In this case, the command-queue associated with the abnormally terminated command and all other command-queues in the same context may no longer be available and their behavior is implementation defined.

A command submitted to a device will not launch until prerequisites that constrain the order of commands have been resolved. These prerequisites have three sources:

- They may arise from commands submitted to a command-queue that constrain the order in which commands are launched. For example, commands that follow a command queue barrier will not launch until all commands prior to the barrier are complete.
- The second source of prerequisites is dependencies between commands expressed through events. A command may include an optional list of events. The command will wait and not launch until all the events in the list are in the state CL COMPLETE. By this mechanism, event objects define order constraints between commands and coordinate execution between the host and one or more devices.
- The third source of prerequisities can be the presence of non-trivial C initializers or C `constructors for program scope global variables.In this case, OpenCL C/C` compiler shall generate program initialization kernels that perform C initialization or C `construction.These kernels must be executed by OpenCL runtime on a device before any kernel from the same program can be executed on the same device.The ND-range for any program initialization kernel is (1,1,1).When multiple programs are linked together, the order of execution of program initialization kernels that belong to different programs is undefined.`

  `Program clean up may result in the execution of one or more program clean up`

kernels by the OpenCL runtime.This is due to the presence of non-trivial C\ destructors for program scope variables. The ND-range for executing any program clean up kernel is (1,1,1). The order of execution of clean up kernels from different programs (that are linked together) is undefined.

Note that C initializers, C constructors, or C destructors for program scope variables cannot use pointers to coarse grain and fine grain SVM allocations.

A command may be submitted to a device and yet have no visible side effects outside of waiting on and satisfying event dependences. Examples include markers, kernels executed over ranges of no work-items or copy operations with zero sizes. Such commands may pass directly from the *ready* state to the *ended* state.

Command execution can be blocking or non-blocking. Consider a sequence of OpenCL commands. For blocking commands, the OpenCL API functions that enqueue commands don't return until the command has completed. Alternatively, OpenCL functions that enqueue non-blocking commands return immediately and require that a programmer defines dependencies between enqueued commands to ensure that enqueued commands are not launched before needed resources are available. In both cases, the actual execution of the command may occur asynchronously with execution of the host program.

Commands within a single command-queue execute relative to each other in one of two modes:

- **In-order Execution**: Commands and any side effects associated with commands appear to the OpenCL application as if they execute in the same order they are enqueued to a command-queue.
- **Out-of-order Execution**: Commands execute in any order constrained only by explicit synchronization points (e.g. through command queue barriers) or explicit dependencies on events.

Multiple command-queues can be present within a single context. Multiple command-queues execute commands independently. Event objects visible to the host program can be used to define synchronization points between commands in multiple command queues. If such synchronization points are established between commands in multiple command-queues, an implementation must assure that the command-queues progress concurrently and correctly account for the dependencies established by the synchronization points. For a detailed explanation of synchronization points, see section 3.2.4.

The core of the OpenCL execution model is defined by how the kernels execute. When a kernel-enqueue command submits a kernel for execution, an index space is defined. The kernel, the argument values associated with the arguments to the kernel, and the parameters that define the index space define a *kernel-instance*. When a kernel-instance executes on a device, the kernel function executes for each point in the defined index space. Each of these executing kernel functions is called a *work-item*. The work-items associated with a given kernel-instance are managed by the device in groups called *work-groups*. These work-groups define a coarse grained decomposition of the Index space. Work-groups are further divided into *sub-groups*, which provide an additional level of control over execution.

Work-items have a global ID based on their coordinates within the Index space. They can also be defined in terms of their work-group and the local ID within a work-group. The details of this mapping are described in the following section.

3.2.1 Execution Model: Mapping work-items onto an NDRange

The index space supported by OpenCL is called an NDRange. An NDRange is an N-dimensional index space, where N is one, two or three. The NDRange is decomposed into work-groups forming blocks that cover the Index space. An NDRange is defined by three integer arrays of length N:

- The extent of the index space (or global size) in each dimension.
- An offset index F indicating the initial value of the indices in each dimension (zero by default).

- The size of a work-group (local size) in each dimension.

Each work-items global ID is an N-dimensional tuple. The global ID components are values in the range from F, to F plus the number of elements in that dimension minus one.

If a kernel is created from OpenCL C 2.0 or SPIR-V, the size of work-groups in an NDRange (the local size) need not be the same for all work-groups. In this case, any single dimension for which the global size is not divisible by the local size will be partitioned into two regions. One region will have work-groups that have the same number of work items as was specified for that dimension by the programmer (the local size). The other region will have work-groups with less than the number of work items specified by the local size parameter in that dimension (the *remainder work-groups*). Work-group sizes could be non-uniform in multiple dimensions, potentially producing work-groups of up to 4 different sizes in a 2D range and 8 different sizes in a 3D range.

Each work-item is assigned to a work-group and given a local ID to represent its position within the work-group. A work-item's local ID is an N-dimensional tuple with components in the range from zero to the size of the work-group in that dimension minus one.

Work-groups are assigned IDs similarly. The number of work-groups in each dimension is not directly defined but is inferred from the local and global NDRanges provided when a kernel-instance is enqueued. A work-group's ID is an N-dimensional tuple with components in the range 0 to the ceiling of the global size in that dimension divided by the local size in the same dimension. As a result, 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 3-3. We input the index space for the work-items (G_x, G_y), the size of each work-group (S_x, S_y) and the global ID offset (F_x, F_y). The global indices define an $G_x$by G_y index space where the total number of work-items is the product of G_x and G_y. The local indices define an 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 work-group such that

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

The number of work-groups can be computed as:

$$(W_x, W_y) = (\text{ceil}(G_x / S_x),\text{ceil}(G_y / S_y))$$

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

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

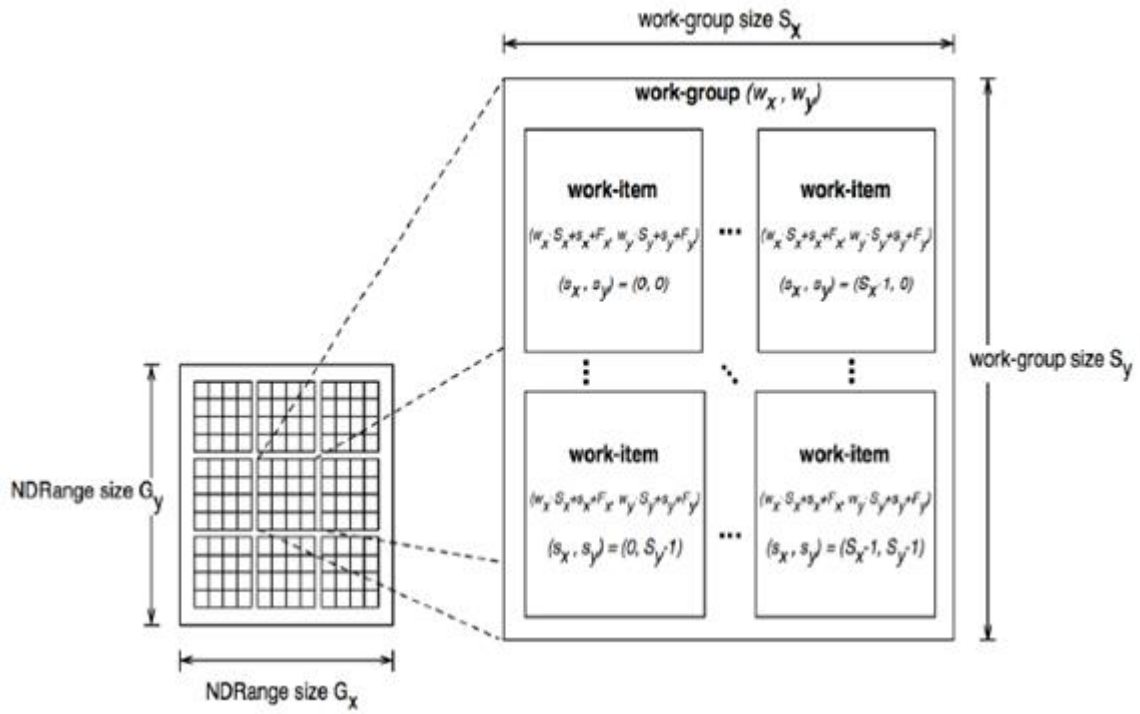

Figure 3-3: An example of an NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs. In this case, we assume that in each dimension, the size of the work-group evenly divides the global NDRange size (i.e. all work-groups have the same size) and that the offset is equal to zero.

Within a work-group work-items may be divided into sub-groups. The mapping of work-items to sub-groups is implementation-defined and may be queried at runtime. While sub-groups may be used in multi-dimensional work-groups, each sub-group is 1-dimensional and any given work-item may query which sub-group it is a member of.

Work items are mapped into sub-groups through a combination of compile-time decisions and the parameters of the dispatch. The mapping to sub-groups is invariant for the duration of a kernels execution, across dispatches of a given kernel with the same work-group dimensions, between dispatches and query operations consistent with the dispatch parameterization, and from one work-group to another within the dispatch (excluding the trailing edge work-groups in the presence of non-uniform work-group sizes). In addition, all sub-groups within a work-group will be the same size, apart from the sub-group with the maximum index which may be smaller if the size of the work-group is not evenly divisible by the size of the sub-groups.

In the degenerate case, a single sub-group must be supported for each work-group. In this situation all sub-group scope functions are equivalent to their work-group level equivalents.

3.2.2 Execution Model: Execution of kernel-instances

The work carried out by an OpenCL program occurs through the execution of kernel-instances on compute devices. To understand the details of OpenCLs execution model, we need to consider how a kernel object moves from the kernel-enqueue command, into a command-queue, executes on a device, and completes.

A kernel-object is defined from a function within the program object and a collection of arguments connecting the kernel to a set of argument values. The host program enqueues a kernel-object to the command queue along with the NDRange, and the work-group decomposition. These define a *kernel-instance*. In addition, an optional set of events may be defined when the kernel is enqueued. The events associated with a particular kernel-instance are used to constrain when the kernel-instance is launched with respect to other commands in the queue or to commands in other queues within the same

context.

A kernel-instance is submitted to a device. For an in-order command queue, the kernel instances appear to launch and then execute in that same order; where we use the term appear to emphasize that when there are no dependencies between commands and hence differences in the order that commands execute cannot be observed in a program, an implementation can reorder commands even in an in-order command queue. For an out of order command-queue, kernel-instances wait to be launched until:

- Synchronization commands enqueued prior to the kernel-instance are satisfied.
- Each of the events in an optional event list defined when the kernel-instance was enqueued are set to CL_COMPLETE.

Once these conditions are met, the kernel-instance is launched and the work-groups associated with the kernel-instance are placed into a pool of ready to execute work-groups. This pool is called a *work-pool*. The work-pool may be implemented in any manner as long as it assures that work-groups placed in the pool will eventually execute. The device schedules work-groups from the work-pool for execution on the compute units of the device. The kernel-enqueue command is complete when all work-groups associated with the kernel-instance end their execution, updates to global memory associated with a command are visible globally, and the device signals successful completion by setting the event associated with the kernel-enqueue command to CL_COMPLETE.

While a command-queue is associated with only one device, a single device may be associated with multiple command-queues all feeding into the single work-pool. A device may also be associated with command queues associated with different contexts within the same platform, again all feeding into the single work-pool. The device will pull work-groups from the work-pool and execute them on one or several compute units in any order; possibly interleaving execution of work-groups from multiple commands. A conforming implementation may choose to serialize the work-groups so a correct algorithm cannot assume that work-groups will execute in parallel. There is no safe and portable way to synchronize across the independent execution of work-groups since once in the work-pool, they can execute in any order.

The work-items within a single sub-group execute concurrently but not necessarily in parallel (i.e. they are not guaranteed to make independent forward progress). Therefore, only high-level synchronization constructs (e.g. sub-group functions such as barriers) that apply to all the work-items in a sub-group are well defined and included in OpenCL.

Sub-groups execute concurrently within a given work-group and with appropriate device support (*see Section__4.2*) may make independent forward progress with respect to each other, with respect to host threads and with respect to any entities external to the OpenCL system but running on an OpenCL device, even in the absence of work-group barrier operations. In this situation, sub-groups are able to internally synchronize using barrier operations without synchronizing with each other and may perform operations that rely on runtime dependencies on operations other sub-groups perform.

The work-items within a single work-group execute concurrently but are only guaranteed to make independent progress in the presence of sub-groups and device support. In the absence of this capability, only high-level synchronization constructs (e.g. work-group functions such as barriers) that apply to all the work-items in a work-group are well defined and included in OpenCL for synchronization within the work-group.

In the absence of synchronization functions (e.g. a barrier), work-items within a sub-group may be serialized. In the presence of sub -group functions, work-items within a sub -group may be serialized before any given sub -group function, between dynamically encountered pairs of sub -group functions and between a work-group function and the end of the kernel.

In the absence of independent forward progress of constituent sub-groups, work-items within a work-group may be serialized before, after or between work-group synchronization functions.

3.2.3 Execution Model: Device-side enqueue

Algorithms may need to generate additional work as they execute. In many cases, this additional work cannot be determined statically; so the work associated with a kernel only emerges at runtime as the kernel-instance executes. This

capability could be implemented in logic running within the host program, but involvement of the host may add significant overhead and/or complexity to the application control flow. A more efficient approach would be to nest kernel-enqueue commands from inside other kernels. This **nested parallelism** can be realized by supporting the enqueuing of kernels on a device without direct involvement by the host program; so-called **device-side enqueue**.

Device-side kernel-enqueue commands are similar to host-side kernel-enqueue commands. The kernel executing on a device (the **parent kernel**) enqueues a kernel-instance (the **child kernel**) to a device-side command queue. This is an out-of-order command-queue and follows the same behavior as the out-of-order command-queues exposed to the host program. Commands enqueued to a device side command-queue generate and use events to enforce order constraints just as for the command-queue on the host. These events, however, are only visible to the parent kernel running on the device. When these prerequisite events take on the value CL_COMPLETE, the work-groups associated with the child kernel are launched into the devices work pool. The device then schedules them for execution on the compute units of the device. Child and parent kernels execute asynchronously. However, a parent will not indicate that it is complete by setting its event to CL_COMPLETE until all child kernels have ended execution and have signaled completion by setting any associated events to the value CL_COMPLETE. Should any child kernel complete with an event status set to a negative value (i.e. abnormally terminate), the parent kernel will abnormally terminate and propagate the childs negative event value as the value of the parents event. If there are multiple children that have an event status set to a negative value, the selection of which childs negative event value is propagated is implementation-defined.

3.2.4 Execution Model: Synchronization

Synchronization refers to mechanisms that constrain the order of execution between two or more units of execution. Consider the following three domains of synchronization in OpenCL:

- Work-group synchronization: Constraints on the order of execution for work-items in a single work-group
- Sub-group synchronization: Contraints on the order of execution for work-items in a single sub-group
- Command synchronization: Constraints on the order of commands launched for execution

Synchronization across all work-items within a single work-group is carried out using a *work-group function*. These functions carry out collective operations across all the work-items in a work-group. Available collective operations are: barrier, reduction, broadcast, prefix sum, and evaluation of a predicate. A work-group function must occur within a converged control flow; i.e. all work-items in the work-group must encounter precisely the same work-group function. For example, if a work-group function occurs within a loop, the work-items must encounter the same work-group function in the same loop iterations. All the work-items of a work-group must execute the work-group function and complete reads and writes to memory before any are allowed to continue execution beyond the work-group function. Work-group functions that apply between work-groups are not provided in OpenCL since OpenCL does not define forward-progress or ordering relations between work-groups, hence collective synchronization operations are not well defined.

Synchronization across all work-items within a single sub-group is carried out using a *sub-group function*. These functions carry out collective operations across all the work-items in a sub-group. Available collective operations are: barrier, reduction, broadcast, prefix sum, and evaluation of a predicate. A sub-group function must occur within a converged control flow; i.e. all work-items in the sub-group must encounter precisely the same sub-group function. For example, if a work-group function occurs within a loop, the work-items must encounter the same sub-group function in the same loop iterations. All the work-items of a sub-group must execute the sub-group function and complete reads and writes to memory before any are allowed to continue execution beyond the sub-group function. Synchronization between sub-groups must either be performed using work-group functions, or through memory operations. Using memory operations for sub-group synchronization should be used carefully as forward progress of sub-groups relative to each other is only supported optionally by OpenCL implementations.

Command synchronization is defined in terms of distinct **synchronization points**. The synchronization points occur between commands in host command-queues and between commands in device-side command-queues. The synchronization points defined in OpenCL include:

- **Launching a command:** A kernel-instance is launched onto a device after all events that kernel is waiting-on have been set to CL_COMPLETE.

- **Ending a command:** Child kernels may be enqueued such that they wait for the parent kernel to reach the *end* state before they can be launched. In this case, the ending of the parent command defines a synchronization point.
- **Completion of a command:** A kernel-instance is complete after all of the work-groups in the kernel and all of its child kernels have completed. This is signaled to the host, a parent kernel or other kernels within command queues by setting the value of the event associated with a kernel to CL_COMPLETE.
- **Blocking Commands:** A blocking command defines a synchronization point between the unit of execution that calls the blocking API function and the enqueued command reaching the complete state.
- **Command-queue barrier:** The command-queue barrier ensures that all previously enqueued commands have completed before subsequently enqueued commands can be launched.
- **clFinish:** This function blocks until all previously enqueued commands in the command queue have completed after which clFinish defines a synchronization point and the clFinish function returns.

A synchronization point between a pair of commands (A and B) assures that results of command A happens-before command B is launched. This requires that any updates to memory from command A complete and are made available to other commands before the synchronization point completes. Likewise, this requires that command B waits until after the synchronization point before loading values from global memory. The concept of a synchronization point works in a similar fashion for commands such as a barrier that apply to two sets of commands. All the commands prior to the barrier must complete and make their results available to following commands. Furthermore, any commands following the barrier must wait for the commands prior to the barrier before loading values and continuing their execution.

These *happens-before* relationships are a fundamental part of the OpenCL memory model. When applied at the level of commands, they are straightforward to define at a language level in terms of ordering relationships between different commands. Ordering memory operations inside different commands, however, requires rules more complex than can be captured by the high level concept of a synchronization point. These rules are described in detail in section 3.3.6.

3.2.5 Execution Model: Categories of Kernels

The OpenCL execution model supports three types of kernels:

- **OpenCL kernels** are managed by the OpenCL API as kernel-objects associated with kernel functions within program-objects. OpenCL kernels are provided via a kernel language. All OpenCL implementations must support OpenCL kernels supplied in the standard SPIR-V intermediate language with the appropriate environment specification, and the OpenCL C programming language defined in earlier versions of the OpenCL specification. Implementations must also support OpenCL kernels in SPIR-V intermediate language. SPIR-V binaries nay be generated from an OpenCL kernel language or by a third party compiler from an alternative input.
- **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. The ability to execute native kernels is optional 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.
- **Built-in kernels** are tied to particular device and are not built at runtime from source code in a program object. The common use of built in kernels is to expose fixed-function hardware or firmware associated with a particular OpenCL device or custom device. The semantics of a built-in kernel may be defined outside of OpenCL and hence are implementation defined.

All three types of kernels are manipulated through the OpenCL command queues and must conform to the synchronization points defined in the OpenCL execution model.

3.3 Memory Model

The OpenCL memory model describes the structure, contents, and behavior of the memory exposed by an OpenCL platform as an OpenCL program runs. The model allows a programmer to reason about values in memory as the host

program and multiple kernel-instances execute.

An OpenCL program defines a context that includes a host, one or more devices, command-queues, and memory exposed within the context. Consider the units of execution involved with such a program. The host program runs as one or more host threads managed by the operating system running on the host (the details of which are defined outside of OpenCL). There may be multiple devices in a single context which all have access to memory objects defined by OpenCL. On a single device, multiple work-groups may execute in parallel with potentially overlapping updates to memory. Finally, within a single work-group, multiple work-items concurrently execute, once again with potentially overlapping updates to memory.

The memory model must precisely define how the values in memory as seen from each of these units of execution interact so a programmer can reason about the correctness of OpenCL programs. We define the memory model in four parts.

- Memory regions: The distinct memories visible to the host and the devices that share a context.
- Memory objects: The objects defined by the OpenCL API and their management by the host and devices.
- Shared Virtual Memory: A virtual address space exposed to both the host and the devices within a context.
- Consistency Model: Rules that define which values are observed when multiple units of execution load data from memory plus the atomic/fence operations that constrain the order of memory operations and define synchronization relationships.

3.3.1 Memory Model: Fundamental Memory Regions

Memory in OpenCL is divided into two parts.

- **Host Memory:** The memory directly available to the host. The detailed behavior of host memory is defined outside of OpenCL. Memory objects move between the Host and the devices through functions within the OpenCL API or through a shared virtual memory interface.
- **Device Memory:** Memory directly available to kernels executing on OpenCL devices.

Device memory consists of four named address spaces or *memory regions*:

- **Global Memory:** This memory region permits read/write access to all work-items in all work-groups running on any device within a context. 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-instance. 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.
- **Private Memory**: A region of memory private to a work-item. Variables defined in one work-items private memory are not visible to another work-item.

The memory regions and their relationship to the OpenCL Platform model are summarized in figure 3-4. Local and private memories are always associated with a particular device. The global and constant memories, however, are shared between all devices within a given context. An OpenCL device may include a cache to support efficient access to these shared memories

To understand memory in OpenCL, it is important to appreciate the relationships between these named address spaces. The four named address spaces available to a device are disjoint meaning they do not overlap. This is a logical relationship, however, and an implementation may choose to let these disjoint named address spaces share physical memory.

Programmers often need functions callable from kernels where the pointers manipulated by those functions can point to multiple named address spaces. This saves a programmer from the error-prone and wasteful practice of creating multiple copies of functions; one for each named address space. Therefore the global, local and private address spaces belong to a single *generic address space*. This is closely modeled after the concept of a generic address space used in the embedded C standard (ISO/IEC 9899:1999). Since they all belong to a single generic address space, the following properties are supported for pointers to named address spaces in device memory:

- A pointer to the generic address space can be cast to a pointer to a global, local or private address space
- A pointer to a global, local or private address space can be cast to a pointer to the generic address space.
- A pointer to a global, local or private address space can be implicitly converted to a pointer to the generic address space, but the converse is not allowed.

The constant address space is disjoint from the generic address space.

The addresses of memory associated with memory objects in Global memory are not preserved between kernel instances, between a device and the host, and between devices. In this regard global memory acts as a global pool of memory objects rather than an address space. This restriction is relaxed when shared virtual memory (SVM) is used.

SVM causes addresses to be meaningful between the host and all of the devices within a context hence supporting the use of pointer based data structures in OpenCL kernels. It logically extends a portion of the global memory into the host address space giving work-items access to the host address space. On platforms with hardware support for a shared address space between the host and one or more devices, SVM may also provide a more efficient way to share data between devices and the host. Details about SVM are presented in section 3.3.3.

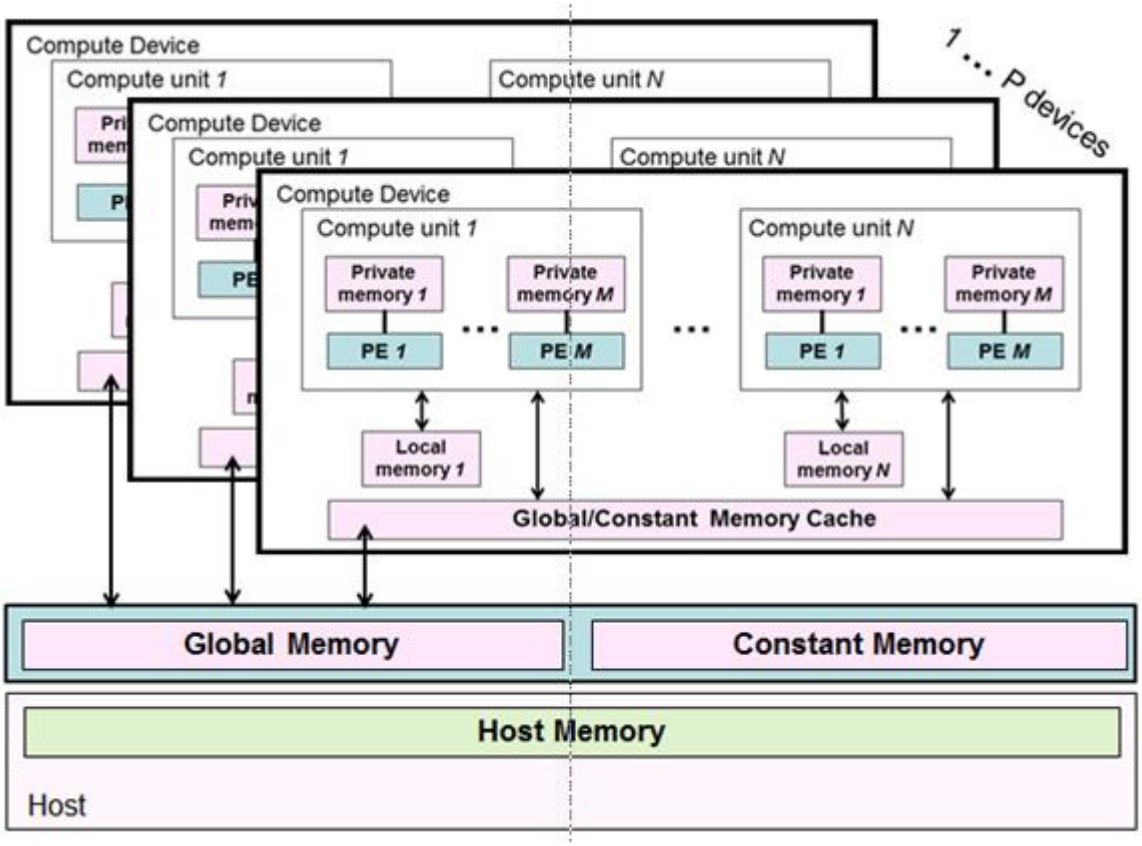

Figure 3-4: The named address spaces exposed in an OpenCL Platform. Global and Constant memories are shared between the one or more devices within a context, while local and private memories are associated with a single

device. Each device may include an optional cache to support efficient access to their view of the global and constant address spaces.

A programmer may use the features of the memory consistency model (section 3.3.4) to manage safe access to global memory from multiple work-items potentially running on one or more devices. In addition, when using shared virtual memory (SVM), the memory consistency model may also be used to ensure that host threads safely access memory locations in the shared memory region.

3.3.2 Memory Model: Memory Objects

The contents of global memory are *memory objects*. A memory object is a handle to a reference counted region of global memory. Memory objects use the OpenCL type *cl_mem* and fall into three distinct classes.

- **Buffer**: A memory object stored as a block of contiguous memory and used as a general purpose object to hold data used in an OpenCL program. The types of the values within a buffer may be any of the built in types (such as int, float), vector types, or user-defined structures. The buffer can be manipulated through pointers much as one would with any block of memory in C.
- **Image**: An image memory object holds one, two or three dimensional images. The formats are based on the standard image formats used in graphics applications. An image is an opaque data structure managed by functions defined in the OpenCL API. To optimize the manipulation of images stored in the texture memories found in many GPUs, OpenCL kernels have traditionally been disallowed from both reading and writing a single image. In OpenCL 2.0, however, we have relaxed this restriction by providing synchronization and fence operations that let programmers properly synchronize their code to safely allow a kernel to read and write a single image.
- **Pipe**: The *pipe* memory object conceptually is an ordered sequence of data items. A pipe has two endpoints: a write endpoint into which data items are inserted, and a read endpoint from which data items are removed. At any one time, only one kernel instance may write into a pipe, and only one kernel instance may read from a pipe. To support the producer consumer design pattern, one kernel instance connects to the write endpoint (the producer) while another kernel instance connects to the reading endpoint (the consumer).

Memory objects are allocated by host APIs. The host program can provide the runtime with a pointer to a block of continuous memory to hold the memory object when the object is created (CL_MEM_USE_HOST_PTR). Alternatively, the physical memory can be managed by the OpenCL runtime and not be directly accessible to the host program.

Allocation and access to memory objects within the different memory regions varies between the host and work-items running on a device. This is summarized in table 3.1 which__describes whether the kernel or the host can allocate from a memory region, the type of allocation (static at compile time vs. dynamic at 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	Dynamic Allocation	Dynamic Allocation	No Allocation
	Read/Write access to buffers and images but not pipes	Read/Write access	No access	No access
Kernel	Static Allocation for program scope variables	Static Allocation	Static Allocation. Dynamic allocation for child kernel	Static Allocation
	Read/Write access	Read-only access	Read/Write access. No access to child's local memory.	Read/Write access

Table 3 1: The different memory regions in OpenCL and how memory objects are allocated and accessed by the host and by an executing instance of a kernel. For the case of kernels, we distinguish between the behavior of local memory with respect to a kernel (self) and its child kernels.

Once allocated, a memory object is made available to kernel-instances running on one or more devices. In addition to shared virtual memory (section 3.3.3) there are three basic ways to manage the contents of buffers between the host and devices.

- **Read/Write/Fill commands**: The data associated with a memory object is explicitly read and written between the host and global memory regions using commands enqueued to an OpenCL command queue.
- **Map/Unmap commands**: Data from the memory object is mapped into a contiguous block of memory accessed through a host accessible pointer. The host program enqueues a *map* command on block of a memory object before it can be safely manipulated by the host program. When the host program is finished working with the block of memory, the host program enqueues an *unmap* command to allow a kernel-instance to safely read and/or write the buffer.**
- **Copy commands:** The data associated with a memory object is copied between two buffers, each of which may reside either on the host or on the device.

In both cases, the commands to transfer data between devices and the host can be blocking or non-blocking operations. 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.

Memory objects are bound to a context and hence can appear in multiple kernel-instances running on more than one physical device. The OpenCL platform must support a large range of hardware platforms including systems that do not support a single shared address space in hardware; hence the ways memory objects can be shared between kernel-instances is restricted. The basic principle is that multiple read operations on memory objects from multiple kernel-instances that overlap in time are allowed, but mixing overlapping reads and writes into the same memory objects from different kernel instances is only allowed when fine grained synchronization is used with shared virtual memory (see section 3.3.3).

When global memory is manipulated by multiple kernel-instances running on multiple devices, the OpenCL runtime system must manage the association of memory objects with a given device. In most cases the OpenCL runtime will implicitly associate a memory object with a device. A kernel instance is naturally associated with the command queue to which the kernel was submitted. Since a command-queue can only access a single device, the queue uniquely defines

which device is involved with any given kernel-instance; hence defining a clear association between memory objects, kernel-instances and devices. Programmers may anticipate these associations in their programs and explicitly manage association of memory objects with devices in order to improve performance.

3.3.3 Memory Model: Shared Virtual Memory

OpenCL extends the global memory region into the host memory region through a shared virtual memory (SVM) mechanism. There are three types of SVM in OpenCL

- **Coarse-Grained buffer SVM**: Sharing occurs at the granularity of regions of OpenCL buffer memory objects. Consistency is enforced at synchronization points and with map/unmap commands to drive updates between the host and the device. This form of SVM is similar to non-SVM use of memory; however, it lets kernel-instances share pointer-based data structures (such as linked-lists) with the host program. Program scope global variables are treated as per-device coarse-grained SVM for addressing and sharing purposes.
- **Fine-Grained buffer SVM**: Sharing occurs at the granularity of individual loads/stores into bytes within OpenCL buffer memory objects. Loads and stores may be cached. This means consistency is guaranteed at synchronization points. If the optional OpenCL atomics are supported, they can be used to provide fine-grained control of memory consistency.
- **Fine-Grained system SVM**: Sharing occurs at the granularity of individual loads/stores into bytes occurring anywhere within the host memory. Loads and stores may be cached so consistency is guaranteed at synchronization points. If the optional OpenCL atomics are supported, they can be used to provide fine-grained control of memory consistency.

Table 3.1: **A summary of shared virtual memory (SVM) options in OpenCL**

	Granularity of sharing	Memory Allocation	Mechanisms to enforce Consistency	Explicit updates between host and device
Non-SVM buffers	OpenCL Memory objects(buffer)	clCreateBuffer	Host synchronization points on the same or between devices.	yes, through Map and Unmap commands.
Coarse-Grained buffer SVM	OpenCL Memory objects (buffer)	clSVMAlloc	Host synchronization points between devices	yes, through Map and Unmap commands.
Fine Grained buffer SVM	Bytes within OpenCL Memory objects (buffer)	clSVMAlloc	Synchronization points plus atomics (if supported)	No
Fine-Grained system SVM	Bytes within Host memory (system)	Host memory allocation mechanisms (e.g. malloc)	Synchronization points plus atomics (if supported)	No

Coarse-Grained buffer SVM is required in the core OpenCL specification. The two finer grained approaches are optional features in OpenCL. The various SVM mechanisms to access host memory from the work-items associated with a kernel instance are summarized in table 3-2.

3.3.4 Memory Model: Memory Consistency Model

The OpenCL memory model tells programmers what they can expect from an OpenCL implementation; which memory operations are guaranteed to happen in which order and which memory values each read operation will return. The

memory model tells compiler writers which restrictions they must follow when implementing compiler optimizations; which variables they can cache in registers and when they can move reads or writes around a barrier or atomic operation. The memory model also tells hardware designers about limitations on hardware optimizations; for example, when they must flush or invalidate hardware caches.

The memory consistency model in OpenCL is based on the memory model from the ISO C11 programming language. To help make the presentation more precise and self-contained, we include modified paragraphs taken verbatim from the ISO C11 international standard. When a paragraph is taken or modified from the C11 standard, it is identified as such along with its original location in the C11 standard.

For programmers, the most intuitive model is the *sequential consistency* memory model. Sequential consistency interleaves the steps executed by each of the units of execution. Each access to a memory location sees the last assignment to that location in that interleaving. While sequential consistency is relatively straightforward for a programmer to reason about, implementing sequential consistency is expensive. Therefore, OpenCL implements a relaxed memory consistency model; i.e. it is possible to write programs where the loads from memory violate sequential consistency. Fortunately, if a program does not contain any races and if the program only uses atomic operations that utilize the sequentially consistent memory order (the default memory ordering for OpenCL), OpenCL programs appear to execute with sequential consistency.

Programmers can to some degree control how the memory model is relaxed by choosing the memory order for synchronization operations. The precise semantics of synchronization and the memory orders are formally defined in section 3.3.6. Here, we give a high level description of how these memory orders apply to atomic operations on atomic objects shared between units of execution. OpenCL memory_order choices are based on those from the ANSI C11 standard memory model. They are specified in certain OpenCL functions through the following enumeration constants:

- **memory_order_relaxed**: implies no order constraints. This memory order can be used safely to increment counters that are concurrently incremented, but it doesnt guarantee anything about the ordering with respect to operations to other memory locations. It can also be used, for example, to do ticket allocation and by expert programmers implementing lock-free algorithms.
- **memory_order_acquire**: A synchronization operation (fence or atomic) that has acquire semantics "acquires" side-effects from a release operation that synchronises with it: if an acquire synchronises with a release, the acquiring unit of execution will see all side-effects preceding that release (and possibly subsequent side-effects.) As part of carefully-designed protocols, programmers can use an "acquire" to safely observe the work of another unit of execution.
- **memory_order_release**: A synchronization operation (fence or atomic operation) that has release semantics "releases" side effects to an acquire operation that synchronises with it. All side effects that precede the release are included in the release. As part of carefully-designed protocols, programmers can use a "release" to make changes made in one unit of execution visible to other units of execution.

Note

In general, no acquire must *always* synchronise with any particular release. However, synchronisation can be forced by certain executions. See 3.3.6.2 for detailed rules for when synchronisation must occur.

- **memory_order_acq_rel**: A synchronization operation with acquire-release semantics has the properties of both the acquire and release memory orders. It is typically used to order read-modify-write operations.
- **memory_order_seq_cst**: The loads and stores of each unit of execution appear to execute in program (i.e., sequenced-before) order, and the loads and stores from different units of execution appear to be simply interleaved.

Regardless of which memory_order is specified, resolving constraints on memory operations across a heterogeneous platform adds considerable overhead to the execution of a program. An OpenCL platform may be able to optimize certain operations that depend on the features of the memory consistency model by restricting the scope of the memory operations. Distinct memory scopes are defined by the values of the memory_scope enumeration constant:

- **memory_scope_work_item**: memory-ordering constraints only apply within the work-item.[1].

[1] This value for memory_scope can only be used with atomic_work_item_fence with flags set to LCK_IMAGE_MEM_FENCE.

- **memory_scope_sub_group**:memory-ordering constraints only apply within the sub-group.
- **memory_scope_work_group**: memory-ordering constraints only apply to work-items executing within a single work-group.
- **memory_scope_device:** memory-ordering constraints only apply to work-items executing on a single device
- **memory_scope_all_svm_devices**: memory-ordering constraints apply to work-items executing across multiple devices and (when using SVM) the host. A release performed with **memory_scope_all_svm_devices** to a buffer that does not have the CL_MEM_SVM_ATOMICS flag set will commit to at least **memory_scope_device** visibility, with full synchronization of the buffer at a queue synchronization point (e.g. an OpenCL event).

  These memory scopes define a hierarchy of visibilities when analyzing the ordering constraints of memory operations. For example if a programmer knows that a sequence of memory operations will only be associated with a collection of work-items from a single work-group (and hence will run on a single device), the implementation is spared the overhead of managing the memory orders across other devices within the same context. This can substantially reduce overhead in a program. All memory scopes are valid when used on global memory or local memory. For local memory, all visibility is constrained to within a given work-group and scopes wider than **memory_scope_work_group** carry no additional meaning.

  In the following subsections (leading up to section 3.4), we will explain the synchronization constructs and detailed rules needed to use OpenCLs relaxed memory models. It is important to appreciate, however, that many programs do not benefit from relaxed memory models. Even expert programmers have a difficult time using atomics and fences to write correct programs with relaxed memory models. A large number of OpenCL programs can be written using a simplified memory model. This is accomplished by following these guidelines.
- Write programs that manage safe sharing of global memory objects through the synchronization points defined by the command queues.
- Restrict low level synchronization inside work-groups to the work-group functions such as barrier.
- If you want sequential consistency behavior with system allocations or fine-grain SVM buffers with atomics support, use only memory_order_seq_cst operations with the scope memory_scope_all_svm_devices.
- If you want sequential consistency behavior when not using system allocations or fine-grain SVM buffers with atomics support, use only memory_order_seq_cst operations with the scope memory_scope_device or memory_scope_all_svm_devices.
- Ensure your program has no races.

  If these guidelines are followed in your OpenCL programs, you can skip the detailed rules behind the relaxed memory models and go directly to section 3.4.

3.3.5 Memory Model: Overview of atomic and fence operations

The OpenCL 2.0 specification defines a number of *synchronization operations* that are used to define memory order constraints in a program. They play a special role in controlling how memory operations in one unit of execution (such as work-items or, when using SVM a host thread) are made visible to another. There are two types of synchronization operations in OpenCL; *atomic operations* and *fences*.

Atomic operations are indivisible. They either occur completely or not at all. These operations are used to order memory operations between units of execution and hence they are parameterized with the memory_order and memory_scope parameters defined by the OpenCL memory consistency model. The atomic operations for OpenCL kernel languages are similar to the corresponding operations defined by the C11 standard.

The OpenCL 2.0 atomic operations apply to variables of an atomic type (a subset of those in the C11 standard) including atomic versions of the int, uint, long, ulong, float, double, half, intptr_t, uintptr_t, size_t, and ptrdiff_t types. However, support for some of these atomic types depends on support for the corresponding regular types.

An atomic operation on one or more memory locations is either an acquire operation, a release operation, or both an

acquire and release operation. An atomic operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which do not have synchronization properties, and atomic read-modify-write operations, which have special characteristics. [C11 standard, Section 5.1.2.4, paragraph 5, modified]

The orders memory_order_acquire (used for reads), memory_order_release (used for writes), and memory_order_acq_rel (used for read-modify-write operations) are used for simple communication between units of execution using shared variables. Informally, executing a memory_order_release on an atomic object A makes all previous side effects visible to any unit of execution that later executes a memory_order_acquire on A. The orders memory_order_acquire, memory_order_release, and memory_order_acq_rel do not provide sequential consistency for race-free programs because they will not ensure that atomic stores followed by atomic loads become visible to other threads in that order.

The fence operation is atomic_work_item_fence, which includes a memory_order argument as well as the memory_scope and cl_mem_fence_flags arguments. Depending on the memory_order argument, this operation:

- has no effects, if memory_order_relaxed;
- is an acquire fence, if memory_order_acquire;
- is a release fence, if memory_order_release;
- is both an acquire fence and a release fence, if memory_order_acq_rel;
- is a sequentially-consistent fence with both acquire and release semantics, if memory_order_seq_cst.

If specified, the cl_mem_fence_flags argument must be CLK_IMAGE_MEM_FENCE, CLK_GLOBAL_MEM_FENCE, CLK_LOCAL_MEM_FENCE, or CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE.

The atomic_work_item_fence(CLK_IMAGE_MEM_FENCE) built-in function must be

used to make sure that sampler-less writes are visible to later reads by the same work-item. Without use of the atomic_work_item_fence function, write-read coherence on image objects is not guaranteed: if a work-item reads from an image to which it has previously written without an intervening atomic_work_item_fence, it is not guaranteed that those previous writes are visible to the work-item.

The synchronization operations in OpenCL can be parameterized by a memory_scope. Memory scopes control the extent that an atomic operation or fence is visible with respect to the memory model. These memory scopes may be used when performing atomic operations and fences on global memory and local memory. When used on global memory visibility is bounded by the capabilities of that memory. When used on a fine-grained non-atomic SVM buffer, a coarse-grained SVM buffer, or a non-SVM buffer, operations parameterized with memory_scope_all_svm_devices will behave as if they were parameterized with memory_scope_device. When used on local memory, visibility is bounded by the work-group and, as a result, memory_scope with wider visibility than memory_scope_work_group will be reduced to memory_scope_work_group.

Two actions **A** and **B** are defined to have an inclusive scope if they have the same scope **P** such that:

- **P** is memory_scope_sub_group and **A** and **B** are executed by work-items within the same sub-group.
- **P** is memory_scope_work_group and **A** and **B** are executed by work-items within the same work-group.
- **P** is memory_scope_device and **A** and **B** are executed by work-items on the same device when **A** and **B** apply to an SVM allocation or **A** and **B** are executed by work-items in the same kernel or one of its children when **A** and **B** apply to a cl_mem buffer.
- **P** is memory_scope_all_svm_devices if **A** and **B** are executed by host threads or by work-items on one or more devices that can share SVM memory with each other and the host process.

3.3.6 Memory Model: Memory Ordering Rules

Fundamentally, the issue in a memory model is to understand the orderings in time of modifications to objects in memory. Modifying an object or calling a function that modifies an object are side effects, i.e. changes in the state of the execution environment. Evaluation of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object. [C11 standard, Section 5.1.2.3, paragraph 2, modified]

We assume that the OpenCL kernel language and host programming languages have a sequenced-before relation between the evaluations executed by a single unit of execution. This sequenced-before relation is an asymmetric, transitive, pair-wise relation between those evaluations, which induces a partial order among them. Given any two evaluations **A** and **B**, if **A** is sequenced-before **B**, then the execution of **A** shall precede the execution of **B**. (Conversely, if **A** is sequenced-before **B**, then **B** is sequenced-after **A**.) If **A** is not sequenced-before or sequenced-after **B**, then **A** and **B** are unsequenced. Evaluations **A** and **B** are indeterminately sequenced when **A** is either sequenced-before or sequenced-after **B**, but it is unspecified which. [C11 standard, Section 5.1.2.3, paragraph 3, modified]

NOTE: sequenced-before is a partial order of the operations executed by a single unit of execution (e.g. a host thread or work-item). It generally corresponds to the source program order of those operations, and is partial because of the undefined argument evaluation order of OpenCLs kernel C language.

In an OpenCL kernel language, the value of an object visible to a work-item W at a particular point is the initial value of the object, a value stored in the object by W, or a value stored in the object by another work-item or host thread, according to the rules below. Depending on details of the host programming language, the value of an object visible to a host thread may also be the value stored in that object by another work-item or host thread. [C11 standard, Section 5.1.2.4, paragraph 2, modified]

Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location. [C11 standard, Section 5.1.2.4, paragraph 4]

All modifications to a particular atomic object **M** occur in some particular total order, called the modification order of **M**. If **A** and **B** are modifications of an atomic object **M**, and **A** happens-before **B**, then **A** shall precede **B** in the modification order of **M**, which is defined below. Note that the modification order of an atomic object **M** is independent of whether **M** is in local or global memory. [C11 standard, Section 5.1.2.4, paragraph 7, modified]

A release sequence begins with a release operation **A** on an atomic object **M** and is the maximal contiguous sub-sequence of side effects in the modification order of **M**, where the first operation is **A** and every subsequent operation either is performed by the same work-item or host thread that performed the release or is an atomic read-modify-write operation. [C11 standard, Section 5.1.2.4, paragraph 10, modified]

OpenCLs local and global memories are disjoint. Kernels may access both kinds of memory while host threads may only access global memory. Furthermore, the *flags* argument of OpenCLs work_group_barrier function specifies which memory operations the function will make visible: these memory operations can be, for example, just the ones to local memory, or the ones to global memory, or both. Since the visibility of memory operations can be specified for local memory separately from global memory, we define two related but independent relations, *global-synchronizes-with* and *local-synchronizes-with*. Certain operations on global memory may global-synchronize-with other operations performed by another work-item or host thread. An example is a release atomic operation in one work- item that global-synchronizes-with an acquire atomic operation in a second work-item. Similarly, certain atomic operations on local objects in kernels can local-synchronize- with other atomic operations on those local objects. [C11 standard, Section 5.1.2.4, paragraph 11, modified]

We define two separate happens-before relations: global-happens-before and local-happens-before.

A global memory action **A** global-happens-before a global memory action *B*__if

- **A** is sequenced before **B**, or
- **A** global-synchronizes-with **B**, or

- For some global memory action **C**, **A** global-happens-before **C** and **C** global-happens-before **B**.

A local memory action **A** local-happens-before a local memory action *B*__if

- **A** is sequenced before **B**, or
- **A** local-synchronizes-with **B**, or
- For some local memory action **C**, **A** local-happens-before **C** and **C** local-happens-before **B**.

An OpenCL implementation shall ensure that no program execution demonstrates a cycle in either the local-happens-before relation or the global-happens-before relation.

NOTE: The global- and local-happens-before relations are critical to defining what values are read and when data races occur. The global-happens-before relation, for example, defines what global memory operations definitely happen before what other global memory operations. If an operation **A** global-happens-before operation **B** then **A** must occur before **B**; in particular, any write done by **A** will be visible to **B**. The local-happens-before relation has similar properties for local memory. Programmers can use the local- and global-happens-before relations to reason about the order of program actions.

A visible side effect **A** on a global object **M** with respect to a value computation **B** of **M** satisfies the conditions:

- **A** global-happens-before **B**, and
- there is no other side effect **X** to **M** such that **A** global-happens-before **X** and **X** global-happens-before **B**.

We define visible side effects for local objects **M** similarly. The value of a non-atomic scalar object **M**, as determined by evaluation **B**, shall be the value stored by the visible side effect **A**. [C11 standard, Section 5.1.2.4, paragraph 19, modified]

The execution of a program contains a data race if it contains two conflicting actions A and B in different units of execution, and

- (1) at least one of **A** or **B** is not atomic, or **A** and **B** do not have inclusive memory scope, and
- (2) the actions are global actions unordered by the global-happens-before relation or are local actions unordered by the local-happens-before relation.

Any such data race results in undefined behavior. [C11 standard, Section 5.1.2.4, paragraph 25, modified]

We also define the visible sequence of side effects on local and global atomic objects. The remaining paragraphs of this subsection define this sequence for a global atomic object **M**; the visible sequence of side effects for a local atomic object is defined similarly by using the local-happens-before relation.

The visible sequence of side effects on a global atomic object **M**, with respect to a value computation **B** of **M**, is a maximal contiguous sub-sequence of side effects in the modification order of **M**, where the first side effect is visible with respect to **B**, and for every side effect, it is not the case that **B** global-happens-before it. The value of*M*, as determined by evaluation **B**, shall be the value stored by some operation in the visible sequence of **M** with respect to **B**. [C11 standard, Section 5.1.2.4, paragraph 22, modified]

If an operation **A** that modifies an atomic object **M** global-happens before an operation **B** that modifies **M**, then **A** shall be earlier than **B** in the modification order of **M**. This requirement is known as write-write coherence.

If a value computation **A** of an atomic object **M** global-happens-before a value computation **B** of **M**, and **A** takes its value from a side effect **X** on **M**, then the value computed by **B** shall either equal the value stored by **X**, or be the value stored by

a side effect **Y** on **M**, where **Y** follows **X** in the modification order of **M**. This requirement is known as read-read coherence. [C11 standard, Section 5.1.2.4, paragraph 22, modified]

If a value computation **A** of an atomic object **M** global-happens-before an operation **B** on **M**, then **A** shall take its value from a side effect **X** on **M**, where **X** precedes **B** in the modification order of **M**. This requirement is known as read-write coherence.

If a side effect **X** on an atomic object **M** global-happens-before a value computation **B** of **M**, then the evaluation **B** shall take its value from **X** or from a side effect **Y** that follows **X*in the modification order of *M**. This requirement is known as write-read coherence.

3.3.6.1 Memory Ordering Rules: Atomic Operations

This and following sections describe how different program actions in kernel C code and the host program contribute to the local- and global-happens-before relations. This section discusses ordering rules for OpenCL 2.0s atomic operations.

Section 3.2.3 defined the enumerated type memory_order.

- For memory_order_relaxed, no operation orders memory.
- For memory_order_release, memory_order_acq_rel, and memory_order_seq_cst, a store operation performs a release operation on the affected memory location.
- For memory_order_acquire, memory_order_acq_rel, and memory_order_seq_cst, a load operation performs an acquire operation on the affected memory location. [C11 standard, Section 7.17.3, paragraphs 2-4, modified]

Certain built-in functions synchronize with other built-in functions performed by another unit of execution. This is true for pairs of release and acquire operations under specific circumstances. An atomic operation **A** that performs a release operation on a global object **M** global-synchronizes-with an atomic operation **B** that performs an acquire operation on **M** and reads a value written by any side effect in the release sequence headed by **A**. A similar rule holds for atomic operations on objects in local memory: an atomic operation **A** that performs a release operation on a local object **M** local-synchronizes-with an atomic operation **B** that performs an acquire operation on **M** and reads a value written by any side effect in the release sequence headed by **A**. [C11 standard, Section 5.1.2.4, paragraph 11, modified]

NOTE: Atomic operations specifying memory_order_relaxed are relaxed only with respect to memory ordering. Implementations must still guarantee that any given atomic access to a particular atomic object be indivisible with respect to all other atomic accesses to that object.

There shall exist a single total order **S** for all memory_order_seq_cst operations that is consistent with the modification orders for all affected locations, as well as the appropriate global-happens-before and local-happens-before orders for those locations, such that each memory_order_seq operation **B** that loads a value from an atomic object **M** in global or local memory observes one of the following values:

- the result of the last modification **A** of **M** that precedes **B** in **S**, if it exists, or
- if **A** exists, the result of some modification of **M** in the visible sequence of side effects with respect to **B** that is not memory_order_seq_cst and that does not happen before **A**, or
- if **A** does not exist, the result of some modification of **M** in the visible sequence of side effects with respect to **B** that is not memory_order_seq_cst. [C11 standard, Section 7.17.3, paragraph 6, modified]

Let X and Y be two memory_order_seq_cst operations. If X local-synchronizes-with or global-synchronizes-with Y then X both local-synchronizes-with Y and global-synchronizes-with Y.

If the total order **S** exists, the following rules hold:

- For an atomic operation **B** that reads the value of an atomic object **M**, if there is a memory_order_seq_cst fence **X** sequenced-before **B**, then **B** observes either the last memory_order_seq_cst modification of **M** preceding **X** in the total order **S** or a later modification of **M** in its modification order. [C11 standard, Section 7.17.3, paragraph 9]

- For atomic operations **A** and **B** on an atomic object **M**, where **A** modifies **M** and **B** takes its value, if there is a memory_order_seq_cst fence **X** such that **A** is sequenced-before **X** and **B** follows **X** in **S**, then **B** observes either the effects of **A** or a later modification of **M** in its modification order. [C11 standard, Section 7.17.3, paragraph 10]

- For atomic operations **A** and **B** on an atomic object **M**, where **A*modifies *M** and **B** takes its value, if there are memory_order_seq_cst fences **X** and **Y** such that **A** is sequenced-before **X**, **Y** is sequenced-before **B**, and **X** precedes **Y** in **S**, then **B** observes either the effects of **A** or a later modification of **M** in its modification order. [C11 standard, Section 7.17.3, paragraph 11]

- For atomic operations **A** and **B** on an atomic object **M**, if there are memory_order_seq_cst fences **X** and **Y** such that **A** is sequenced-before **X**, **Y** is sequenced-before **B**, and **X** precedes **Y** in **S**, then **B** occurs later than **A** in the modification order of **M**.

Note

memory_order_seq_cst ensures sequential consistency only for a program that is (1) free of data races, and (2) exclusively uses memory_order_seq_cst synchronization operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular, memory_order_seq_cst fences ensure a total order only for the fences themselves. Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications.

Atomic read-modify-write operations should always read the last value (in the modification order) stored before the write associated with the read-modify-write operation. [C11 standard, Section 7.17.3, paragraph 12]

Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation.

Note: Under the rules described above, and independent to the previously footnoted C++ issue, it is known that $x == y == 42$ is a valid final state in the following problematic example:

```
global atomic_int x = ATOMIC_VAR_INIT(0);

local atomic_int y = ATOMIC_VAR_INIT(0);


unit_of_execution_1:

... [execution not reading or writing x or y, leading up to:]

int t = atomic_load_explicit(&y, memory_order_acquire);

atomic_store_explicit(&x, t, memory_order_release);
```

```
unit_of_execution_2:

... [execution not reading or writing x or y, leading up to:]

int t = atomic_load_explicit(&x, memory_order_acquire);

atomic_store_explicit(&y, t,
memory_order_release);link:#_msocom_6[[BA6]]~
```

This is not useful behavior and implementations should not exploit this phenomenon. It should be expected that in the future this may be disallowed by appropriate updates to the memory model description by the OpenCL committee.

Implementations should make atomic stores visible to atomic loads within a reasonable amount of time. [C11 standard, Section 7.17.3, paragraph 16]

As long as the following conditions are met, a host program sharing SVM memory with a kernel executing on one or more OpenCL devices may use atomic and synchronization operations to ensure that its assignments, and those of the kernel, are visible to each other:

1. Either fine-grained buffer or fine-grained system SVM must be used to share memory. While coarse-grained buffer SVM allocations may support atomic operations, visibility on these allocations is not guaranteed except at map and unmap operations.
2. The optional OpenCL 2.0 SVM atomic-controlled visibility specified by provision of the CL_MEM_SVM_ATOMICS flag must be supported by the device and the flag provided to the SVM buffer on allocation.
3. The host atomic and synchronization operations must be compatible with those of an OpenCL kernel language. This requires that the size and representation of the data types that the host atomic operations act on be consistent with the OpenCL kernel language atomic types.

If these conditions are met, the host operations will apply at all_svm_devices scope.

3.3.6.2 Memory Ordering Rules: Fence Operations

This section describes how the OpenCL 2.0 fence operations contribute to the local- and global-happens-before relations.

Earlier, we introduced synchronization primitives called fences. Fences can utilize the acquire memory_order, release memory_order, or both. A fence with acquire semantics is called an acquire fence; a fence with release semantics is called a release fence.

A global release fence **A** global-synchronizes-with a global acquire fence **B** if there exist atomic operations **X** and **Y**, both operating on some global atomic object **M**, such that **A** is sequenced-before **X**, **X** modifies **M**, **Y** is sequenced-before **B**, **Y** reads the value written by **X** or a value written by any side effect in the hypothetical release sequence **X** would head if it were a release operation, and that the scopes of **A**, **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 2, modified.]

A global release fence **A** global-synchronizes-with an atomic operation **B** that performs an acquire operation on a global atomic object **M** if there exists an atomic operation **X** such that **A** is sequenced-before **X**, **X** modifies **M**, **B** reads the value written by **X** or a value written by any side effect in the hypothetical release sequence **X** would head if it were a release operation, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 3, modified.]

An atomic operation **A** that is a release operation on a global atomic object **M** global-synchronizes-with a global acquire fence **B** if there exists some atomic operation **X** on **M** such that **X** is sequenced-before **B** and reads the value written by **A** or a value written by any side effect in the release sequence headed by **A**, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 4, modified.]

A local release fence **A** local-synchronizes-with a local acquire fence **B** if there exist atomic operations **X** and **Y**, both operating on some local atomic object **M**, such that **A** is sequenced-before **X**, **X** modifies **M**, **Y** is sequenced-before **B**, and **Y** reads the value written by **X** or a value written by any side effect in the hypothetical release sequence **X** would head if it were a

release operation, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 2, modified.]

A local release fence **A** local-synchronizes-with an atomic operation **B** that performs an acquire operation on a local atomic object **M** if there exists an atomic operation **X** such that **A** is sequenced-before **X**, **X** modifies **M**, and **B** reads the value written by **X** or a value written by any side effect in the hypothetical release sequence **X** would head if it were a release operation, and

the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 3, modified.]

An atomic operation **A** that is a release operation on a local atomic object **M** local-synchronizes-with a local acquire fence **B** if there exists some atomic operation **X** on **M** such that **X** is sequenced-before **B** and reads the value written by **A** or a value written by any side effect in the release sequence headed by **A**, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 4, modified.]

Let **X** and **Y** be two work item fences that each have both the CLK_GLOBAL_MEM_FENCE and CLK_LOCAL_MEM_FENCE flags set. **X** global-synchronizes-with **Y** and **X** local synchronizes with **Y** if the conditions required for **X** to global-synchronize with **Y** are met, the conditions required for **X** to local-synchronize-with **Y** are met, or both sets of conditions are met.

3.3.6.3 Memory Ordering Rules: Work-group Functions

The OpenCL kernel execution model includes collective operations across the work-items within a single work-group. These are called work-group functions. Besides the work-group barrier function, they include the scan, reduction and pipe work-group functions described in the SPIR-V IL specifications . We will first discuss the work-group barrier. The other work-group functions are discussed afterwards.

The barrier function provides a mechanism for a kernel to synchronize the work-items within a single work-group: informally, each work-item of the work-group must execute the barrier before any are allowed to proceed. It also orders memory operations to a specified combination of one or more address spaces such as local memory or global memory, in a similar manner to a fence.

To precisely specify the memory ordering semantics for barrier, we need to distinguish between a dynamic and a static instance of the call to a barrier. A call to a barrier can appear in a loop, for example, and each execution of the same static barrier call results in a new dynamic instance of the barrier that will independently synchronize a work-groups work-items.

A work-item executing a dynamic instance of a barrier results in two operations, both fences, that are called the entry and exit fences. These fences obey all the rules for fences specified elsewhere in this chapter as well as the following:

- The entry fence is a release fence with the same flags and scope as requested for the barrier.
- The exit fence is an acquire fence with the same flags and scope as requested for the barrier.
- For each work-item the entry fence is sequenced before the exit fence.
- If the flags have CLK_GLOBAL_MEM_FENCE set then for each work-item the entry fence global-synchronizes-with the exit fence of all other work-items in the same work-group.
- If the flags have CLK_LOCAL_MEM_FENCE set then for each work-item the entry fence local-synchronizes-with the exit fence of all other work-items in the same work-group.

The other work-group functions include such functions as work_group_all() and work_group_broadcast() and are described in the kernel language and IL specifications. The use of these work-group functions implies sequenced-before relationships between statements within the execution of a single work-item in order to satisfy data dependencies. For example, a work item that provides a value to a work-group function must behave as if it generates that value before beginning execution of that work-group function. Furthermore, the programmer must ensure that all work items in a work group must execute the same work-group function call site, or dynamic work-group function instance.

3.3.6.4 Memory Ordering Rules: Sub-group Functions

The OpenCL kernel execution model includes collective operations across the work-items within a single sub-group. These are called sub-group functions. Besides the sub-group-barrier function, they include the scan, reduction and pipe sub-group functions described in the SPIR-V IL specification. We will first discuss the sub-group barrier. The other sub-group functions are discussed afterwards.

The barrier function provides a mechanism for a kernel to synchronize the work-items within a single sub-group: informally, each work-item of the sub-group must execute the barrier before any are allowed to proceed. It also orders memory operations to a specified combination of one or more address spaces such as local memory or global memory, in a similar manner to a fence.

To precisely specify the memory ordering semantics for barrier, we need to distinguish between a dynamic and a static instance of the call to a barrier. A call to a barrier can appear in a loop, for example, and each execution of the same static barrier call results in a new dynamic instance of the barrier that will independently synchronize a sub-groups work-items.

A work-item executing a dynamic instance of a barrier results in two operations, both fences, that are called the entry and exit fences. These fences obey all the rules for fences specified elsewhere in this chapter as well as the following:

- The entry fence is a release fence with the same flags and scope as requested for the barrier.
- The exit fence is an acquire fence with the same flags and scope as requested for the barrier.
- For each work-item the entry fence is sequenced before the exit fence.
- If the flags have CLK_GLOBAL_MEM_FENCE set then for each work-item the entry fence global-synchronizes-with the exit fence of all other work-items in the same sub-group.
- If the flags have CLK_LOCAL_MEM_FENCE set then for each work-item the entry fence local-synchronizes-with the exit fence of all other work-items in the same sub-group.

The other sub-group functions include such functions as sub_group_all() and sub_group_broadcast() and are described in OpenCL kernel languages specifications. The use of these sub-group functions implies sequenced-before relationships between statements within the execution of a single work-item in order to satisfy data dependencies. For example, a work item that provides a value to a sub-group function must behave as if it generates that value before beginning execution of that sub-group function. Furthermore, the programmer must ensure that all work items in a sub-group must execute the same sub-group function call site, or dynamic sub-group function instance.

3.3.6.5 Memory Ordering Rules: Host-side and Device-side Commands

This section describes how the OpenCL API functions associated with command-queues contribute to happens-before relations. There are two types of command queues and associated API functions in OpenCL 2.0; *host command-queues* and *device command-queues*. The interaction of these command queues with the memory model are for the most part equivalent. In a few cases, the rules only applies to the host command-queue. We will indicate these special cases by specifically denoting the host command-queue in the memory ordering rule. SVM memory consistency in such instances is implied only with respect to synchronizing host commands.

Memory ordering rules in this section apply to all memory objects (buffers, images and pipes) as well as to SVM allocations where no earlier, and more fine-grained, rules apply.

In the remainder of this section, we assume that each command **C** enqueued onto a command-queue has an associated event object **E** that signals its execution status, regardless of whether **E*was returned to the unit of execution that enqueued *C**. We also distinguish between the API function call that enqueues a command **C** and creates an event **E**, the execution of **C**, and the completion of **C**(which marks the event **E** as complete).

The ordering and synchronization rules for API commands are defined as following:

1. If an API function call **X** enqueues a command **C**, then **X** global-synchronizes-with **C**. For example, a host API function to enqueue a kernel global-synchronizes-with the start of that kernel-instances execution, so that memory updates sequenced-before the enqueue kernel function call will global-happen-before any kernel reads or writes to those same memory locations. For a device-side enqueue, global memory updates sequenced before **X** happens-before **C** reads or writes to those memory locations only in the case of fine-grained SVM.
2. If **E** is an event upon which a command **C** waits, then **E** global-synchronizes-with **C**. In particular, if **C** waits on an event **E** that is tracking the execution status of the command **C1**, then memory operations done by **C1** will global-happen-before memory operations done by **C**. As an example, assume we have an OpenCL program using coarse-grain SVM sharing that enqueues a kernel to a host command-queue to manipulate the contents of a region of a buffer that the host thread then accesses after the kernel completes. To do this, the host thread can call clEnqueueMapBuffer to enqueue a blocking-mode map command to map that buffer region, specifying that the map command must wait on an event signaling the kernels completion. When clEnqueueMapBuffer returns, any memory operations performed by the kernel to that buffer region will global- happen-before subsequent memory operations made by the host thread.
3. If a command **C** has an event **E** that signals its completion, then **C** global- synchronizes-with **E**.
4. For a command **C** enqueued to a host-side command queue, if **C** has an event **E** that signals its completion, then **E** global- synchronizes-with an API call **X** that waits on **E**. For example, if a host thread or kernel-instance calls the wait-for-events function on **E**(e.g. the clWaitForEvents function called from a host thread)**,*then *E** global-synchronizes-with that wait-for-events function call.
5. If commands **C** and **C1** are enqueued in that sequence onto an in-order command-queue, then the event (including the event implied between **C** and **C1*due to the in-order queue) signaling *C*s completion global-synchronizes-with *C1**. Note that in OpenCL 2.0, only a host command-queue can be configured as an in-order queue.
6. If an API call enqueues a marker command **C** with an empty list of events upon which **C** should wait, then the events of all commands enqueued prior to **C** in the command-queue global-synchronize-with*C*.
7. If a host API call enqueues a command-queue barrier command **C** with an empty list of events on which **C** should wait, then the events of all commands enqueued prior to **C** in the command-queue global-synchronize-with **C**. In addition, the event signaling the completion of **C** global-synchronizes-with all commands enqueued after **C** in the command-queue.
8. If a host thread executes a clFinish call **X**, then the events of all commands enqueued prior to **X** in the command-queue global-synchronizes-with **X**.

9. The start of a kernel-instance **K** global-synchronizes-with all operations in the work items of **K**. Note that this includes the execution of any atomic operations by the work items in a program using fine-grain SVM.
10. All operations of all work items of a kernel-instance **K** global-synchronizes-with the event signaling the completion of **K**. Note that this also includes the execution of any atomic operations by the work items in a program using fine-grain SVM.
11. If a callback procedure **P** is registered on an event **E**, then **E** global-synchronizes-with all operations of **P**. Note that callback procedures are only defined for commands within host command-queues.
12. If **C** is a command that waits for an event **E**'s completion, and API function call **X** sets the status of a user event **E**'s status to CL_COMPLETE (for example, from a host thread using a clSetUserEventStatus function), then **X** global-synchronizes-with **C**.
13. If a device enqueues a command **C** with the CLK_ENQUEUE_FLAGS_WAIT_KERNEL flag, then the end state of the parent kernel instance global-synchronizes with **C**.
14. If a work-group enqueues a command **C** with the CLK_ENQUEUE_FLAGS_WAIT_WORK_GROUP flag, then the end state of the work-group global-synchronizes with **C**.

When using an out-of-order command queue, a wait on an event or a marker or command-queue barrier command can be used to ensure the correct ordering of dependent commands. In those cases, the wait for the event or the marker or barrier command will provide the necessary global-synchronizes-with relation.

In this situation:

- access to shared locations or disjoint locations in a single cl_mem object when using atomic operations from different kernel instances enqueued from the host such that one or more of the atomic operations is a write is implementation-defined and correct behavior is not guaranteed except at synchronization points.
- access to shared locations or disjoint locations in a single cl_mem object when using atomic operations from different kernel instances consisting of a parent kernel and any number of child kernels enqueued by that kernel is guaranteed under the memory ordering rules described earlier in this section.
- access to shared locations or disjoint locations in a single program scope global variable, coarse-grained SVM allocation or fine-grained SVM allocation when using atomic operations from different kernel instances enqueued from the host to a single device is guaranteed under the memory ordering rules described earlier in this section.

If fine-grain SVM is used but without support for the OpenCL 2.0 atomic operations, then the host and devices can concurrently read the same memory locations and can concurrently update non-overlapping memory regions, but attempts to update the same memory locations are undefined. Memory consistency is guaranteed at the OpenCL synchronization points without the need for calls to clEnqueueMapBuffer and clEnqueueUnmapMemObject. For fine-grained SVM buffers it is guaranteed that at synchronization points only values written by the kernel will be updated. No writes to fine-grained SVM buffers can be introduced that were not in the original program.

In the remainder of this section, we discuss a few points regarding the ordering rules for commands with a host command queue.

The OpenCL 1.2 standard describes a synchronization point as a kernel-instance or host program location where the contents of memory visible to different work-items or command-queue commands are the same. It also says that waiting on an event and a command-queue barrier are synchronization points between commands in command- queues. Four of the rules listed above (2, 4, 7, and 8) cover these OpenCL synchronization points.

A map operation (clEnqueueMapBuffer or clEnqueueMapImage) performed on a non-SVM buffer or a coarse-grained SVM buffer is allowed to overwrite the entire target region with the latest runtime view of the data as seen by the command with which the map operation synchronizes, whether the values were written by the executing kernels or not.

Any values that were changed within this region by another kernel or host thread while the kernel synchronizing with the map operation was executing may be overwritten by the map operation.

Access to non-SVM cl_mem buffers and coarse-grained SVM allocations is ordered at synchronization points between host commands. In the presence of an out-of-order command queue or a set of command queues mapped to the same device, multiple kernel instances may execute concurrently on the same device.

3.4 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. SPIR-V intermediate language, OpenCL C, OpenCL C++, and OpenCL C language versions from earlier OpenCL specifications are supported by the compiler. Other input languages may be supported by some implementations.

3.4.1 OpenCL Framework: Mixed Version Support

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

The platform version indicates the version of the OpenCL runtime that is supported. This includes all of the APIs that the host can use to interact with resources exposed by the OpenCL runtime; including contexts, memory objects, devices, and command queues.

The device version is an indication of the device's capabilities separate from the runtime and compiler as represented by the device info returned by **clGetDeviceInfo**. Examples of attributes associated with the device version are resource limits (e.g., minimum size of local memory per compute unit) and extended functionality (e.g., list of supported KHR extensions). The version returned corresponds to the highest version of the OpenCL specification for which the device is conformant, but is not higher than the platform version.

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

Backwards compatibility is an important goal for the OpenCL standard. Backwards compatibility is expected such that a device will consume earlier versions of the SPIR-V and OpenCL C programming languages with the following minimum requirements:

1. An OpenCL 1.x device must support at least one 1.x version of the OpenCL C programming language.
2. An OpenCL 2.0 device must support all the requirements of an OpenCL 1.x device in addition to the OpenCL C 2.0 programming language. If multiple language versions are supported, the compiler defaults to using the highest OpenCL 1.x language version supported for the device (typically OpenCL 1.2). To utilize the OpenCL 2.0 Kernel programming language, a programmer must specifically set the appropriate compiler flag (-cl-std=CL2.0). The language version must not be higher than the platform version, but may exceed the device version (see section 5.8.4.5).

3. An OpenCL 2.1 device must support all the requirements of an OpenCL 2.0 device in addition to the SPIR-V intermediate language at version 1.0 or above. Intermediate language versioning is encoded as part of the binary object and no flags are required to be passed to the compiler.
4. An OpenCL 2.2 device must support all the requirements of an OpenCL 2.0 device in addition to the SPIR-V intermediate language at version 1.2 or above. Intermediate language is encoded as a part of the binary object and no flags are required to be passed to the compiler.

Chapter 4

The OpenCL Platform Layer

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

4.1 Querying Platform Info

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

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

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

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

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

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

- CL_INVALID_VALUE if *num_entries* is equal to zero and *platforms* is not NULL or if both *num_platforms* and *platforms* are NULL.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

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

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

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

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

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

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

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

Table 4.1: *OpenCL Platform Queries*

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 specification (functionality defined as part of the core specification and does not require any extensions to be supported). EMBEDDED_PROFILE - if the CL_PLATFORM_VERSION char[] 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 2.2 is described in *section 7*.
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 2.2.
CL_PLATFORM_NAME	char[]	Platform name string.
CL_PLATFORM_VENDOR	char[]	Platform vendor string.

Table 4.1: (continued)

CL_PLATFORM_EXTENSIONS	char[]	Returns a space separated list of extension names (the extension names themselves do not contain any spaces) supported by the platform. Each extension that is supported by all devices associated with this platform must be reported here.
CL_PLATFORM_HOST_TIMER_RESOLUTION	cl_ulong	Returns the resolution of the host timer in nanoseconds as used by **clGetDeviceAndHostTimer**.

clGetPlatformInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors.[2]:

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

4.2 Querying Devices

The list of devices available on a platform can be obtained using the following function.[3].

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

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

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

cl_device_type	Description
CL_DEVICE_TYPE_CPU	An OpenCL device that is the host processor. The host processor runs the OpenCL implementations and is a single or multi-core CPU.
CL_DEVICE_TYPE_GPU	An OpenCL device that is a GPU. By this we mean that the device can also be used to accelerate a 3D API such as OpenGL or DirectX.
CL_DEVICE_TYPE_ACCELERATOR	Dedicated OpenCL accelerators (for example the IBM CELL Blade). These devices communicate with the host processor using a peripheral interconnect such as PCIe.
CL_DEVICE_TYPE_CUSTOM	Dedicated accelerators that do not support programs written in an OpenCL kernel language,

[1] A null terminated string is returned by OpenCL query function calls if the return type of the information being queried is a char[].
[2] The OpenCL specification does not describe the order of precedence for error codes returned by API calls.
[3] **clGetDeviceIDs** may returnal all or a subset of the actual physical devices present in the platform and that maths *device_type*

CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the system. The default device cannot be a **CL_DEVICE_TYPE_CUSTOM** device.
CL_DEVICE_TYPE_ALL	All OpenCL devices available in the system except **CL_DEVICE_TYPE_CUSTOM** devices..

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

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

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

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

- CL_INVALID_PLATFORM if *platform* is not a valid platform.
- CL_INVALID_DEVICE_TYPE if *device_type* is not a valid value.
- CL_INVALID_VALUE if *num_entries* is equal to zero and *devices* is not NULL or if both *num_devices* and *devices* are NULL.
- CL_DEVICE_NOT_FOUND if no OpenCL devices that matched *device_type* were found.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The application can query specific capabilities of the OpenCL device(s) returned by **clGetDeviceIDs**. This can be used by the application to determine which device(s) to use.

The function

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

gets specific information about an OpenCL device.

device may be a device returned by **clGetDeviceIDs** or a sub-device created by **clCreateSubDevices**. If *device* is a sub-device, the specific information for the sub-device will be returned. The information that can be queried using **clGetDeviceInfo** is specified in *table 4.3*.

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

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

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

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

Table 4.2: *OpenCL Device Queries*

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, a combination of the above types or CL_DEVICE_TYPE_CUSTOM.
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 units on the OpenCL device. A work-group executes on a single compute unit. The minimum value is 1.
CL_DEIVCE_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 for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_MAX_ WORK_ITEM_SIZES	size_t []	Maximum number of work-items that can be specified in each dimension of the work-group to **clEnqueueNDRangeKernel**. Returns n size_t entries, where n is the value returned by the query for CL_DEVICE_MAX_WORK_ITEM_DIMEN SIONS. The minimum value is (1, 1, 1) for devices that are not of type CL_DEVICE_TYPE_CUSTOM .
CL_DEVICE_MAX_ WORK_GROUP_SIZE	size_t	Maximum number of work-items in a work-group that a device is capable of executing on a single compute unit, for any given kernel-instance running on the device. (Refer also to **clEnqueueNDRangeKernel and CL_KERNEL_WORK_GROUP_SIZE**). The minimum value is 1.

Table 4.2: (continued)

CL_DEVICE_PREFERRED_ VECTOR_WIDTH_CHAR CL_DEVICE_PREFERRED_ VECTOR_WIDTH_SHORT CL_DEVICE_PREFERRED_ VECTOR_WIDTH_INT CL_DEVICE_PREFERRED_ VECTOR_WIDTH_LONG CL_DEVICE_PREFERRED_ VECTOR_WIDTH_FLOAT CL_DEVICE_PREFERRED_ VECTOR_WIDTH_DOUBLE CL_DEVICE_PREFERRED_ VECTOR_WIDTH_HALF	cl_uint	Preferred native vector width size for built- in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. If double precision is not sup-ported,CL_DEVICE_PREFERRED_VECTOR_WIDTH_ DOUBLE must return 0. If the **cl_khr_fp16** extension is not supported, CL_DEVICE_PREFERRED_VECTOR_WIDTH_ HALF must return 0.
CL_DEVICE_NATIVE_ VECTOR_WIDTH_CHAR CL_DEVICE_NATIVE_ VECTOR_WIDTH_SHORT CL_DEVICE_NATIVE_ VECTOR_WIDTH_INT CL_DEVICE_NATIVE_ VECTOR_WIDTH_LONG CL_DEVICE_NATIVE_ VECTOR_WIDTH_FLOAT CL_DEVICE_NATIVE_ VECTOR_WIDTH_DOUBLE CL_DEVICE_NATIVE_ VECTOR_WIDTH_HALF	cl_uint	Returns the native ISA vector width. The vector width is defined as the number of scalar elements that can be stored in the vector. If double precision is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE must return 0. If the **cl_khr_fp16** extension is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_ HALF must return 0.
CL_DEVICE_MAX_ CLOCK_FREQUENCY	cl_uint	Clock frequency of the device in MHz. The meaning of this value is implementation- defined. For devices with multiple clock domains, the clock frequency for any of the clock domains may be returned. For devices that dynamically change frequency for power or thermal reasons, the returned clock frequency may be any valid frequency.
CL_DEVICE_ADDRESS_BITS	cl_uint	The default compute device address space size of the global address space specified as an unsigned integer value in bits. Currently supported values are 32 or 64 bits.

Table 4.2: (continued)

CL_DEVICE_MAX_ MEM_ALLOC_SIZE	cl_ulong	Max size of memory object allocation in bytes. The minimum value is max (min(1024*1024*1024, 1/4th of **CL_DEVICE_GLOBAL_MEM_SIZE**), 32*1024*1024) for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
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 image objects arguments of a kernel declared with the read_only qualifier. The minimum value is 128 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_ WRITE_IMAGE_ARGS	cl_uint	Max number of image objects arguments of a kernel declared with the write_only qualifier. The minimum value is 64 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_ READ_WRITE_IMAGE_ARGS[4]	cl_uint	Max number of image objects arguments of a kernel declared with the write_only or read_write qualifier. The minimum value is 64 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE .
CL_DEVICE_IL_VERSION	char[]	The intermediate languages that can be supported by **clCreateProgramWithIL** for this device. Returns a space-separated list of IL version strings of the form <IL_Prefix>_<Major_Version>.<Minor_Version>. For OpenCL 2.2, SPIR-V is a required IL prefix.
CL_DEVICE_IMAGE2D_ MAX_WIDTH	size_t	Max width of 2D image or 1D image not created from a buffer object in pixels. The minimum value is 16384 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 16384 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_IMAGE_ MAX_BUFFER_SIZE	size_t	Max number of pixels for a 1D image created from a buffer object. The minimum value is 65536 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE .

Table 4.2: (continued)

CL_DEVICE_IMAGE_ MAX_ARRAY_SIZE	size_t	Max number of images in a 1D or 2D image array. 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. The minimum value is 16 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE .
CL_DEVICE_IMAGE_ PITCH_ALIGNMENT	cl_uint	The row pitch alignment size in pixels for 2D images created from a buffer. The value returned must be a power of 2. If the device does not support images, this value must be 0.
CL_DEVICE_IMAGE_ BASE_ADDRESS_ ALIGNMENT	cl_uint	This query should be used when a 2D image is created from a buffer which was created using CL_MEM_USE_HOST_PTR. The value returned must be a power of 2. This query specifies the minimum alignment in pixels of the host_ptr specified to **clCreateBuffer**. If the device does not support images, this value must be 0.
CL_DEVICE_MAX_ PIPE_ARGS	cl_uint	The maximum number of pipe objects that can be passed as arguments to a kernel. The minimum value is 16.
CL_DEVICE_PIPE_ MAX_ACTIVE_RESERVATIONS	cl_uint	The maximum number of reservations that can be active for a pipe per work-item in a kernel. A work-group reservation is counted as one reservation per work-item. The minimum value is 1.
CL_DEVICE_PIPE_ MAX_PACKET_SIZE	cl_uint	The maximum size of pipe packet in bytes. The minimum value is 1024 bytes.
CL_DEVICE_MAX_ PARAMETER_SIZE	size_t	Max size in bytes of all arguments that can be passed to a kernel. The minimum value is 1024 for devices that are not of type CL_DEVICE_TYPE_CUSTOM . For this minimum value, only a maximum of 128 arguments can be passed to a kernel
CL_DEVICE_MEM_ BASE_ADDR_ALIGN	cl_uint	Alignment requirement (in bits) for sub-buffer offsets. The minimum value is the size (in bits) of the largest OpenCL built-in data type supported by the device (long16 in FULL profile, long16 or int16 in EMBEDDED profile) for devices that are not of type CL_DEVICE_TYPE_CUSTOM.

Table 4.2: (continued)

CL_DEVICE_SINGLE_ FP_CONFIG[5]	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 positive and negative infinity rounding modes supported CL_FP_FMA – IEEE754-2008 fused multiply- add is supported. CL_FP_CORRECTLY_ROUNDED_DIVIDE _SQRT – divide and sqrt are correctly rounded as defined by the IEEE754 specification. CL_FP_SOFT_FLOAT – Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. For the full profile, the mandated minimum floating-point capability for devices that are not of type CL_DEVICE_TYPE_CUSTOM is: CL_FP_ROUND_TO_NEAREST | CL_FP_INF_NAN. For the embedded profile, see section 10.

Table 4.2: (continued)

CL_DEVICE_DOUBLE_ FP_CONFIG[6]	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 positive and negative infinity rounding modes supported. CP_FP_FMA – IEEE754-2008 fused multiply-add is supported. CL_FP_SOFT_FLOAT – Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. Double precision is an optional feature so the mandated minimum double precision floating-point capability is 0. If double precision is supported by the device, then the minimum double precision floating-point capability must be: CL_FP_FMA | CL_FP_ROUND_TO_NEAREST | CL_FP_INF_NAN | CL_FP_DENORM .
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 for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_MAX_ CONSTANT_ARGS	cl_uint	Max number of arguments declared with the __constant qualifier in a kernel. The minimum value is 8 for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_MAX_ GLOBAL_VARIABLE_SIZE	size_t	The maximum number of bytes of storage that may be allocated for any single variable in program scope or inside a function in an OpenCL kernel language declared in the global address space. The minimum value is 64 KB.

Table 4.2: (continued)

CL_DEVICE_GLOBAL_ VARIABLE_PREFERRED_ TOTAL_SIZE	size_t	Maximum preferred total size, in bytes, of all program variables in the global address space. This is a performance hint. An implementation may place such variables in storage with optimized device access. This query returns the capacity of such storage. The minimum value is 0.
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 . For custom devices, CL_NONE can also be returned indicating no local memory support.
CL_DEVICE_LOCAL_ MEM_SIZE	cl_ulong	Size of local memory region in bytes. The minimum value is 32 KB for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_ERROR_ CORRECTION_SUPPORT	cl_bool	Is CL_TRUE if the device implements error correction for all accesses to compute device memory (global and constant). Is CL_FALSE if the device does not implement such error correction.
CL_DEVICE_PROFILING_ TIMER_RESOLUTION	size_t	Describes the resolution of device timer. This is measured in nanoseconds. Refer to *section 5.14* 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 otherwise. A device is considered to be available if the device can be expected to successfully execute commands enqueued to the device.
CL_DEVICE_COMPILER_ AVAILABLE	cl_bool	Is CL_FALSE if the implementation does not have a compiler available to compile the program source. Is CL_TRUE if the compiler is available. This can be CL_FALSE for the embedded platform profile only.
CL_DEVICE_LINKER_ AVAILABLE	cl_bool	Is CL_FALSE if the implementation does not have a linker available. Is CL_TRUE if the linker is available. This can be CL_FALSE for the embedded platform profile only. This must be CL_TRUE if CL_DEVICE_COMPILER_AVAILABLE is CL_TRUE .

Table 4.2: (continued)

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_ ON_HOST_PROPERTIES[7]	cl_command_ queue_properties	Describes the on host command-queue properties supported by the device. This is a bit-field that describes one or more of the following values: CL_QUEUE_OUT_OF_ORDER_EXEC_ MODE_ENABLE CL_QUEUE_PROFILING_ENABLE These properties are described in table 5.1. The mandated minimum capability is: CL_QUEUE_PROFILING_ENABLE .
CL_DEVICE_QUEUE_ ON_DEVICE_PROPERTIES	cl_command_ queue_properties	Describes the on device command-queue properties supported by the device. This is a bit-field that describes one or more of the following values: CL_QUEUE_OUT_OF_ORDER_EXEC_ MODE_ENABLE CL_QUEUE_PROFILING_ENABLE These properties are described in table 5.1. The mandated minimum capability is: CL_QUEUE_OUT_OF_ORDER_EXEC_ MODE_ENABLE | CL_QUEUE_PROFILING_ENABLE .
CL_DEVICE_QUEUE_ ON_DEVICE_PREFERRED_ SIZE	cl_uint	The size of the device queue in bytes preferred by the implementation. Applications should use this size for the device queue to ensure good performance. The minimum value is 16 KB
CL_DEVICE_QUEUE_ ON_DEVICE_MAX_SIZE	cl_uint	The max. size of the device queue in bytes. The minimum value is 256 KB for the full profile and 64 KB for the embedded profile
CL_DEVICE_MAX_ ON_DEVICE_QUEUES	cl_uint	The maximum number of device queues that can be created for this device in a single context. The minimum value is 1.

Table 4.2: (continued)

CL_DEVICE_MAX_ ON_DEVICE_EVENTS	cl_uint	The maximum number of events in use by a device queue. These refer to events returned by the enqueue_ built-in functions to a device queue or user events returned by the create_user_event built-in function that have not been released. The minimum value is 1024.
CL_DEVICE_BUILT_IN_ KERNELS	char[]	A semi-colon separated list of built-in kernels supported by the device. An empty string is returned if no built-in kernels are supported by the device.
CL_DEVICE_PLATFORM	cl_platform_id	The platform associated with this device.
CL_DEVICE_NAME	char[]	Device name string.
CL_DEVICE_VENDOR	char[]	Vendor name string.
CL_DRIVER_VERSION	char[]	OpenCL software driver version string. Follows a vendor-specific format.
CL_DEVICE_PROFILE[8]	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_v ersion><space><vendor-specific information>* The major_version.minor_version value returned will be 2.2.

Table 4.2: (continued)

CL_DEVICE_OPENCL_C_ VERSION	char[]	OpenCL C version string. Returns the highest OpenCL C version supported by the compiler for this device that is not of type CL_DEVICE_TYPE_CUSTOM . This version string has the following format: *OpenCL<space>C<space><major_versio n.minor_version><space><vendor- specific information>* The major_version.minor_version value returned must be 2.0 if CL_DEVICE_VERSION is OpenCL 2.0. The major_version.minor_version value returned must be 1.2 if CL_DEVICE_VERSION is OpenCL 1.2. The major_version.minor_version value returned must be 1.1 if CL_DEVICE_VERSION is OpenCL 1.1. The major_version.minor_version value returned can be 1.0 or 1.1 if CL_DEVICE_VERSION is OpenCL 1.0.
CL_DEVICE_EXTENSIONS	char[]	Returns a space separated list of extension names (the extension names themselves do not contain any spaces) supported by the device. The list of extension names returned can be vendor supported extension names and one or more of the following Khronos approved extension names: **cl_khr_int64_base_atomics cl_khr_int64_extended_atomics cl_khr_fp16 cl_khr_gl_sharing cl_khr_gl_event cl_khr_d3d10_sharing cl_khr_dx9_media_sharing cl_khr_d3d11_sharing cl_khr_gl_depth_images cl_khr_gl_msaa_sharing cl_khr_initialize_memory cl_khr_terminate_context cl_khr_spir cl_khr_srgb_image_writes** **The following approved Khronos extension names must be returned by all devices that support OpenCL C 2.0:** **cl_khr_byte_addressable_store cl_khr_fp64 (for backward compatibility if double precision is supported) cl_khr_3d_image_writes cl_khr_image2d_from_buffer cl_khr_depth_images** Please refer to the OpenCL 2.0 Extension Specification for a detailed description of these extensions.
CL_DEVICE_PRINTF_ BUFFER_SIZE	size_t	Maximum size in bytes of the internal buffer that holds the output of printf calls from a kernel. The minimum value for the FULL profile is 1 MB.

Table 4.2: (continued)

CL_DEVICE_PREFERRED_INTEROP_USER_SYNC	cl_bool	Is CL_TRUE if the devices preference is for the user to be responsible for synchronization, when sharing memory objects between OpenCL and other APIs such as DirectX, CL_FALSE if the device / implementation has a performant path for performing synchronization of memory object shared between OpenCL and other APIs such as DirectX.
CL_DEVICE_PARENT_DEVICE	cl_device_id	Returns the cl_device_id of the parent device to which this sub-device belongs. If *device* is a root-level device, a NULL value is returned.
CL_DEVICE_PARTITION_MAX_SUB_DEVICES	cl_uint	Returns the maximum number of sub- devices that can be created when a device is partitioned. The value returned cannot exceed CL_DEVICE_MAX_COMPUTE_UNITS .
CL_DEVICE_PARTITION_PROPERTIES	cl_device_partition_property[]	Returns the list of partition types supported by *device*. The is an array of cl_device_partition_property values drawn from the following list: CL_DEVICE_PARTITION_EQUALLY CL_DEVICE_PARTITION_BY_COUNTS CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN If the device cannot be partitioned (i.e. there is no partitioning scheme supported by the device that will return at least two subdevices), a value of 0 will be returned.
CL_DEVICE_PARTITION_AFFINITY_DOMAIN	cl_device_affinity_domain	Returns the list of supported affinity domains for partitioning the device using CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN . This is a bit-field that describes one or more of the following values: CL_DEVICE_AFFINITY_DOMAIN_NUMA CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE If the device does not support any affinity domains, a value of 0 will be returned.

Table 4.2: (continued)

CL_DEVICE_PARTITION_TYPE	cl_device_partition_property[]	Returns the properties argument specified in **clCreateSubDevices** if device is a sub- device. In the case where the properties argument to **clCreateSubDevices** is CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN , CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE , the affinity domain used to perform the partition will be returned. This can be one of the following values: CL_DEVICE_AFFINITY_DOMAIN_NUMA CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE Otherwise the implementation may either return a *param_value_size_ret* of 0 i.e. there is no partition type associated with device or can return a property value of 0 (where 0 is used to terminate the partition property list) in the memory that *param_value* points to.
CL_DEVICE_REFERENCE_COUNT	cl_uint	Returns the *device* reference count. If the device is a root-level device, a reference count of one is returned.

Table 4.2: (continued)

CL_DEVICE_SVM_ CAPABILITIES	cl_device_svm_ capabilities	Describes the various shared virtual memory (a.k.a. SVM) memory allocation types the device supports. Coarse-grain SVM allocations are required to be supported by all OpenCL 2.0 devices. This is a bit-field that describes a combination of the following values: CL_DEVICE_SVM_COARSE_GRAIN_ BUFFER – Support for coarse-grain buffer sharing using **clSVMAlloc**. Memory consistency is guaranteed at synchronization points and the host must use calls to **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject**. CL_DEVICE_SVM_FINE_GRAIN_BUFFER – Support for fine-grain buffer sharing using **clSVMAlloc**. Memory consistency is guaranteed at synchronization points without need for **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject**. CL_DEVICE_SVM_FINE_GRAIN_SYSTEM – Support for sharing the host's entire virtual memory including memory allocated using **malloc**. Memory consistency is guaranteed at synchronization points. CL_DEVICE_SVM_ATOMICS – Support for the OpenCL 2.0 atomic operations that provide memory consistency across the host and all OpenCL devices supporting fine-grain SVM allocations. The mandated minimum capability is CL_DEVICE_SVM_COARSE_GRAIN_BUFFER.
CL_DEVICE_PREFERRED_ PLATFORM_ATOMIC_ ALIGNMENT	cl_uint	Returns the value representing the preferred alignment in bytes for OpenCL 2.0 fine-grained SVM atomic types. This query can return 0 which indicates that the preferred alignment is aligned to the natural size of the type.
CL_DEVICE_PREFERRED_ GLOBAL_ATOMIC_ ALIGNMENT	cl_uint	Returns the value representing the preferred alignment in bytes for OpenCL 2.0 atomic types to global memory. This query can return 0 which indicates that the preferred alignment is aligned to the natural size of the type.
CL_DEVICE_PREFERRED_ LOCAL_ATOMIC_ ALIGNMENT	cl_uint	Returns the value representing the preferred alignment in bytes for OpenCL 2.0 atomic types to local memory. This query can return 0 which indicates that the preferred alignment is aligned to the natural size of the type.
CL_DEVICE_MAX_ NUM_SUB_GROUPS	cl_uint	Maximum number of sub-groups in a work-group that a device is capable of executing on a single compute unit, for any given kernel-instance running on the device. The minimum value is 1. (Refer also to **clGetKernelSubGroupInfo**.)

Table 4.2: (continued)

CL_DEVICE_SUB_ GROUP_INDEPENDENT_ FORWARD_PROGRESS	cl_bool	Is CL_TRUE if this device supports independent forward progress of sub-groups, CL_FALSE otherwise. If cl_khr_subgroups is supported by the device this must return CL_TRUE.

The device queries described in *table 4.3* should return the same information for a root-level device i.e. a device returned by **clGetDeviceIDs** and any sub-devices created from this device except for the following queries:

```
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE
CL_DEVICE_BUILT_IN_KERNELS
CL_DEVICE_PARENT_DEVICE
CL_DEVICE_PARTITION_TYPE
CL_DEVICE_REFERENCE_COUNT
```

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

- CL_INVALID_DEVICE if *device* is not valid.
- CL_INVALID_VALUE if *param_name* is not one of the supported values or if size in bytes specified by *param_value_size_is < size of return type as specified in _table 4.3* and *param_value* is not a NULL value or if *param_name* is a value that is available as an extension and the corresponding extension is not supported by the device.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clGetDeviceAndHostTimer(cl_device_id device,
                               cl_ulong* device_timestamp,
                               cl_ulong* host_timestamp)
```

Returns a reasonably synchronized pair of timestamps from the device timer and the host timer as seen by *device*. Implementations may need to execute this query with a high latency in order to provide reasonable synchronization of the timestamps. The host timestamp and device timestamp returned by this function and **clGetHostTimer** each have an implementation defined timebase. The timestamps will always be in their respective timebases regardless of which query function is used. The timestamp returned from **clGetEventProfilingInfo** for an event on a device and a device timestamp queried from the same device will always be in the same timebase.

[4] NOTE: **CL_DEVICE_MAX_WRITE_IMAGE_ARGS** is only there for backward compatibility. **CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS** should be used instead.

[5] The optional rounding modes should be included as a device capability only if it is supported natively. All explicit conversion functions with specific rounding modes must still operate correctly.

[6] The optional rounding modes should be included as a device capability only if it is supported natively. All explicit conversion functions with specific rounding modes must still operate correctly.

[7] CL_DEVICE_QUEUE_PROPERTIES is deprecated and replaced by CL_DEVICE_QUEUE_ON_HOST_PROPERTIES.

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

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

device_timestamp will be updated with the value of the device timer in nanoseconds. The resolution of the timer is the same as the device profiling timer returned by *clGetDeviceInfo*and the CL_DEVICE_PROFILING_TIMER_RESOLUTION query.

host_timestamp will be updated with the value of the host timer in nanoseconds at the closest possible point in time to that at which *device_timer* was returned. The resolution of the timer may be queried via **clGetPlatformInfo** and the flag CL_PLATFORM_HOST_TIMER_RESOLUTION.

clGetDeviceAndHostTimer will return CL_SUCCESS with a time value in *host_timestamp* if provided. Otherwise, it returns one of the following errors:

- CL_INVALID_DEVICE if *device* is not a valid OpenCL device.
- CL_INVALID_VALUE if *host_timestamp* or _device_timestamp_is NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clGetHostTimer(cl_device_id device,
                      cl_ulong* host_timestamp)
```

Return the current value of the host clock as seen by *device*. This value is in the same timebase as the host_timestamp returned from **clGetDeviceAndHostTimer**. The implementation will return with as low a latency as possible to allow a correlation with a subsequent application sampled time. The host timestamp and device timestamp returned by this function and **clGetDeviceAndHostTimer** each have an implementation defined timebase. The timestamps will always be in their respective timebases regardless of which query function is used. The timestamp returned from **clGetEventProfilingInfo** for an event on a device and a device timestamp queried from the same device will always be in the same timebase.

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

host_timestamp will be updated with the value of the current timer in nanoseconds. The resolution of the timer may be queried via **clGetPlatformInfo** and the flag CL_PLATFORM_HOST_TIMER_RESOLUTION.

clGetHostTimer will return CL_SUCCESS with a time value in *host_timestamp* if provided. Otherwise, it returns one of the following errors:

- CL_INVALID_DEVICE if *device* is not a valid OpenCL device.
- CL_INVALID_VALUE if _host_timestamp_is NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

4.3 Partitioning a Device

The function

```
cl_int clCreateSubDevices(cl_device_id in_device,
                          const cl_device_partition_property *properties,
                          cl_uint num_devices,
                          cl_device_id *out_devices,
                          cl_uint *num_devices_ret)
```

creates an array of sub-devices that each reference a non-intersecting set of compute units within in_device, according to a partition scheme given by *properties*. The output sub-devices may be used in every way that the root (or parent) device can be used, including creating contexts, building programs, further calls to **clCreateSubDevices** and creating command-queues. When a command-queue is created against a sub-device, the commands enqueued on the queue are executed only on the sub-device.

in_device is the device to be partitioned.

properties specifies how *in_device* is to be partition described by a partition name and its corresponding value. Each partition name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported partitioning schemes is described in *table 4.4*. Only one of the listed partitioning schemes can be specified in *properties*.

Table 4.3: *List of supported partition schemes by* **clCreateSubDevices**

cl_device_partition_property enum	**Partition value**	**Description**
CL_DEVICE_PARTITION_ EQUALLY	cl_uint	Split the aggregate device into as many smaller aggregate devices as can be created, each containing *n* compute units. The value *n* is passed as the value accompanying this property. If *n* does not divide evenly into CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS, then the remaining compute units are not used.
CL_DEVICE_PARTITION_ BY_COUNTS	cl_uint	This property is followed by a CL_DEVICE_PARTITION_BY_COUNTS_LIST_END terminated list of compute unit counts. For each non-zero count *m* in the list, a sub-device is created with *m* compute units in it. CL_DEVICE_PARTITION_BY_COUNTS_LIST_END is defined to be 0. The number of non-zero count entries in the list may not exceed CL_DEVICE_PARTITION_MAX_SUB_DEVICES. The total number of compute units specified may not exceed CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS .

Table 4.3: (continued)

CL_DEVICE_PARTITION_ BY_AFFINITY_DOMAIN	cl_device_affinity_ domain	Split the device into smaller aggregate devices containing one or more compute units that all share part of a cache hierarchy. The value accompanying this property may be drawn from the following list: CL_DEVICE_AFFINITY_DOMAIN_NUMA – Split the device into sub-devices comprised of compute units that share a NUMA node. CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE – Split the device into sub-devices comprised of compute units that share a level 4 data cache. CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE – Split the device into sub-devices comprised of compute units that share a level 3 data cache. CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE – Split the device into sub-devices comprised of compute units that share a level 2 data cache. CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE – Split the device into sub-devices comprised of compute units that share a level 1 data cache. CL_DEVICE_AFFINITY_ DOMAIN_NEXT_ PARTITIO NABLE – Split the device along the next partitionable affinity domain. The implementation shall find the first level along which the device or sub-device may be further subdivided in the order NUMA, L4, L3, L2, L1, and partition the device into sub-devices comprised of compute units that share memory subsystems at this level. The user may determine what happened by calling clGetDeviceInfo(CL_DEVICE_PARTITION_TYPE) on the sub-devices.

num_devices is the size of memory pointed to by *out_devices* specified as the number of cl_device_id entries.

out_devices is the buffer where the OpenCL sub-devices will be returned. If *out_devices_is NULL, this argument is ignored. If _out_devices* is not NULL, *num_devices* must be greater than or equal to the number of sub-devices that *device* may be partitioned into according to the partitioning scheme specified in *properties*.

num_devices_ret returns the number of sub-devices that *device* may be partitioned into according to the partitioning scheme specified in *properties*. If *num_devices_ret* is NULL, it is ignored.

clCreateSubDevices returns CL_SUCCESS if the partition is created successfully. Otherwise, it returns a NULL value with the following error values returned in *errcode_ret*:

- CL_INVALID_DEVICE if *in_device* is not valid.
- CL_INVALID_VALUE if values specified in *properties* are not valid or if values specified in *properties* are valid but not supported by the device.

- CL_INVALID_VALUE if *out_devices* is not NULL and *num_devices* is less than the number of sub-devices created by the partition scheme.
- CL_DEVICE_PARTITION_FAILED if the partition name is supported by the implementation but in_device could not be further partitioned.
- CL_INVALID_DEVICE_PARTITION_COUNT if the partition name specified in *properties* is CL_DEVICE_PARTITION_BY_COUNTS and the number of sub-devices requested exceeds CL_DEVICE_PARTITION_MAX_SUB_DEVICES or the total number of compute units requested exceeds CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS for *in_device*, or the number of compute units requested for one or more sub-devices is less than zero or the number of sub-devices requested exceeds CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS for *in_device*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

A few examples that describe how to specify partition properties in *properties* argument to **clCreateSubDevices** are given below:

To partition a device containing 16 compute units into two sub-devices, each containing 8 compute units, pass the following in *properties*:

```
{ CL_DEVICE_PARTITION_EQUALLY, 8, 0 }
```

To partition a device with four compute units into two sub-devices with one sub-device containing 3 compute units and the other sub-device 1 compute unit, pass the following in properties argument:

```
{ CL_DEVICE_PARTITION_BY_COUNTS,
  3, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0 }
```

To split a device along the outermost cache line (if any), pass the following in properties argument:

```
{ CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN,
  CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE,
  0 }
```

The function

```
cl_int clRetainDevice(cl_device_id device)
```

increments the *device* reference count if *device* is a valid sub-device created by a call to **clCreateSubDevices**. If *device* is a root level device i.e. a cl_device_id returned by **clGetDeviceIDs**, the *device* reference count remains unchanged. **clRetainDevice** returns CL_SUCCESS if the function is executed successfully or the device is a root-level device. Otherwise, it returns one of the following errors:

- CL_INVALID_DEVICE if *device* is not a valid sub-device created by a call to **clCreateSubDevices**.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clReleaseDevice(cl_device_id device)
```

decrements the *device* reference count if device is a valid sub-device created by a call to **clCreateSubDevices**. If *device* is a root level device i.e. a cl_device_id returned by **clGetDeviceIDs**, the *device* reference count remains unchanged. **clReleaseDevice** returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_DEVICE if *device* is not a valid sub-device created by a call to **clCreateSubDevices**.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

After the *device* reference count becomes zero and all the objects attached to *device* (such as command-queues) are released, the *device* object is deleted. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainDevice** causes undefined behavior.

4.4 Contexts

The function

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

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

properties_specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties is described in _table 4.5. properties can be NULL in which case the platform that is selected is implementation-defined.

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

cl_context_properties enum	Property value	Description
CL_CONTEXT_PLATFORM	cl_platform_id	Specifies the platform to use.
CL_CONTEXT_INTEROP_ USER_SYNC	cl_bool	Specifies whether the user is responsible for synchronization between OpenCL and other APIs. Please refer to the specific sections in the OpenCL 2.0 extension specification that describe sharing with other APIs for restrictions on using this flag. If CL_CONTEXT_INTEROP_USER_ SYNC is not specified, a default of CL_FALSE is assumed.

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

devices is a pointer to a list of unique deviceslink[9] returned by **clGetDeviceIDs** or sub-devices created by **clCreateSubDevices** for a platform.

pfn_notify is a callback function that can be registered by the application. This callback function will be used by the OpenCL implementation to report information on errors during context creation as well as errors that occur at runtime in this context. This callback function may be called asynchronously by the OpenCL implementation. It is the applications 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.

Note

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

---

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

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

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

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

- CL_INVALID_PROPERTY if context property name in *properties* is not a supported property name, if the value specified for a supported property name is not valid, or if the same property name is specified more than once.
- CL_INVALID_VALUE if _devices_is NULL.

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

- CL_INVALID_VALUE if _num_devices_is equal to zero.
- CL_INVALID_VALUE if *pfn_notify* is NULL but *user_data* is not NULL.

- CL_INVALID_DEVICE if *devices* contains an invalid device.

- CL_DEVICE_NOT_AVAILABLE if a device in *devices* is currently not available even though the device was returned by **clGetDeviceIDs**.

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

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

The function[10]

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

creates an OpenCL context from a device type that identifies the specific device(s) to use. Only devices that are returned by **clGetDeviceIDs** for *device_type* are used to create the context. The context does not reference any sub-devices that may have been created from these devices.

properties_specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list of supported properties is described in _table 4.5. properties can also be NULL in which case the platform that is selected is implementation-defined.

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

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

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

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

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

- CL_INVALID_PLATFORM if *properties_is NULL and no platform could be selected or if platform value specified in _properties* is not a valid platform.
- CL_INVALID_PROPERTY if context property name in *properties* is not a supported property name, if the value specified for a supported property name is not valid, or if the same property name is specified more than once.
- CL_INVALID_VALUE if *pfn_notify* is NULL but *user_data* is not NULL.
- CL_INVALID_DEVICE_TYPE if *device_type* is not a valid value.
- CL_DEVICE_NOT_AVAILABLE if no devices that match *device_type* and property values specified in *properties* are currently available.
- CL_DEVICE_NOT_FOUND if no devices that match *device_type* and property values specified in *properties* were found.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clRetainContext(cl_context context)
```

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

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

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

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

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

The function

```
cl_int clReleaseContext(cl_context context)
```

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

- CL_INVALID_CONTEXT if *context* is not a valid OpenCL context.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

After the *context* reference count becomes zero and all the objects attached to *context* (such as memory objects, command-queues) are released, the *context* is deleted. Using this function to release a reference that was not obtained by creating the object or by calling *clRetainContext*causes undefined behavior. The function

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

can be used to query information about a context.

context specifies the OpenCL context being queried.

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

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

param_value_size specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of return type as described in *table 4.6*.

param_value_size_ret returns the actual size in bytes of data being queried by *param_name*. 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 4.6*.

List of supported param_names by **clGetContextInfo**

cl_context_info	**Return Type**	**Information returned in param_value**
CL_CONTEXT_ REFERENCE_COUNT [11]	cl_uint	Return the *context* reference count.
CL_CONTEXT_NUM_DEVICES	cl_uint	Return the number of devices in *context*.
CL_CONTEXT_DEVICES	cl_device_id[]	Return the list of devices and sub-devices in *context*.

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

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

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if *param_name* is not one of the supported values or if size in bytes specified by *param_value_size_is < size of return type as specified in _table 4.6* and *param_value* is not a NULL value.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

[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.

Chapter 5

The OpenCL Runtime

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

5.1 Command Queues

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

The function

```
cl_command_queue clCreateCommandQueueWithProperties(
    cl_context context,
    cl_device_id device,
    const cl_queue_properties *properties,
    cl_int *errcode_ret)
```

creates a host or device command-queue on a specific device.

context must be a valid OpenCL context.

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

properties specifies a list of properties for the command-queue and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties is described in the table below_._ If a supported property and its value is not specified in *properties*, its default value will be used. *properties* can be NULL in which case the default values for supported command-queue properties will be used.

Table 5.1: *List of supported cl_queue_properties values and description*

Table 5.1: (continued)

Queue Properties	Property Value	Description
CL_QUEUE_PROPERTIES	cl_bitfield	This is a bitfield and can be set to a 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. 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. CL_QUEUE_ON_DEVICE – Indicates that this is a device queue. If CL_QUEUE_ON_DEVICE is set, CL_QUEUE_OUT_OF_ORDER_ EXEC_MODE_ENABLE[1]: must also be set. CL_QUEUE_ ON_DEVICE_DEFAULT[2]:–indicates that this is the default device queue. This can only be used with CL_QUEUE_ON_DEVICE. If CL_QUEUE_PROPERTIES is not specified an in-order host command queue is created for the specified device

Table 5.1: (continued)

CL_QUEUE_SIZE	cl_uint	Specifies the size of the device queue in bytes. This can only be specified if CL_QUEUE_ON_DEVICE is set in CL_QUEUE_PROPERTIES. This must be a value ⇐ CL_DEVICE_QUEUE_ ON_DEVICE_MAX_SIZE. For best performance, this should be ⇐ CL_DEVICE_QUEUE_ ON_DEVICE_PREFERRED_SIZE. If CL_QUEUE_SIZE is not specified, the device queue is created with CL_DEVICE_QUEUE_ ON_DEVICE_PREFERRED_SIZE as the size of the queue.

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

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_INVALID_DEVICE if *device_is not a valid device or is not associated with _context*.
- CL_INVALID_VALUE if values specified in *properties* are not valid.
- CL_INVALID_QUEUE_PROPERTIES if values specified in *properties* are valid but are not supported by the device.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clSetDefaultDeviceCommandQueue(cl_context context,
                                      cl_device_id device,
                                      cl_command_queue command_queue)
```

replaces the default command queue on the *device*.

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

[1] Only out-of-order device queues are supported.

[2] The application must create the default device queue if any kernels containing calls to get_default_queue are enqueued. There can only be one default device queue for each device within a context. clCreateCommandQueueWithProperties with CL_QUEUE_PROPERTIES set to CL_QUEUE_ON_DEVICE or CL_QUEUE_ON_DEVICE_DEFAULT will return the default device queue that has already been created and increment its retain count by 1.

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_DEVICE if *device* is not a valid device or is not associated with *context*.
- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue for *device*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

clSetDefaultDeviceCommandQueue may be used to replace a default device command queue created with **clCreateCommandQueueWithProperties** and the CL_QUEUE_ON_DEVICE_DEFAULT flag.

The function

```
cl_int clRetainCommandQueue(cl_command_queue command_queue)
```

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

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

The function

```
cl_int clReleaseCommandQueue(cl_command_queue command_queue)
```

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

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

clReleaseCommandQueue performs an implicit flush to issue any previously queued OpenCL commands in *command_queue*. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainCommandQueue** causes undefined behavior.

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_name*. 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*.

Table 5.2: *List of supported param_names by clGetCommandQueueInfo*

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[3]	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 value associated with the CL_COMMAND_QUEUE_ PROPERTIES passed in *properties* argument in **clCreateCommandQueueWithProperties.**
CL_QUEUE_SIZE	cl_uint	Return the currently specified size for the device command-queue. This query is only supported for device command queues.
CL_QUEUE_DEVICE_DEFAULT	cl_command_queue	Return the current default command queue for the underlying device.

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

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

[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.

NOTE

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

5.2 Buffer Objects

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

5.2.1 Creating Buffer Objects

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

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

context is a valid OpenCL context used to create the buffer object.

flags is a bit-field that is used to specify allocation and usage information such as the memory arena that should be used to allocate the buffer object and how it will be used. *Table 5.3* describes the possible values for *flags*. If value specified for *flags* is 0, the default is used which is CL_MEM_READ_WRITE.

Table 5.3: *List of supported cl_mem_flags values*

cl_mem_flags	Description
CL_MEM_READ_WRITE	This flag specifies that the memory object will be read and written by a kernel. This is the default.
CL_MEM_WRITE_ONLY	This flag specifies that the memory object will be written but not read by a kernel. Reading from a buffer or image object created with CL_MEM_WRITE_ONLY inside a kernel is undefined. CL_MEM_READ_WRITE and CL_MEM_WRITE_ONLY are mutually exclusive.
CL_MEM_READ_ONLY	This flag specifies that the memory object is a readonly 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_READ_WRITE or CL_MEM_WRITE_ONLY and CL_MEM_READ_ONLY are mutually exclusive.

Table 5.3: (continued)

CL_MEM_USE_HOST_PTR	This flag is valid only if host_ptr is not NULL. If specified, it indicates that the application wants the OpenCL implementation to use memory referenced by host_ptr as the storage bits for the memory object. OpenCL implementations are allowed to cache the buffer contents pointed to by host_ptr in device memory. This cached copy can be used when kernels are executed on a device. The result of OpenCL commands that operate on multiple buffer objects created with the same host_ptr or from overlapping host or SVM regions is considered to be undefined.
CL_MEM_ALLOC_HOST_PTR	This flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory. CL_MEM_ALLOC_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.
CL_MEM_COPY_HOST_PTR	This flag is valid only if host_ptr is not NULL. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by host_ptr. The implementation will copy the memory immediately and host_ptr is available for reuse by the application when the clCreateBuffer or clCreateImage operation returns. 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 hostaccessible (e.g. PCIe) memory.
CL_MEM_HOST_WRITE_ONLY	This flag specifies that the host will only write to the memory object (using OpenCL APIs that enqueue a write or a map for write). This can be used to optimize write access from the host (e.g. enable write-combined allocations for memory objects for devices that communicate with the host over a system bus such as PCIe).
CL_MEM_HOST_READ_ONLY	This flag specifies that the host will only read the memory object (using OpenCL APIs that enqueue a read or a map for read). CL_MEM_HOST_WRITE_ONLY and CL_MEM_HOST_READ_ONLY are mutually exclusive.

Table 5.3: (continued)

CL_MEM_HOST_NO_ACCESS	This flag specifies that the host will not read or write the memory object. CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_READ_ONLY and CL_MEM_HOST_NO_ACCESS are mutually exclusive.

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.

The user is responsible for ensuring that data passed into and out of OpenCL images are natively aligned relative to the start of the buffer as per kernel language or IL requirements. OpenCL buffers created with CL_MEM_USE_HOST_PTR need to provide an appropriately aligned host memory pointer that is aligned to the data types used to access these buffers in a kernel(s).

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

If **clCreateBuffer** is called with a pointer returned by **clSVMAlloc** as its *host_ptr* argument, and CL_MEM_USE_HOST_PTR is set in its *flags* argument, **clCreateBuffer** will succeed and return a valid non-zero buffer object as long as the *size* argument to **clCreateBuffer** is no larger than the *size* argument passed in the original **clSVMAlloc** call. The new buffer object returned has the shared memory as the underlying storage. Locations in the buffers underlying shared memory can be operated on using atomic operations to the devices level of support as defined in the memory model.

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_INVALID_VALUE if values specified in *flags_are not valid as defined in _table 5.3*.
- CL_INVALID_BUFFER_SIZE if *size* is 0[4]:.
- CL_INVALID_HOST_PTR if *host_ptr* is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in *flags* or if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in *flags*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for buffer object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

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

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

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

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

flags is a bit-field that is used to specify allocation and usage information about the sub-buffer memory object being created and is described in *table 5.3*. If the CL_MEM_READ_WRITE, CL_MEM_READ_ONLY or CL_MEM_WRITE_ONLY values are not specified in *flags*, they are inherited from the corresponding memory access qualifers associated with *buffer*. The CL_MEM_USE_HOST_PTR, CL_MEM_ALLOC_HOST_PTR and CL_MEM_COPY_HOST_PTR values cannot be specified in *flags* but are inherited from the corresponding memory access qualifiers associated with *buffer*. If CL_MEM_COPY_HOST_PTR is specified in the memory access qualifier values associated with *buffer* it does not imply any additional copies when the sub-buffer is created from *buffer*. If the CL_MEM_HOST_WRITE_ONLY, CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_NO_ACCESS values are not specified in *flags*, they are inherited from the corresponding memory access qualifiers associated with *buffer*.

buffer_create_type_and _buffer_create_info describe the type of buffer object to be created. The list of supported values for *buffer_create_type* and corresponding descriptor that *buffer_create_info* points to is described in *table 5.4*.

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

cl_buffer_create_type	**Description**
CL_BUFFER_CREATE_TYPE_REGION	Create a buffer object that represents a specific region in buffer. buffer_create_info is a pointer to the following structure: typedef struct _cl_buffer_region { size_t origin; size_t size; } cl_buffer_region; (origin, size) defines the offset and size in bytes in buffer. If buffer is created with CL_MEM_USE_HOST_PTR, the host_ptr associated with the buffer object returned is host_ptr + origin. The buffer object returned references the data store allocated for buffer and points to a specific region given by (origin, size) in this data store. CL_INVALID_VALUE is returned in errcode_ret if the region specified by (origin, size) is out of bounds in buffer. CL_INVALID_BUFFER_SIZE if size is 0. CL_MISALIGNED_SUB_BUFFER_OFFSET is returned in errcode_ret if there are no devices in context associated with buffer for which the origin value is aligned to the CL_DEVICE_MEM_BASE_ADDR_ALIGN value.

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

- CL_INVALID_MEM_OBJECT if *buffer* is not a valid buffer object or is a sub-buffer object.
- CL_INVALID_VALUE if *buffer* was created with CL_MEM_WRITE_ONLY and *flags* specifies CL_MEM_READ_WRITE or CL_MEM_READ_ONLY, or if *buffer* was created with CL_MEM_READ_ONLY and

flags specifies CL_MEM_READ_WRITE or CL_MEM_WRITE_ONLY, or if *flags* specifies CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR or CL_MEM_COPY_HOST_PTR.

- CL_INVALID_VALUE if *buffer* was created with CL_MEM_HOST_WRITE_ONLY and *flags* specify CL_MEM_HOST_READ_ONLY, or if *buffer* was created with CL_MEM_HOST_READ_ONLY and *flags* specify CL_MEM_HOST_WRITE_ONLY, or if *buffer* was created with CL_MEM_HOST_NO_ACCESS and *flags* specify CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_WRITE_ONLY.
- CL_INVALID_VALUE if value specified in _buffer_create_type_is not valid.
- CL_INVALID_VALUE if value(s) specified in *buffer_create_info* (for a given *buffer_create_type*) is not valid or if *buffer_create_info* is NULL.
- CL_INVALID_BUFFER_SIZE if *size* is 0.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for sub-buffer object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

NOTE:

Concurrent reading from, writing to and copying between both a buffer object and its sub-buffer object(s) is undefined. Concurrent reading from, writing to and copying between overlapping sub-buffer objects created with the same buffer object is undefined. Only reading from both a buffer object and its sub-buffer objects or reading from multiple overlapping sub-buffer objects is defined.

5.2.2 Reading, Writing and Copying Buffer Objects

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

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

command_queue is a valid host 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.

size is the size in bytes of data being read or written.

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

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

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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue* and *buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *buffer* is not a valid buffer object.
- CL_INVALID_VALUE if the region being read or written specified by (*offset*, *size*) is out of bounds or if *ptr* is a NULL value.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the read and write operations are blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *buffer*.
- CL_INVALID_OPERATION if **clEnqueueReadBuffer** is called on *buffer* which has been created with CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_NO_ACCESS.

- CL_INVALID_OPERATION if **clEnqueueWriteBuffer** is called on *buffer* which has been created with CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_NO_ACCESS.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

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

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

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

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

buffer refers to a valid buffer object.

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

If *blocking_read* is CL_TRUE i.e. the read command is blocking, **clEnqueueReadBufferRect** does not return until the buffer data has been read and copied into memory pointed to by *ptr*.

If *blocking_read* is CL_FALSE i.e. the read command is non-blocking, **clEnqueueReadBufferRect** queues a non-blocking read command and returns. The contents of the buffer that *ptr* points to cannot be used until the read command has completed. The *event* argument returns an event object which can be used to query the execution status of the read command. When the read command has completed, the contents of the buffer that _ptr_points to__can be used by the application.

If *blocking_write_is CL_TRUE, the OpenCL implementation copies the data referred to by _ptr* and enqueues the write operation in the command-queue. The memory pointed to by *ptr* can be reused by the application after the **clEnqueueWriteBufferRect** call returns.

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

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

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

*region_defines the_*(*width* in bytes, *height* in rows_,_ *depth_in slices) of the 2D or 3D rectangle being read or written. For a 2D rectangle copy, the _depth* value given by *region*[2] should be 1. The values in region cannot be 0.

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

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

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

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

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

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

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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue* and *buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *buffer* is not a valid buffer object.
- CL_INVALID_VALUE if the region being read or written specified by (*buffer_origin*, *region, buffer_row_pitch, buffer_slice_pitch*) is out of bounds.
- CL_INVALID_VALUE if *ptr* is a NULL value.
- CL_INVALID_VALUE if any *region* array element is 0.
- CL_INVALID_VALUE if *buffer_row_pitch* is not 0 and is less than *region*[0].
- CL_INVALID_VALUE if *host_row_pitch* is not 0 and is less than *region*[0].
- CL_INVALID_VALUE if *buffer_slice_pitch* is not 0 and is less than *region*[1] * *buffer_row_pitch* and not a multiple of *buffer_row_pitch*.

- CL_INVALID_VALUE if *host_slice_pitch* is not 0 and is less than *region*[1] * *host_row_pitch* and not a multiple of *host_row_pitch*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the read and write operations are blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *buffer*.
- CL_INVALID_OPERATION if **clEnqueueReadBufferRect** is called on *buffer* which has been created with CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_NO_ACCESS.
- CL_INVALID_OPERATION if **clEnqueueWriteBufferRect** is called on *buffer* which has been created with CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_NO_ACCESS.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

NOTE:

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

- All commands that use this buffer object or a memory object (buffer or image) created from this buffer object have finished execution before the read command begins execution.
- The buffer object or memory objects created from this buffer object are not mapped.
- The buffer object or memory objects created from this buffer object are not used by any command-queue until the read command has finished execution.

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

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

- The host memory region given by (*host_ptr* + *offset*, *cb*) contains the latest bits when the enqueued write command begins execution.
- The buffer object or memory objects created from this buffer object are not mapped.
- The buffer object or memory objects created from this buffer object are not used by any command-queue until the write command has finished execution.

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

- The host memory region given by (*buffer_origin region*) contains the latest bits when the enqueued write command begins execution.
- The buffer object or memory objects created from this buffer object are not mapped.
- The buffer object or memory objects created from this buffer object are not used by any command-queue until the write command has finished execution.

The function

```
cl_int clEnqueueCopyBuffer(cl_command_queue command_queue,
                           cl_mem src_buffer,
                           cl_mem dst_buffer,
                           size_t src_offset,
                           size_t dst_offset,
                           size_t size,
                           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 a host 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*.

size refers to the size in bytes to copy.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_buffer* and *dst_buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *src_buffer* and *dst_buffer* are not valid buffer objects.
- CL_INVALID_VALUE if *src_offset*, *dst_offset*, *size*, *src_offset* + *size* or *dst_offset* + *size* require accessing elements outside the *src_buffer* and *dst_buffer* buffer objects respectively.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *src_buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.

- CL_MISALIGNED_SUB_BUFFER_OFFSET if *dst_buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_MEM_COPY_OVERLAP if *src_buffer* and *dst_buffer* are the same buffer or sub-buffer object and the source and destination regions overlap or if *src_buffer* and *dst_buffer* are different sub-buffers of the same associated buffer object and they overlap. The regions overlap if *src_offset* ⇐ *dst_offset* ⇐ *src_offset* + *size* 1 or if *dst_offset* ⇐ *src_offset* ⇐ *dst_offset* + *size* 1.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_buffer* or *dst_buffer*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

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

enqueues a command to copy a 2D or 3D rectangular region from the buffer object identified by *src_buffer_to a 2D or 3D region in the buffer object identified by _dst_buffer*. Copying begins at the source offset and destination offset which are computed as described below in the description for *src_origin* and *dst_origin*. Each byte of the region's width is copied from the source offset to the destination offset. After copying each width, the source and destination offsets are incremented by their respective source and destination row pitches. After copying each 2D rectangle, the source and destination offsets are incremented by their respective source and destination slice pitches.

Note

If *src_buffer* and *dst_buffer* are the same buffer object, *src_row_pitch* must equal *dst_row_pitch* and *src_slice_pitch* must equal *dst_slice_pitch*.

---

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

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

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

*region_defines the_*(*width* in bytes, *height* in rows_,_ *depth_in slices) of the 2D or 3D rectangle being copied. For a 2D rectangle, the _depth* value given by *region*[2] should be 1. The values in region cannot be 0.

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

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

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

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

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_buffer* and *dst_buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *src_buffer* and *dst_buffer* are not valid buffer objects.
- CL_INVALID_VALUE if (*src_origin, region, src_row_pitch, src_slice_pitch*) or (*dst_origin, region, dst_row_pitch, dst_slice_pitch*) require accessing elements outside the *src_buffer* and *dst_buffer* buffer objects respectively.
- CL_INVALID_VALUE if any *region* array element is 0.
- CL_INVALID_VALUE if *src_row_pitch* is not 0 and is less than *region*[0].
- CL_INVALID_VALUE if *dst_row_pitch* is not 0 and is less than *region*[0].
- CL_INVALID_VALUE if *src_slice_pitch* is not 0 and is less than *region*[1] * *src_row_pitch* or if *src_slice_pitch* is not 0 and is not a multiple of *src_row_pitch*.
- CL_INVALID_VALUE if *dst_slice_pitch* is not 0 and is less than *region*[1] * *dst_row_pitch* or if *dst_slice_pitch* is not 0 and is not a multiple of *dst_row_pitch*.
- CL_INVALID_VALUE if *src_buffer* and *dst_buffer* are the same buffer object and *src_slice_pitch* is not equal to *dst_slice_pitch* and *src_row_pitch* is not equal to *dst_row_pitch*.
- 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 or sub-buffer object and the source and destination regions overlap or if *src_buffer* and *dst_buffer* are different sub-buffers of the same associated buffer object and they overlap. Refer to Appendix D for details on how to determine if source and destination regions overlap.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *src_buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *dst_buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.

- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_buffer* or *dst_buffer*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.2.3 Filling Buffer Objects

The function

```
cl_int clEnqueueFillBuffer(cl_command_queue command_queue,
                           cl_mem buffer,
                           const void *pattern,
                           size_t pattern_size,
                           size_t offset,
                           size_t size,
                           cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list,
                           cl_event *event)
```

enqueues a command to fill a buffer object with a pattern of a given pattern size. The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the cl_mem_flags argument value specified when *buffer* is created is ignored by **clEnqueueFillBuffer**.

command_queue refers to the host command-queue in which the fill command will be queued. The OpenCL context associated with *command_queue* and *buffer* must be the same.

buffer is a valid buffer object.

pattern is a pointer to the data pattern of size *pattern_size* in bytes. *pattern* will be used to fill a region in *buffer* starting at *offset* and is *size* bytes in size. The data pattern must be a scalar or vector integer or floating-point data type supported by OpenCL as described in *sections 6.1.1* and *6.1.2*. For example, if *buffer* is to be filled with a pattern of float4 values, then *pattern* will be a pointer to a cl_float4 value and *pattern_size* will be sizeof(cl_float4). The maximum value of *pattern_size* is the size of the largest integer or floating-point vector data type supported by the OpenCL device. The memory associated with *pattern* can be reused or freed after the function returns.

offset is the location in bytes of the region being filled in *buffer* and must be a multiple of *pattern_size*.

size is the size in bytes of region being filled in *buffer* and must be a multiple of *pattern_size*.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

clEnqueueFillBuffer 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 host command-queue.

- CL_INVALID_CONTEXT if the context associated with *command_queue* and *buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *buffer* is not a valid buffer object.
- CL_INVALID_VALUE if *offset* or *offset* + *size* require accessing elements outside the *buffer* buffer object respectively.
- CL_INVALID_VALUE if *pattern* is NULL or if *pattern_size* is 0 or if *pattern_size* is not one of {1, 2, 4, 8, 16, 32, 64, 128}.
- CL_INVALID_VALUE if *offset* and *size* are not a multiple of *pattern_size*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *buffer* is a sub-buffer object and offset specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *buffer*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.2.4 Mapping Buffer Objects

The function

```
void clEnqueueMapBuffer(cl_command_queue command_queue,
                        cl_mem buffer,
                        cl_bool blocking_map,
                        cl_map_flags map_flags,
                        size_t offset,
                        size_t size,
                        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 host 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* is mapped into the host address space and the application can access the contents of the mapped region using the pointer returned by **clEnqueueMapBuffer**.

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 is described in *table 5.5*.

buffer is a valid buffer object. The OpenCL context associated with *command_queue* and *buffer* must be the same.

offset and *size* are the offset in bytes and the size of the region in the buffer object that is being mapped.

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

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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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

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

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

- CL_INVALID_COMMAND_QUEUE if _command_queue_is not a valid host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue_and _buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *buffer* is not a valid buffer object.
- CL_INVALID_VALUE if region being mapped given by (*offset*, *size*) is out of bounds or if *size* is 0 or if values specified in _map_flags_are not valid.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for the device associated with *queue*.
- CL_MAP_FAILURE if there is a failure to map the requested region into the host address space. This error cannot occur for buffer objects created with CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR.
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the map operation is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *buffer*.
- CL_INVALID_OPERATION if buffer_ has been created with CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_NO_ACCESS and CL_MAP_READ is set in *map_flags* or if *buffer* has been created with CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_NO_ACCESS and CL_MAP_WRITE or CL_MAP_WRITE_INVALIDATE_REGION is set in *map_flags*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.
- CL_INVALID_OPERATION if mapping would lead to overlapping regions being mapped for writing.

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

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

- The *host_ptr* specified in **clCreateBuffer** to contain the latest bits in the region being mapped when the **clEnqueueMapBuffer** command has completed.
- The pointer value returned by **clEnqueueMapBuffer** will be derived from the *host_ptr* specified when the buffer object is created.

Mapped buffer objects are unmapped using **clEnqueueUnmapMemObject**. This is described in *section 5.5.2*.

Table 5.5: *List of supported cl_map_flags values*

cl_map_flags	Description
CL_MAP_READ	This flag specifies that the region being mapped in the memory object is being mapped for reading. The pointer returned by clEnqueueMapBuffer (clEnqueueMapImage) is guaranteed to contain the latest bits in the region being mapped when the clEnqueueMapBuffer (clEnqueueMapImage) command has completed.
CL_MAP_WRITE	This flag specifies that the region being mapped in the memory object is being mapped for writing. The pointer returned by clEnqueueMap{Buffer
Image} is guaranteed to contain the latest bits in the region being mapped when the clEnqueueMapBuffer (clEnqueueMapImage) command has completed	**CL_MAP_WRITE_INVALIDATE_REGION**

5.3 Image Objects

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

5.3.1 Creating Image Objects

A **1D image**,**1D image buffer, 1D image array**,**2D image**,**2D image array and 3D image object** can be created using the following function

```
cl_mem clCreateImage(cl_context context,
                     cl_mem_flags flags,
                     const cl_image_format *image_format,
                     const cl_image_desc *image_desc,
                     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*.

For all image types except CL_MEM_OBJECT_IMAGE1D_BUFFER, if value specified for *flags* is 0, the default is used which is CL_MEM_READ_WRITE.

For CL_MEM_OBJECT_IMAGE1D_BUFFER image type, or an image created from another memory object (image or buffer), if the CL_MEM_READ_WRITE, CL_MEM_READ_ONLY or CL_MEM_WRITE_ONLY values are not specified in *flags*, they are inherited from the corresponding memory access qualifers associated with *mem_object*. The CL_MEM_USE_HOST_PTR, CL_MEM_ALLOC_HOST_PTR and CL_MEM_COPY_HOST_PTR values cannot be specified in *flags* but are inherited from the corresponding memory access qualifiers associated with *mem_object*. If CL_MEM_COPY_HOST_PTR is specified in the memory access qualifier values associated with *mem_object* it does not imply any additional copies when the image is created from *mem_object*. If the CL_MEM_HOST_WRITE_ONLY, CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_NO_ACCESS values are not specified in *flags*, they are inherited from the corresponding memory access qualifiers associated with *mem_object*.

image_format is a pointer to a structure that describes format properties of the image to be allocated. A 1D image buffer or 2D image can belink:#*msocom_12 created from a buffer by specifying a buffer object in the _image_desc→mem_object*. A 2D image can be created from another 2D image object by specifyging an image object in the *image_desc→mem_object*. Refer to *section 5.3.1.1* for a detailed description of the image format descriptor.

image_desc is a pointer to a structure that describes type and dimensions of the image to be allocated. Refer to *section 5.3.1.2* for a detailed description of the image descriptor.

host_ptr is a pointer to the image data that may already be allocated by the application. It is only used to initialize the image, and can be freed after the call to **clCreateImage**. Refer to table below for a description of how large the buffer that *host_ptr* points to must be.

Image Type	**Size of buffer that *host_ptr* points to**
CL_MEM_OBJECT_IMAGE1D	>= image_row_pitch
CL_MEM_OBJECT_IMAGE1D_BUFFER	>= image_row_pitch
CL_MEM_OBJECT_IMAGE2D	>= image_row_pitch * image_height
CL_MEM_OBJECT_IMAGE3D	>= image_slice_pitch * image_depth
CL_MEM_OBJECT_IMAGE1D_ARRAY	>= image_slice_pitch * image_array_size
CL_MEM_OBJECT_IMAGE2D_ARRAY	>= image_slice_pitch * image_array_size

For a 3D image or 2D image array, the image data specified by *host_ptr* is stored as a linear sequence of adjacent 2D image slices or 2D images respectively. Each 2D image is a linear sequence of adjacent scanlines. Each scanline is a linear sequence of image elements.

For a 2D image, the image data specified by *host_ptr* is stored as a linear sequence of adjacent scanlines. Each scanline is a linear sequence of image elements.

For a 1D image array, the image data specified by *host_ptr* is stored as a linear sequence of adjacent 1D images. Each 1D image is stored as a single scanline which is a linear sequence of adjacent elements.

For 1D image or 1D image buffer, the image data specified by *host_ptr* is stored as a single scanline which is a linear sequence of adjacent elements.

Image elements are stored according to their image format as described in section 5.3.1.1

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

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_INVALID_VALUE if values specified in _flags_are not valid.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in *image_format_are not valid or if _image_format* is NULL.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if a 2D image is created from a buffer and the row pitch and base address alignment does not follow the rules described for creating a 2D image from a buffer.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if a 2D image is created from a 2D image object and the rules described above are not followed.
- CL_INVALID_IMAGE_DESCRIPTOR if values specified in *image_desc_are not valid or if _image_desc* is NULL.
- CL_INVALID_IMAGE_SIZE if image dimensions specified in *image_desc* exceed the maximum image dimensions described in *table 4.3* for all devices in_context_.
- CL_INVALID_HOST_PTR if *host_ptr* is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in *flags* or if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in *flags*.

- CL_INVALID_VALUE if an image is being created from another memory object (buffer or image) under one of the following circumstances: 1) *mem_object* was created with CL_MEM_WRITE_ONLY and *flags* specifies CL_MEM_READ_WRITE or CL_MEM_READ_ONLY, 2) *mem_object* was created with CL_MEM_READ_ONLY and *flags* specifies CL_MEM_READ_WRITE or CL_MEM_WRITE_ONLY, 3) *flags* specifies CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR or CL_MEM_COPY_HOST_PTR.
- CL_INVALID_VALUE if an image is being created from another memory object (buffer or image) and *mem_object* object was created with CL_MEM_HOST_WRITE_ONLY and *flags* specifies CL_MEM_HOST_READ_ONLY, or if *mem_object* was created with CL_MEM_HOST_READ_ONLY and *flags* specifies CL_MEM_HOST_WRITE_ONLY, or if *mem_object* was created with CL_MEM_HOST_NO_ACCESS and_flags_ specifies CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_WRITE_ONLY.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if the *image_format* is not supported.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for image object.
- CL_INVALID_OPERATION if there are no devices in *context* that support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.3.1.1 Image Format Descriptor

The image format descriptor structure is defined as

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

image_channel_order specifies the number of channels and the channel layout i.e. the memory layout in which channels are stored in the image. Valid values are described in *table 5.6.*

image_channel_data_type describes the size of the channel data type. The list of supported values is described in *table 5.7*. The number of bits per element determined by the image_channel_data_type and image_channel_order must be a power of two.

Table 5.6: *List of supported Image Channel Order Values*

Enum values that can be specified in channel_order
CL_R, **CL_Rx** or **CL_A**
CL_INTENSITY
CL_LUMINANCE
CL_DEPTH
CL_RG, **CL_RGx** or **CL_RA**
CL_RGB or **CL_RGBx**
CL_RGBA
CL_sRGB, CL_sRGBx, CL_sRGBA, CL_sBGRA
CL_ARGB, CL_BGRA, CL_ABGR

Table 5.7: *List of supported Image Channel Data Types*

Table 5.7: (continued)

Image Channel Data Type	Description
CL_SNORM_INT8	Each channel component is a normalized signed 8-bit integer value
CL_SNORM_INT16	Each channel component is a normalized signed 16-bit integer value
CL_UNORM_INT8	Each channel component is a normalized unsigned 8-bit integer value
CL_UNORM_INT16	Each channel component is a normalized unsigned 16-bit integer value
CL_UNORM_SHORT_565	Represents a normalized 5-6-5 3-channel RGB image. The channel order must be CL_RGB or CL_RGBx.
CL_UNORM_SHORT_555	Represents a normalized x-5-5-5 4-channel xRGB image. The channel order must be CL_RGB or CL_RGBx.
CL_UNORM_INT_101010	Represents a normalized x-10-10-10 4-channel xRGB image. The channel order must be CL_RGB or CL_RGBx.
CL_UNORM_INT_101010_2	Represents a normalized 10-10-10-2 four-channel RGBA image. The channel order must be CL_RGBA.
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

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:

R	G	B	A	. . .

with the corresponding byte offsets

0	1	2	3	. . .

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:

R	G	B	A	. . .

with the corresponding byte offsets

0	2	4	6	...

image_channel_data_type values of CL_UNORM_SHORT_565, CL_UNORM_SHORT_555, CL_UNORM_INT_101010 and CL_UNORM_INT_101010_2 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. For CL_UNORM_INT_101010_2, R is in bits 31:22, G in bits 21:12, B in bits 11:2 and A in bits 1:0.

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 **clCreateImage** will return a NULL memory object.

5.3.1.2 Image Descriptor

The image descriptor structure describes the type and dimensions of the image or image array and is defined as:

```
typedef struct cl_image_desc {
    cl_mem_object_type image_type,
    size_t  image_width;
    size_t  image_height;
    size_t image_depth;
    size_t image_array_size;
    size_t image_row_pitch;
    size_t image_slice_pitch;
    cl_uint num_mip_levels;
    cl_uint num_samples;
    cl_mem mem_object;
} cl_image_desc;
```

image_type describes the image type and must be either CL_MEM_OBJECT_IMAGE1D, CL_MEM_OBJECT_IMAGE1D_BUFFER, CL_MEM_OBJECT_IMAGE1D_ARRAY, CL_MEM_OBJECT_IMAGE2D, CL_MEM_OBJECT_IMAGE2D_ARRAY or CL_MEM_OBJECT_IMAGE3D.

image_width is the width of the image in pixels. For a 2D image and image array, the image width must be a value >= 1 and ⇐ CL_DEVICE_IMAGE2D_MAX_WIDTH. For a 3D image, the image width must be a value >=1 and ⇐ CL_DEVICE_IMAGE3D_MAX_WIDTH. For a 1D image buffer, the image width must be a value >=1 and ⇐ CL_DEVICE_IMAGE_MAX_BUFFER_SIZE. For a 1D image and 1D image array, the image width must be a value >=1 and ⇐ CL_DEVICE_IMAGE2D_MAX_WIDTH.

image_height is height of the image in pixels. This is only used if the image is a 2D or 3D image, or a 2D image array. For a 2D image or image array, the image height must be a value >=1 and ⇐ CL_DEVICE_IMAGE2D_MAX_HEIGHT. For a 3D image, the image height must be a value >=1 and ⇐ CL_DEVICE_IMAGE3D_MAX_HEIGHT.

image_depth is the depth of the image in pixels. This is only used if the image is a 3D image and must be a value >= 1 and ⇐ CL_DEVICE_IMAGE3D_MAX_DEPTH.

image_array_size[5]: is the number of images in the image array. This is only used if the image is a 1D or 2D image array. The values for image_array_size, if specified, must be a value >= 1 and ⇐ CL_DEVICE_IMAGE_MAX_ARRAY_SIZE.

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. For a 2D image created from a buffer, the pitch specified (or computed if pitch

[5] Note that reading and writing 2D image arrays from a kernel with image_array_size =1 may be lower performance than 2D images

specified is 0) must be a multiple of the maximum of the CL_DEVICE_IMAGE_PITCH_ALIGNMENT value for all devices in the context associated with image_desc→mem_object and that support images.

image_slice_pitch is the size in bytes of each 2D slice in the 3D image or the size in bytes of each image in a 1D or 2D image array. This must be 0 if *host_ptr* is NULL. If *host_ptr* is not NULL, image_slice_pitch can be either 0 or >= image_row_pitch * image_height for a 2D image array or 3D image and can be either 0 or >= image_row_pitch for a 1D image array. If *host_ptr* is not NULL and image_slice_pitch= *0, image_slice_pitch is calculated as image_row_pitch * image_height for a 2D image array or 3D image and image_row_pitch for a 1D image array. If image_slice_pitch*is not 0, it must be a multiple of the image_row_pitch.

num_mip_levels and num_samples must be 0.

mem_object may refer to a valid buffer or image memory object. mem_object can be a buffer memory object if image_type is CL_MEM_OBJECT_IMAGE1D_BUFFER or CL_MEM_OBJECT_IMAGE2D[6]:. mem_object can be an image object if image_type is CL_MEM_OBJECT_IMAGE2D[7]:. Otherwise it must be NULL. The image pixels are taken from the memory objects data store. When the contents of the specified memory objects data store are modified, those changes are reflected in the contents of the image object and vice-versa at corresponding synchronization points.

For a 1D image buffer create from a buffer object, the image_width * size of element in bytes must be ⇐ size of the buffer object. The image data in the buffer object is stored as a single scanline which is a linear sequence of adjacent elements.

For a 2D image created from a buffer object, the image_row_pitch * image_height must be ⇐ size of the buffer object specified by mem_object. The image data in the buffer object is stored as a linear sequence of adjacent scanlines. Each scanline is a linear sequence of image elements padded to image_row_pitch bytes.

For an image object created from another image object, the values specified in the image descriptor except for mem_object must match the image descriptor information associated with mem_object.

Image elements are stored according to their image format as described in section 5.3.1.1.

If the buffer object specified by mem_object is created with CL_MEM_USE_HOST_PTR, the *host_ptr* specified to **clCreateBuffer** must be aligned to the minimum of the **CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT** value for all devices in the context associated with the buffer specified by mem_object and that support images.

Creating a 2D image object from another 2D image object allows users to create a new image object that shares the image data store with mem_object but views the pixels in the image with a different channel order. The restrictions are:

- all the values specified in image_desc except for mem_object must match the image descriptor information associated with mem_object.
- The *image_desc* used for creation of *mem_object* may not be equivalent to image descriptor information associated with mem_object. To ensure the values in *image_desc* will match one can query mem_object for associated information using **clGetImageInfo** function described in section 5.3.7.
- the channel data type specified in image_format must match the channel data type associated with mem_object. The channel order values[8]: supported are:

image_channel_order specified in image_format	**image channel order of mem_object**
CL_sBGRA	CL_BGRA
CL_BGRA	CL_sBGRA
CL_sRGBA	CL_RGBA
CL_RGBA	CL_sRGBA
CL_sRGB	CL_RGB
CL_RGB	CL_sRGB
CL_sRGBx	CL_RGBx
CL_RGBx	CL_sRGBx
CL_DEPTH	CL_R

[6] To create a 2D image from a buffer object that share the data store between the image and buffer object

[7] To create an image object from another image object that share the data store between these image objects.

[8] This allows developers to create a sRGB view of the image from a linear RGB view or vice-versa i.e. the pixels stored in the image can be accessed as linear RGB or sRGB values.

- the channel order specified must have the same number of channels as the channel order of mem_object.

NOTE:

Concurrent reading from, writing to and copying between both a buffer object and 1D image buffer or 2D image object associated with the buffer object is undefined. Only reading from both a buffer object and 1D image buffer or 2D image object associated with the buffer object is defined.

Writing to an image created from a buffer and then reading from this buffer in a kernel even if appropriate synchronization operations (such as a barrier) are performed between the writes and reads is undefined. Similarly, writing to the buffer and reading from the image created from this buffer with appropriate synchronization between the writes and reads is undefined.

5.3.2 Querying List of Supported Image Formats

The function

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

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

- Context
- Image type 1D, 2D, or 3D image, 1D image buffer, 1D or 2D image array.
- Image object allocation information

clGetSupportedImageFormats returns a union of image formats supported by all devices in the context.

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 queried and is described in *table 5.3*. To get a list of supported image formats that can be read from or written to by a kernel, *flags* must be set to CL_MEM_READ_WRITE (get a list of images that can be read from and written to by different kernel instances when correctly ordered by event dependencies), CL_MEM_READ_ONLY (list of images that can be read from by a kernel) or CL_MEM_WRITE_ONLY (list of images that can be written to by a kernel). To get a list of supported image formats that can be both read from and written to by the same kernel instance, *flags* must be set to CL_MEM_KERNEL_READ_AND_WRITE. Please see section 5.3.2.2 for clarification.

image_type describes the image type and must be either CL_MEM_OBJECT_IMAGE1D, CL_MEM_OBJECT_IMAGE1D_BUFFER, CL_MEM_OBJECT_IMAGE2D, CL_MEM_OBJECT_IMAGE3D, CL_MEM_OBJECT_IMAGE1D_ARRAY or CL_MEM_OBJECT_IMAGE2D_ARRAY. *num_entries* specifies the number of entries that can be returned in the memory location given by *image_formats*.

image_formats is a pointer to a memory location where the list of supported image formats are returned. Each entry describes a *cl_image_format* structure supported by the OpenCL implementation. If *image_formats* is NULL, it is ignored. *num_image_formats* is the actual number of supported image formats for a specific *context* and values specified by *flags*. If *num_image_formats* is NULL, it is ignored.

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

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

- CL_INVALID_VALUE if *flags* or *image_type* are not valid, or if *num_entries* is 0 and *image_formats* is not NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

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

5.3.2.1 Minimum List of Supported Image Formats

For 1D, 1D image from buffer, 2D, 3D image objects, 1D and 2D image array objects, the mandated minimum list of image formats that can be read from and written to by different kernel instances when correctly ordered by event dependencies and that must be supported by all devices that support images is described in *table 5.8*.

Table 5.8: *Min. list of supported image formats kernel read or write*

num_channels	channel_order	channel_data_type
1	CL_R	CL_UNORM_INT8 CL_UNORM_INT16 CL_SNORM_INT8 CL_SNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
1	CL_DEPTH[9]:	CL_UNORM_INT16 CL_FLOAT
2	CL_RG	CL_UNORM_INT8 CL_UNORM_INT16 CL_SNORM_INT8 CL_SNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT

Table 5.8: (continued)

4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SNORM_INT8 CL_SNORM_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
4	CL_sRGBA	CL_UNORM_INT8[10]:]

For 1D, 1D image from buffer, 2D, 3D image objects, 1D and 2D image array objects, the mandated minimum list of image formats that can be read from and written to by the same kernel instance and that must be supported by all devices that support images is described in *table 5.9*.

Table 5.9: *Min. list of supported image formats kernel read and write*

num_channels	**channel_order**	**channel_data_type**
1	CL_R	CL_UNORM_INT8 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_RGBA	CL_UNORM_INT8 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT

5.3.2.2 Image format mapping to OpenCL kernel language image access qualifiers

Image arguments to kernels may have the read_only, write_only or read_write qualifier. Not all image formats supported by the device and platform are valid to be passed to all of these access qualifiers. For each access qualifier, only images whose format is in the list of formats returned by clGetSupportedImageFormats with the given flag arguments in *table 5.9* are permitted. It is not valid to pass an image supporting writing as both a read_only image and a write_only image parameter, or to a read_write image parameter and any other image parameter.

[9] CL_DEPTH channel order is supported only for 2D image and 2D image array objects.

[10] sRGB channel order support is not required for 1D image buffers. Writes to images with sRGB channel orders requires device support of the cl_khr_srgb_image_writes extension.

Table 5.10: *Mapping from format flags passed to clGetSupportedImage-Formats to OpenCL kernel language image access qualifiers*

Access Qualifier	cl_mem_flags
read_only	CL_MEM_READ_ONLY, CL_MEM_READ_WRITE, CL_MEM_KERNEL_READ_AND_WRITE
write_only	CL_MEM_WRITE_ONLY, CL_MEM_READ_WRITE, CL_MEM_KERNEL_READ_AND_WRITE
read_write	CL_MEM_KERNEL_READ_AND_WRITE

5.3.3 Reading, Writing and Copying Image Objects

The following functions enqueue commands to read from an image or image array object to host memory or write to an image or image array object from host memory.

```
cl_int clEnqueueReadImage(cl_command_queue command_queue,
                          cl_mem image,
                          cl_bool blocking_read,
                          const size_t *origin,
                          const size_t *region,
                          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,
                           const size_t *region,
                           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 host 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 image or image array 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 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *image* is a 2D image object, *origin*[2] must be 0. If *image* is a 1D image or 1D image buffer object, *origin*[1] and *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[1] describes the image index in the 1D image array. If *image* is a 2D image array object, *origin*[2] describes the image index in the 2D image array.

*region_defines the_*(*width*, *height, depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *image* is a 2D image object, *region*[2] must be 1. If *image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.

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 or each image of a 1D or 2D image array being read or written respectively. This must be 0 if *image* is a 1D or 2D image. Otherwise 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. The alignment requirements for ptr are specified in section C.3.

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

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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue* and *image* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if i_mage_ is not a valid image object.
- CL_INVALID_VALUE if the region being read or written specified by *origin* and *region* is out of bounds or if *ptr* is a NULL value.
- CL_INVALID_VALUE if values in *origin* and *region* do not follow rules described in the argument description for *origin* and *region*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *image* are not supported by device associated with *queue*.

- CL_IMAGE_FORMAT_NOT_SUPPORTED if image format (image channel order and data type) for *image* are not supported by device associated with *queue*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *image*.
- CL_INVALID_OPERATION if the device associated with *command_queue* does not support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- CL_INVALID_OPERATION if **clEnqueueReadImage** is called on *image* which has been created with CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_NO_ACCESS.
- CL_INVALID_OPERATION if **clEnqueueWriteImage** is called on *image* which has been created with CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_NO_ACCESS.
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the read and write operations are blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

NOTE:

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

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

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

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

The function

```
cl_int clEnqueueCopyImage(cl_command_queue command_queue,
                          cl_mem src_image,
                          cl_mem dst_image,
                          const size_t *src_origin,
                          const size_t *dst_origin,
                          const size_t *region,
                          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 1D, 2D, 3D image or a 1D, 2D image array objects. It is possible to copy subregions between any combinations of source and destination types, provided that the dimensions of the subregions are the same e.g., one can copy a rectangular region from a 2D image to a slice of a 3D image.

command_queue refers to the host 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 (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *image* is a 2D image object, *src_origin*[2] must be 0. If *src_image* is a 1D image object, *src_origin*[1] and *src_origin*[2] must be 0. If *src_image* is a 1D image array object, *src_origin*[2] must be 0. If *src_image* is a 1D image array object, *src_origin*[1] describes the image index in the 1D image array. If *src_image* is a 2D image array object, *src_origin*[2] describes the image index in the 2D image array.

dst_origin defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *dst_image* is a 2D image object, *dst_origin*[2] must be 0. If *dst_image* is a 1D image or 1D image buffer object, *dst_origin*[1] and *dst_origin*[2] must be 0. If *dst_image* is a 1D image array object, *dst_origin*[2] must be 0. If *dst_image* is a 1D image array object, *dst_origin*[1] describes the image index in the 1D image array. If *dst_image* is a 2D image array object, *dst_origin*[2] describes the image index in the 2D image array.

*region_defines the_*(*width*, *height, depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *src_image* or *dst_image* is a 2D image object, *region*[2] must be 1. If *src_image* or *dst_image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *src_image* or *dst_image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_image* and *dst_image* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *src_image* and *dst_image* are not valid image objects.
- CL_IMAGE_FORMAT_MISMATCH if *src_image* and *dst_image* do not use the same image format.
- CL_INVALID_VALUE if the 2D or 3D rectangular region specified by *src_origin* and *src_origin* *region* refers to a region outside *src_image*, or if the 2D or 3D rectangular region specified by *dst_origin* and *dst_origin* + *region* refers to a region outside *dst_image*.
- CL_INVALID_VALUE if values in *src_origin*, *dst_origin* and *region* do not follow rules described in the argument description for *src_origin*, *dst_origin* and *region*.

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *src_image* or *dst_image* are not supported by device associated with *queue*.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if image format (image channel order and data type) for *src_image* or *dst_image* are not supported by device associated with *queue*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_image* or *dst_image*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.
- CL_INVALID_OPERATION if the device associated with *command_queue* does not support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- CL_MEM_COPY_OVERLAP if *src_image* and *dst_image* are the same image object and the source and destination regions overlap.

5.3.4 Filling Image Objects

The function

```
cl_int clEnqueueFillImage*(cl_command_queue command_queue,
                           cl_mem image,
                           const void *fill_color,
                           const size_t *origin,
                           const size_t *region,
                           cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list,
                           cl_event *event)
```

enqueues a command to fill an image object with a specified color. The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the cl_mem_flags argument value specified when *image* is created is ignored by **clEnqueueFillImage**.

command_queue refers to the host command-queue in which the fill command will be queued. The OpenCL context associated with *command_queue* and *image* must be the same.

image is a valid image object.

fill_color is the color used to fill the image. The fill color is a single floating point value if the channel order is CL_DEPTH. Otherwise, the fill color is a four component RGBA floating-point color value if the *image* channel data type is not an unnormalized signed or unsigned integer type, is a four component signed integer value if the *image* channel data type is an unnormalized signed integer type and is a four component unsigned integer value if the *image* channel data type is an unnormalized unsigned integer type. The fill color will be converted to the appropriate image channel format and order associated with *image* as described in *sections 6.12.14* and *8.3*.

origin defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *image* is a 2D image object, *origin*[2] must be 0. If *image* is a 1D image or 1D image buffer object, *origin*[1] and *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[1] describes the image index in the 1D image array. If *image* is a 2D image array object, *origin*[2] describes the image index in the 2D image array.

*region_defines the_*(*width*, *height, depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *image* is a 2D image object, *region*[2] must be 1. If *image* is a 1D image or 1D image buffer

object, *region*[1] and *region*[2] must be 1. If *image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

clEnqueueFillImage 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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue* and *image* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *image* is not a valid image object.
- CL_INVALID_VALUE if *fill_color* is NULL.
- CL_INVALID_VALUE if the region being filled as specified by *origin* and *region* is out of bounds or if *ptr* is a NULL value.
- CL_INVALID_VALUE if values in *origin* and *region* do not follow rules described in the argument description for *origin* and *region*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *image* are not supported by device associated with *queue*.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if image format (image channel order and data type) for *image* are not supported by device associated with *queue*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *image*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.3.5 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,
                                  const size_t *region,
                                  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 host 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 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *src_image* is a 2D image object, *src_origin*[2] must be 0. If *src_image* is a 1D image or 1D image buffer object, *src_origin*[1] and *src_origin*[2] must be 0. If *src_image* is a 1D image array object, *src_origin*[2] must be 0. If *src_image* is a 1D image array object, *src_origin*[1] describes the image index in the 1D image array. If *src_image* is a 2D image array object, *src_origin*[2] describes the image index in the 2D image array.

*region_defines the_*(*width*, *height, depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *src_image* is a 2D image object, *region*[2] must be 1. If *src_image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *src_image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.

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, is computed as *width * height * bytes/image element* if *src_image* is a 2D image, is computed as *width * height * arraysize * bytes/image element* if *src_image* is a 2D image array object, is computed as *width * bytes/image element* if *src_image* is a 1D image or 1D image buffer object and is computed as *width * arraysize * bytes/image element* if *src_image* is a 1D image array object.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_image* and *dst_buffer* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *src_image* is not a valid image object or *dst_buffer* is not a valid buffer object or if *src_image* is a 1D image buffer object created from *dst_buffer*.
- CL_INVALID_VALUE if the 1D, 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 values in *src_origin* and *region* do not follow rules described in the argument description for *src_origin* and *region*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.

- CL_MISALIGNED_SUB_BUFFER_OFFSET if *dst_buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *src_image* are not supported by device associated with *queue*.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if image format (image channel order and data type) for *src_image* are not supported by device associated with *queue*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_image* or *dst_buffer*.
- CL_INVALID_OPERATION if the device associated with *command_queue* does not support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clEnqueueCopyBufferToImage(cl_command_queue command_queue,
                                  cl_mem src_buffer,
                                  cl_mem dst_image,
                                  size_t src_offset,
                                  const size_t *dst_origin,
                                  const size_t *region,
                                  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 host 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 defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *dst_image* is a 2D image object, *dst_origin*[2] must be 0. If *dst_image* is a 1D image or 1D image buffer object, *dst_origin*[1] and *dst_origin*[2] must be 0. If *dst_image* is a 1D image array object, *dst_origin*[2] must be 0. If *dst_image* is a 1D image array object, *dst_origin*[1] describes the image index in the 1D image array. If *dst_image* is a 2D image array object, *dst_origin*[2] describes the image index in the 2D image array.

*region_defines the_*(*width*, *height, depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *dst_image* is a 2D image object, *region*[2] must be 1. If *dst_image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *dst_image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.

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, is computed as *width * height * bytes/image element* if *dst_image* is a 2D image, is computed as *width * height * arraysize * bytes/image element* if *dst_image* is a 2D image array object, is computed as *width * bytes/image element* if *dst_image* is a 1D image or 1D image buffer object and is computed as *width * arraysize * bytes/image element* if *dst_image* is a 1D image array object.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue*, *src_buffer* and *dst_image* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *src_buffer* is not a valid buffer object or *dst_image* is not a valid image object or if *dst_image* is a 1D image buffer object created from *src_buffer*.
- CL_INVALID_VALUE if the 1D, 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 values in *dst_origin* and *region* do not follow rules described in the argument description for *dst_origin* and *region*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_MISALIGNED_SUB_BUFFER_OFFSET if *src_buffer* is a sub-buffer object and *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *dst_image* are not supported by device associated with *queue*.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if image format (image channel order and data type) for *dst_image* are not supported by device associated with *queue*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *src_buffer* or *dst_image*.
- CL_INVALID_OPERATION if the device associated with *command_queue* does not support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.3.6 Mapping Image Objects

The function

```
void clEnqueueMapImage(cl_command_queue command_queue,
                       cl_mem image,
                       cl_bool blocking_map,
                       cl_map_flags map_flags,
```

```
                        const size_t *origin,
                        const size_t *region,
                        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 host 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 into the host address space and the application can access the contents of the mapped region using the pointer returned by **clEnqueueMapImage**.

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 is described in *table 5.5*.

origin defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *image* is a 2D image object, *origin*[2] must be 0. If *image* is a 1D image or 1D image buffer object, *origin*[1] and *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[1] describes the image index in the 1D image array. If *image* is a 2D image array object, *origin*[2] describes the image index in the 2D image array.

*region_defines the_*(*width*, *height, depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *image* is a 2D image object, *region*[2] must be 1. If *image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.

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 of a 3D image or the size of each 1D or 2D image in a 1D or 2D image array for the mapped region. For a 1D and 2D image, zero is returned if this argument is not NULL. For a 3D image, 1D and 2D image array, *image_slice_pitch* must be a non-NULL value.

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

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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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

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

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

- CL_INVALID_COMMAND_QUEUE if _command_queue_is not a valid host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue_and _image* are not the same or if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if *image* is not a valid image object.
- CL_INVALID_VALUE if region being mapped given by (*origin*, *origin+region*) is out of bounds or if values specified in _map_flags_are not valid.
- CL_INVALID_VALUE if values in *origin* and *region* do not follow rules described in the argument description for *origin* and *region*.
- CL_INVALID_VALUE if *image_row_pitch* is NULL.
- CL_INVALID_VALUE if *image* is a 3D image, 1D or 2D image array object and *image_slice_pitch* is NULL.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_INVALID_IMAGE_SIZE if image dimensions (image width, height, specified or compute row and/or slice pitch) for *image* are not supported by device associated with *queue*.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if image format (image channel order and data type) for *image* are not supported by device associated with *queue*.
- CL_MAP_FAILURE if there is a failure to map the requested region into the host address space. This error cannot occur for image objects created with CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR.
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the map operation is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with *image*.
- CL_INVALID_OPERATION if the device associated with *command_queue* does not support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- CL_INVALID_OPERATION if *image* has been created with CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_NO_ACCESS and CL_MAP_READ is set in *map_flags* or if *image* has been created with CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_NO_ACCESS and CL_MAP_WRITE or CL_MAP_WRITE_INVALIDATE_REGION is set in *map_flags*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.
- CL_INVALID_OPERATION if mapping would lead to overlapping regions being mapped for writing.

The pointer returned maps a 1D, 2D or 3D region starting at *origin* and is at least *region[0]* pixels in size for a 1D image, 1D image buffer or 1D image array, (*image_row_pitch * region[1])* pixels in size for a 2D image or 2D image array, and (*image_slice_pitch * region[2])* pixels in size for a 3D image. The result of a memory access outside this region is undefined.

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

- The *host_ptr* specified in **clCreateImage** is guaranteed to contain the latest bits in the region being mapped when the **clEnqueueMapImage** command has completed.
- The pointer value returned by **clEnqueueMapImage** will be derived from the *host_ptr* specified when the image object is created.

Mapped image objects are unmapped using **clEnqueueUnmapMemObject**. This is described in *section 5.5.2*.

5.3.7 Image Object Queries

To get information that is common to all memory objects, use the **clGetMemObjectInfo** function described in *section 5.5.5*.

To get information specific to an image object created with **clCreateImage**, use the following function

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

image specifies the image object being queried.

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

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 being queried by *param_name*. If *param_value_size_ret* is NULL, it is ignored.

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.10_and _param_value* is not NULL.
- CL_INVALID_MEM_OBJECT if *image* is a not a valid image object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Table 5.11: *List of supported param_names by clGetImageInfo*

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**.
CL_IMAGE_ELEMENT_SIZE	size_t	Return size of each element of the image memory object given by *image* in bytes. 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 calculated row pitch in bytes of a row of elements of the image object given by *image*.
CL_IMAGE_SLICE_PITCH	size_t	Return calculated slice pitch in bytes of a 2D slice for the 3D image object or size of each image in a 1D or 2D image array given by *image*. For a 1D image, 1D image buffer and 2D image object return 0.

Table 5.11: (continued)

CL_IMAGE_WIDTH	size_t	Return width of the image in pixels.
CL_IMAGE_HEIGHT	size_t	Return height of the image in pixels. For a 1D image, 1D image buffer and 1D image array object, height = 0.
CL_IMAGE_DEPTH	size_t	Return depth of the image in pixels. For a 1D image, 1D image buffer, 2D image or 1D and 2D image array object, depth = 0.
CL_IMAGE_ARRAY_SIZE	size_t	Return number of images in the image array. If *image* is not an image array, 0 is returned.
CL_IMAGE_NUM_MIP_ LEVELS	cl_uint	Return num_mip_levels associated with *image*.
CL_IMAGE_NUM_SAMPLES	cl_uint	Return num_samples associated with *image*.

5.4 Pipes

A *pipe* is a memory object that stores data organized as a FIFO. Pipe objects can only be accessed using built-in functions that read from and write to a pipe. Pipe objects are not accessible from the host. A pipe object encapsulates the following information:

- Packet size in bytes
- Maximum capacity in packets
- Information about the number of packets currently in the pipe
- Data packets

5.4.1 Creating Pipe Objects

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

```
cl_mem clCreatePipe(cl_context context,
                    cl_mem_flags flags,
                    cl_uint pipe_packet_size,
                    cl_uint pipe_max_packets,
                    const cl_pipe_properties *properties,
                    cl_int *errcode_ret)
```

context is a valid OpenCL context used to create the pipe 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 pipe object and how it will be used. *Table 5.3* describes the possible values for *flags*. Only CL_MEM_READ_WRITE and CL_MEM_HOST_NO_ACCESS can be specified when creating a pipe object. If the value specified for *flags* is 0, the default is used which is CL_MEM_READ_WRITE | CL_MEM_HOST_NO_ACCESS.

pipe_packet_size is the size in bytes of a pipe packet.

pipe_max_packets specifies the pipe capacity by specifying the maximum number of packets the pipe can hold.

properties specifies a list of properties for the pipe and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. In OpenCL 2.2, *properties* must be NULL.

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

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_INVALID_VALUE if values specified in _flags_are not as defined above.
- CL_INVALID_VALUE if *properties* is not NULL.
- CL_INVALID_PIPE_SIZE if *pipe_packet_size* is 0 or the *pipe_packet_size* exceeds CL_DEVICE_PIPE_MAX_PACKET_SIZE value specified in *table 4.3* for all devices in *context_or if _pipe_max_packets* is 0.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for the pipe object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Pipes follow the same memory consistency model as defined for buffer and image objects. The pipe state i.e. contents of the pipe across kernel-instances (on the same or different devices) is enforced at a synchronization point.

5.4.2 Pipe Object Queries

To get information that is common to all memory objects, use the **clGetMemObjectInfo** function described in *section 5.5.5*.

To get information specific to a pipe object created with **clCreatePipe**, use the following function

```
cl_int clGetPipeInfo(cl_mem pipe,
                     cl_pipe_info param_name,
                     size_t param_value_size,
                     void *param_value,
                     size_t *param_value_size_ret)
```

pipe specifies the pipe 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 **clGetPipeInfo** 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 being queried by *param_name*. If *param_value_size_ret* is NULL, it is ignored.

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.11_and _param_value* is not NULL.
- CL_INVALID_MEM_OBJECT if *pipe* is a not a valid pipe object.

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

Table 5.12: *List of supported param_names by clGetPipeInfo*

cl_pipe_info	Return type	Info. returned in *param_value*
CL_PIPE_PACKET_SIZE	cl_uint	Return pipe packet size specified when *pipe* is created with **clCreatePipe**.
CL_PIPE_MAX_PACKETS	cl_uint	Return max. number of packets specified when *pipe* is created with **clCreatePipe**.

5.5 Handling Memory Objects

5.5.1 Retaining and Releasing Memory Objects

The function

```
cl_int clRetainMemObject(cl_mem memobj)
```

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

- CL_INVALID_MEM_OBJECT if *memobj* is not a valid memory object (buffer or image object).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

clCreateBuffer, **clCreateSubBuffer**, **clCreateImage** and **clCreatePipe** perform an implicit retain.

The function

```
cl_int clReleaseMemObject(cl_mem memobj)
```

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

- CL_INVALID_MEM_OBJECT if *memobj* is not a valid memory object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

After the *memobj* reference count becomes zero and commands queued for execution on a command-queue(s) that use *memobj* have finished, the memory object is deleted. If *memobj* is a buffer object, *memobj* cannot be deleted until all sub-buffer objects associated with *memobj* are deleted. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainMemObject** causes undefined behavior.

The function

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

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

memobj is a valid memory object.

pfn_notify is the callback function that can be registered by the application. This callback function may be called asynchronously by the OpenCL implementation. It is the applications responsibility to ensure that the callback function is thread-safe. The parameters to this callback function are:

- *memobj_is the memory object being deleted. When the user callback is called by the implementation, this memory object is not longer valid. _memobj* is only provided for reference purposes.
- _user_data_is a pointer to user supplied data.

_user_data_will be passed as the _user_data_argument when _pfn_notify_is called. _user_data_can be NULL.

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

- CL_INVALID_MEM_OBJECT if *memobj* is not a valid memory object.
- CL_INVALID_VALUE if *pfn_notify* is NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Note

When the user callback function is called by the implementation, the contents of the memory region pointed to by *host_ptr* (if the memory object is created with CL_MEM_USE_HOST_PTR) are undefined. The callback function is typically used by the application to either free or reuse the memory region pointed to by *host_ptr*.

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

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

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

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

5.5.2 Unmapping Mapped Memory Objects

The function

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

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

command_queue must be a valid host command-queue.

memobj is a valid memory (buffer or image) object. The OpenCL context associated with *command_queue* and *memobj* must be the same.

mapped_ptr is the host address returned by a previous call to **clEnqueueMapBuffer**, or **clEnqueueMapImage** for *memobj*.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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 host command-queue.
- CL_INVALID_MEM_OBJECT if *memobj* is not a valid memory object or is a pipe object.
- CL_INVALID_VALUE if *mapped_ptr* is not a valid pointer returned by **clEnqueueMapBuffer** or **clEnqueueMapImage** for *memobj*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or if *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.

- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.
- CL_INVALID_CONTEXT if context associated with *command_queue* and *memobj* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.

clEnqueueMapBuffer and **clEnqueueMapImage** increment the mapped count of the memory object. The initial mapped count value of the memory object is zero. Multiple calls to **clEnqueueMapBuffer**, or **clEnqueueMapImage** on the same memory object will increment this mapped count by appropriate number of calls.
*clEnqueueUnmapMemObject*decrements the mapped count of the memory object.

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

5.5.3 Accessing mapped regions of a memory object

This section describes the behavior of OpenCL commands that access mapped regions of a memory object.

The contents of the region of a memory object and associated memory objects (sub-buffer objects or 1D image buffer objects that overlap this region) mapped for writing (i.e. CL_MAP_WRITE or CL_MAP_WRITE_INVALIDATE_REGION is set in *map_flags* argument to **clEnqueueMapBuffer**, or **clEnqueueMapImage**) are considered to be undefined until this region is unmapped.

Multiple commands in command-queues can map a region or overlapping regions of a memory object and associated memory objects (sub-buffer objects or 1D image buffer objects that overlap this region) 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 and other OpenCL commands (such as **clEnqueueCopyBuffer**) executing on a device(s).

Mapping (and unmapping) overlapped regions in a memory object and/or associated memory objects (sub-buffer objects or 1D image buffer objects that overlap this region) for writing is an error and will result in CL_INVALID_OPERATION error returned by **clEnqueueMapBuffer**, or **clEnqueueMapImage**.

If a memory object is currently mapped for writing, the application must ensure that the memory object is unmapped before any enqueued kernels or commands that read from or write to this memory object or any of its associated memory objects (sub-buffer or 1D image buffer objects) or its parent object (if the memory object is a sub-buffer or 1D image buffer object) begin execution; otherwise the behavior is undefined.

If a memory object is currently mapped for reading, the application must ensure that the memory object is unmapped before any enqueued kernels or commands that write to this memory object or any of its associated memory objects (sub-buffer or 1D image buffer objects) or its parent object (if the memory object is a sub-buffer or 1D image buffer object) begin execution; otherwise the behavior is undefined.

A memory object is considered as mapped if there are one or more active mappings for the memory object irrespective of whether the mapped regions span the entire memory object.

Accessing the contents of the memory region referred to by the mapped pointer that has been unmapped is undefined.

The mapped pointer returned by **clEnqueueMapBuffer** or **clEnqueueMapImage** can be used as *ptr* argument value to **clEnqueue{Read | Write}Buffer**, **clEnqeue{Read | Write}BufferRect**, **clEnqueue{Read | Write}Image** provided the rules described above are adhered to.

5.5.4 Migrating Memory Objects

This section describes a mechanism for assigning which device an OpenCL memory object resides. A user may wish to have more explicit control over the location of their memory objects on creation. This could be used to:

- Ensure that an object is allocated on a specific device prior to usage.
- Preemptively migrate an object from one device to another.

The function

```
cl_int clEnqueueMigrateMemObjects(cl_command_queue command_queue,
                                  cl_uint num_mem_objects,
                                  const cl_mem *mem_objects,
                                  cl_mem_migration_flags flags,
                                  cl_uint num_events_in_wait_list
                                  const cl_event *event_wait_list,
                                  cl_event *event)
```

enqueues a command to indicate which device a set of memory objects should be associated with. Typically, memory objects are implicitly migrated to a device for which enqueued commands, using the memory object, are targeted. **clEnqueueMigrateMemObjects** allows this migration to be explicitly performed ahead of the dependent commands. This allows a user to preemptively change the association of a memory object, through regular command queue scheduling, in order to prepare for another upcoming command. This also permits an application to overlap the placement of memory objects with other unrelated operations before these memory objects are needed potentially hiding transfer latencies. Once the event, returned from **clEnqueueMigrateMemObjects**, has been marked CL_COMPLETE the memory objects specified in *mem_objects* have been successfully migrated to the device associated with *command_queue*. The migrated memory object shall remain resident on the device until another command is enqueued that either implicitly or explicitly migrates it away.

clEnqueueMigrateMemObjects can also be used to direct the initial placement of a memory object, after creation, possibly avoiding the initial overhead of instantiating the object on the first enqueued command to use it.

The user is responsible for managing the event dependencies, associated with this command, in order to avoid overlapping access to memory objects. Improperly specified event dependencies passed to **clEnqueueMigrateMemObjects** could result in undefined results.

command_queue is a valid host command-queue. The specified set of memory objects in *mem_objects* will be migrated to the OpenCL device associated with *command_queue* or to the host if the CL_MIGRATE_MEM_OBJECT_HOST has been specified.

num_mem_objects_is the number of memory objects specified in _mem_objects.

mem_objects is a pointer to a list of memory objects.

flags is a bit-field that is used to specify migration options. The *table 5.12* describes the possible values for flags.

Table 5.13: *Supported values for cl_mem_migration_flags*

cl_mem_migration flags	**Description**
CL_MIGRATE_MEM_OBJECT_HOST	This flag indicates that the specified set of memory objects are to be migrated to the host, regardless of the target command-queue.
CL_MIGRATE_MEM_OBJECT_ CONTENT_UNDEFINED	This flag indicates that the contents of the set of memory objects are undefined after migration. The specified set of memory objects are migrated to the device associated with _command_queue_without incurring the overhead of migrating their contents.

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

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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

clEnqueueMigrateMemObjects 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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue* and memory objects in *mem_objects* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_MEM_OBJECT if any of the memory objects in *mem_objects* is not a valid memory object.
- CL_INVALID_VALUE if *num_mem_objects* is zero or if *mem_objects* is NULL.
- CL_INVALID_VALUE if *flags* is not 0 or is not any of the values described in the table above.
- 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_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for the specified set of memory objects in *mem_objects*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.5.5 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 5.13*.

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

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

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

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.13* and *param_value* is not NULL.
- CL_INVALID_MEM_OBJECT if *memobj* is a not a valid memory object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.

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

Table 5.14: *List of supported param_names by clGetMemObjectInfo*

cl_mem_info	Return type	Info. returned in *param_value*
CL_MEM_TYPE	cl_mem_object_type	Returns one of the following values: CL_MEM_OBJECT_BUFFER if memobj is created with clCreateBuffer or clCreateSubBuffer. cl_image_desc.image_type argument value if memobj is created with clCreateImage. CL_MEM_OBJECT_PIPE if memobj is created with clCreatePipe.
CL_MEM_FLAGS	cl_mem_flags	Return the flags argument value specified when memobj is created with clCreateBuffer, clCreateSubBuffer, clCreateImage or clCreatePipe. If memobj is a sub-buffer the memory access qualifiers inherited from parent buffer is also returned.
CL_MEM_SIZE	size_t	Return actual size of the data store associated with *memobj* in bytes.
CL_MEM_HOST_PTR	void *	If memobj is created with clCreateBuffer or clCreateImage and CL_MEM_USE_HOST_PTR is specified in mem_flags, return the host_ptr argument value specified when memobj is created. Otherwise a NULL value is returned. If memobj is created with clCreateSubBuffer, return the host_ptr + origin value specified when memobj is created. host_ptr is the argument value specified to clCreateBuffer and CL_MEM_USE_HOST_PTR is specified in mem_flags for memory object from which memobj is created. Otherwise a NULL value is returned.
CL_MEM_MAP_COUNT[11]:	cl_uint	Map count.
CL_MEM_REFERENCE_COUNT[12]:	cl_uint	Return *memobj* reference count.

Table 5.14: (continued)

CL_MEM_CONTEXT	cl_context	Return context specified when memory object is created. If *memobj* is created using **clCreateSubBuffer**, the context associated with the memory object specified as the *buffer* argument to **clCreateSubBuffer** is returned.
CL_MEM_ASSOCIATED_ MEMOBJECT	cl_mem	Return memory object from which memobj is created. This returns the memory object specified as buffer argument to clCreateSubBuffer if memobj is a subbuffer object created using clCreateSubBuffer. This returns the mem_object specified in cl_image_desc if memobj is an image object. Otherwise a NULL value is returned.
CL_MEM_OFFSET	size_t	Return offset if memobj is a sub-buffer object created using clCreateSubBuffer. This return 0 if memobj is not a subbuffer object.
CL_MEM_USES_SVM_ POINTER	cl_bool	Return CL_TRUE if *memobj* is a buffer object that was created with CL_MEM_USE_HOST_PTR or is a sub-buffer object of a buffer object that was created with CL_MEM_USE_HOST_PTR and the *host_ptr* specified when the buffer object was created is a SVM pointer; otherwise returns CL_FALSE.

5.6 Shared Virtual Memory

OpenCL 2.2 adds support for shared virtual memory (a.k.a. SVM). SVM allows the host and kernels executing on devices to directly share complex, pointer-containing data structures such as trees and linked lists. It also eliminates the need to marshal data between the host and devices. As a result, SVM substantially simplifies OpenCL programming and may improve performance.

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

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

5.6.1 SVM sharing granularity: coarse- and fine- grained sharing

OpenCL maintains memory consistency in a coarse-grained fashion in regions of buffers. We call this coarse-grained sharing. Many platforms such as those with integrated CPU-GPU processors and ones using the SVM-related PCI-SIG IOMMU services can do better, and can support sharing at a granularity smaller than a buffer. We call this fine-grained sharing. OpenCL 2.0 requires that the host and all OpenCL 2.2 devices support coarse-grained sharing at a minimum.

- Coarse-grained sharing: Coarse-grain sharing may be used for memory and virtual pointer sharing between multiple devices as well as between the host and one or more devices. The shared memory region is a memory buffer allocated using **clSVMAlloc**. Memory consistency is guaranteed at synchronization points and the host can use calls to **clEnqueueSVMMap** and **clEnqueueSVMUnmap** or create a cl_mem buffer object using the SVM pointer and use OpenCLs existing host API functions **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject** to update regions of the buffer. What coarse-grain buffer SVM adds to OpenCLs earlier buffer support are the ability to share virtual memory pointers and a guarantee that concurrent access to the same memory allocation from multiple kernels on a single device is valid. The coarse-grain buffer SVM provides a memory consistency model similar to the global memory consistency model described in *sections 3.3.1* and *3.4.3* of the OpenCL 1.2 specification. This memory consistency applies to the regions of buffers being shared in a coarse-grained fashion. It is enforced at the synchronization points between commands enqueued to command queues in a single context with the additional consideration that multiple kernels concurrently running on the same device may safely share the data.
- Fine-grained sharing: Shared virtual memory where memory consistency is maintained at a granularity smaller than a buffer. How fine-grained SVM is used depends on whether the device supports SVM atomic operations.

o If SVM atomic operations are supported, they provide memory consistency for loads and stores by the host and kernels executing on devices supporting SVM. This means that the host and devices can concurrently read and update the same memory. The consistency provided by SVM atomics is in addition to the consistency provided at synchronization points. There is no need for explicit calls to **clEnqueueSVMMap** and **clEnqueueSVMUnmap** or **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject** on a cl_mem buffer object created using the SVM pointer.

o If SVM atomic operations are not supported, the host and devices can concurrently read the same memory locations and can concurrently update non-overlapping memory regions, but attempts to update the same memory locations are undefined. Memory consistency is guaranteed at synchronization points without the need for explicit calls to to **clEnqueueSVMMap** and **clEnqueueSVMUnmap** or **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject** on a cl_mem buffer object created using the SVM pointer.

There are two kinds of fine-grain sharing support. Devices may support either fine-grain buffer sharing or fine-grain system sharing.

o Fine-grain buffer sharing provides fine-grain SVM only within buffers and is an extension of coarse-grain sharing. To support fine-grain buffer sharing in an OpenCL context, all devices in the context must support CL_DEVICE_SVM_FINE_GRAIN_BUFFER.

o Fine-grain system sharing enables fine-grain sharing of the hosts entire virtual memory, including memory regions allocated by the system **malloc** API. OpenCL buffer objects are unnecessary and programmers can pass pointers allocated using **malloc** to OpenCL kernels.

As an illustration of fine-grain SVM using SVM atomic operations to maintain memory consistency, consider the following example. The host and a set of devices can simultaneously access and update a shared work-queue data structure holding work-items to be done. The host can use atomic operations to insert new work-items into the queue at the same time as the devices using similar atomic operations to remove work-items for processing.

It is the programmers responsibility to ensure that no host code or executing kernels attempt to access a shared memory region after that memory is freed. We require the SVM implementation to work with either 32- or 64- bit host applications subject to the following requirement: the address space size must be the same for the host and all OpenCL devices in the context.

The function

```
void* clSVMAlloc(cl_context context,
                 cl_svm_mem_flags flags,
```

```
                    size_t size,
                    cl_uint alignment)
```

allocates a shared virtual memory buffer (referred to as a SVM buffer) that can be shared by the host and all devices in an OpenCL context that support shared virtual memory.

context is a valid OpenCL context used to create the SVM buffer.

flags is a bit-field that is used to specify allocation and usage information. *Table 5.14* describes the possible values for *flags*.

Table 5.15: *List of supported cl_svm_mem_flags_values*

cl_svm_mem_flags	**Description**
CL_MEM_READ_WRITE	This flag specifies that the SVM buffer will be read and written by a kernel. This is the default.
CL_MEM_WRITE_ONLY	This flag specifies that the SVM buffer will be written but not read by a kernel. Reading from a SVM buffer created with CL_MEM_WRITE_ONLY inside a kernel is undefined. CL_MEM_READ_WRITE and CL_MEM_WRITE_ONLY are mutually exclusive.
CL_MEM_READ_ONLY	This flag specifies that the SVM buffer object is a read-only memory object when used inside a kernel. Writing to a SVM buffer created with CL_MEM_READ_ONLY inside a kernel is undefined. CL_MEM_READ_WRITE or CL_MEM_WRITE_ONLY and CL_MEM_READ_ONLY are mutually exclusive.
CL_MEM_SVM_FINE_GRAIN_ BUFFER	This specifies that the application wants the OpenCL implementation to do a fine-grained allocation.
CL_MEM_SVM_ATOMICS	This flag is valid only if CL_MEM_SVM_FINE_GRAIN_BUFFER is specified in flags. It is used to indicate that SVM atomic operations can control visibility of memory accesses in this SVM buffer.

If CL_MEM_SVM_FINE_GRAIN_BUFFER is not specified, the buffer can be created as a coarse grained SVM allocation. Similarly, if CL_MEM_SVM_ATOMICS is not specified, the buffer can be created without support for SVM atomic operations (refer to an OpenCL kernel language specifications).

size is the size in bytes of the SVM buffer to be allocated.

alignment is the minimum alignment in bytes that is required for the newly created buffers memory region. It must be a power of two up to the largest data type supported by the OpenCL device. For the full profile, the largest data type is long16. For the embedded profile, it is long16 if the device supports 64-bit integers; otherwise it is int16. If alignment is 0, a default alignment will be used that is equal to the size of largest data type supported by the OpenCL implementation.

clSVMAlloc returns a valid non-NULL shared virtual memory address if the SVM buffer is successfully allocated. Otherwise, like **malloc**, it returns a NULL pointer value. **clSVMAlloc** will fail if

- *context* is not a valid context.

- *flags* does not contain CL_MEM_SVM_FINE_GRAIN_BUFFER but does contain CL_MEM_SVM_ATOMICS.
- Values specified in *flags* do not follow rules described for supported values in *table 5.14*.
- CL_MEM_SVM_FINE_GRAIN_BUFFER or CL_MEM_SVM_ATOMICS is specified in *flags* and these are not supported by at least one device in *context*.
- The values specified in *flags* are not valid i.e. dont match those defined in *table 5.14*.
- *size* is 0 or > CL_DEVICE_MAX_MEM_ALLOC_SIZE value for any device in *context*.
- *alignment* is not a power of two or the OpenCL implementation cannot support the specified alignment for at least one device in *context*.
- There was a failure to allocate resources.

Calling **clSVMAlloc** does not itself provide consistency for the shared memory region. When the host cant use the SVM atomic operations, it must rely on OpenCLs guaranteed memory consistency at synchronization points.

For SVM to be used efficiently, the host and any devices sharing a buffer containing virtual memory pointers should have the same endianness. If the context passed to **clSVMAlloc** has devices with mixed endianness and the OpenCL implementation is unable to implement SVM because of that mixed endianness, **clSVMAlloc** will fail and return NULL.

Although SVM is generally not supported for image objects, **clCreateImage** may create an image from a buffer (a 1D image from a buffer or a 2D image from buffer) if the buffer specified in its image description parameter is a SVM buffer. Such images have a linear memory representation so their memory can be shared using SVM. However, fine grained sharing and atomics are not supported for image reads and writes in a kernel.

The function

```
void clSVMFree(cl_context context,
               void * svm_pointer)
```

frees a shared virtual memory buffer allocated using **clSVMAlloc**.

context is a valid OpenCL context used to create the SVM buffer.

svm_pointer must be the value returned by a call to **clSVMAlloc**. If a NULL pointer is passed in *svm_pointer*, no action occurs.

Note that **clSVMFree** does not wait for previously enqueued commands that may be using *svm_pointer* to finish before freeing *svm_pointer*. It is the responsibility of the application to make sure that enqueued commands that use *svm_pointer* have finished before freeing *svm_pointer*. This can be done by enqueuing a blocking operation such as **clFinish**, **clWaitForEvents**, **clEnqueueReadBuffer** or by registering a callback with the events associated with enqueued commands and when the last enqueued comamnd has finished freeing *svm_pointer*.

The behavior of using *svm_pointer* after it has been freed is undefined. In addition, if a buffer object is created using **clCreateBuffer** with *svm_pointer*, the buffer object must first be released before the *svm_pointer* is freed.

The **clEnqueueSVMFree** API can also be used to enqueue a callback to free the shared virtual memory buffer allocated using **clSVMAlloc** or a shared system memory pointer.

The function

```
cl_int clEnqueueSVMFree(cl_command_queue command_queue,
                        cl_uint num_svm_pointers,
                        void *svm_pointers[],
                        void (CL_CALLBACK *_pfn_free_func_)
                            (cl_command_queue queue,
                            cl_uint num_svm_pointers,
                            void *svm_pointers[],
                            void *user_data),
                        void *user_data,
                        cl_uint num_events_in_wait_list,
                        const cl_event *event_wait_list,
                        cl_event *event)
```

enqueues a command to free the shared virtual memory allocated using **clSVMAlloc** or a shared system memory pointer.

command_queue is a valid host command-queue.

svm_pointers and *num_svm_pointers* specify shared virtual memory pointers to be freed. Each pointer in *svm_pointers* that was allocated using **clSVMAlloc** must have been allocated from the same context from which *command_queue* was created. The memory associated with *svm_pointers* can be reused or freed after the function returns.

pfn_free_func specifies the callback function to be called to free the SVM pointers. *pfn_free_func* takes four arguments: *queue* which is the command queue in which **clEnqueueSVMFree** was enqueued, the count and list of SVM pointers to free and *user_data* which is a pointer to user specified data. If *pfn_free_func* is NULL, all pointers specified in *svm_pointers* must be allocated using **clSVMAlloc** and the OpenCL implementation will free these SVM pointers. *pfn_free_func* must be a valid callback function if any SVM pointer to be freed is a shared system memory pointer i.e. not allocated using **clSVMAlloc**. If *pfn_free_func* is a valid callback function, the OpenCL implementation will call *pfn_free_func* to free all the SVM pointers specified in *svm_pointers*.

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

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

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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

clEnqueueSVMFree 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 host command-queue.
- CL_INVALID_VALUE if *num_svm_pointers* is 0 and *svm_pointers_is non-NULL, _or* if *svm_pointers* is NULL and _num_svm_pointers_is not 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_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The following function enqueues a command to do a memcpy operation.

```
cl_int clEnqueueSVMMemcpy(cl_command_queue command_queue,
                          cl_bool blocking_copy,
                          void *dst_ptr,
                          const void *src_ptr,
                          size_t size,
                          cl_uint num_events_in_wait_list,
                          const cl_event *event_wait_list,
                          cl_event *event)
```

command_queue refers to the host command-queue in which the read / write command will be queued. If either *dst_ptr* or *src_ptr* is allocated using clSVMAlloc then the OpenCL context allocated against must match that of *command_queue*.

blocking_copy indicates if the copy operation is *blocking* or *non-blocking*.

If *blocking_copy* is CL_TRUE i.e. the copy command is blocking, **clEnqueueSVMMemcpy** does not return until the buffer data has been copied into memory pointed to by *dst_ptr*.

If *blocking_copy* is CL_FALSE i.e. the copy command is non-blocking, **clEnqueueSVMMemcpy** queues a non-blocking copy command and returns. The contents of the buffer that *dst_ptr* point to cannot be used until the copy 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 copy command has completed, the contents of the buffer that _dst_ptr_points to__can be used by the application.

size is the size in bytes of data being copied.

dst_ptr is the pointer to a host or SVM memory allocation where data is copied to.

src_ptr is the pointer to a host or SVM memory allocation where data is copied from.

If the memory allocation(s) containing *dst_ptr* and/or *src_ptr* are allocated using **clSVMAlloc** and either is not allocated from the same context from which *command_queue* was created the behavior is undefined.

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

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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

clEnqueueSVMMemcpy 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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- 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_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the copy operation is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.
- CL_INVALID_VALUE if *dst_ptr* or *src_ptr* are NULL.
- CL_MEM_COPY_OVERLAP if the values specified for *dst_ptr*, *src_ptr* and *size* result in an overlapping copy.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clEnqueueSVMMemFill(cl_command_queue command_queue,
                           void *svm_ptr,
                           const void *pattern,
                           size_t pattern_size,
                           size_t size,
                           cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list,
                           cl_event *event)
```

enqueues a command to fill a region in memory with a pattern of a given pattern size.

command_queue refers to the host command-queue in which the fill command will be queued. The OpenCL context associated with *command_queue* and SVM pointer referred to by *svm_ptr* must be the same.

svm_ptr is a pointer to a memory region that will be filled with *pattern*. It must be aligned to *pattern_size* bytes. If *svm_ptr* is allocated using **clSVMAlloc** then it must be allocated from the same context from which *command_queue* was created. Otherwise the behavior is undefined.

pattern is a pointer to the data pattern of size *pattern_size* in bytes. *pattern* will be used to fill a region in *buffer* starting at *svm_ptr* and is *size* bytes in size. The data pattern must be a scalar or vector integer or floating-point data type supported by OpenCL as described in *sections 6.1.1* and *6.1.2*. For example, if region pointed to by *svm_ptr* is to be filled with a pattern of float4 values, then *pattern* will be a pointer to a cl_float4 value and *pattern_size* will be sizeof(cl_float4). The maximum value of *pattern_size* is the size of the largest integer or floating-point vector data type supported by the OpenCL device. The memory associated with *pattern* can be reused or freed after the function returns.

size is the size in bytes of region being filled starting with *svm_ptr* and must be a multiple of *pattern_size*.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

clEnqueueSVMMemFill 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 host command-queue.
- CL_INVALID_CONTEXT if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_VALUE if *svm_ptr* is NULL.
- CL_INVALID_VALUE if *svm_ptr* is not aligned to *pattern_size* bytes.
- CL_INVALID_VALUE if *pattern* is NULL or if *pattern_size* is 0 or if *pattern_size* is not one of {1, 2, 4, 8, 16, 32, 64, 128}.
- CL_INVALID_VALUE if *size* is not a multiple of *pattern_size*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clEnqueueSVMMap(cl_command_queue command_queue,
                       cl_bool blocking_map,
                       cl_map_flags map_flags,
                       void *svm_ptr,
                       size_t size,
                       cl_uint num_events_in_wait_list,
                       const cl_event *event_wait_list,
                       cl_event *event)
```

enqueues a command that will allow the host to update a region of a SVM buffer. Note that since we are enqueuing a command with a SVM buffer, the region is already mapped in the host address space.

command_queue must be a valid host command-queue.

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

If *blocking_map* is CL_TRUE, **clEnqueueSVMMap** does not return until the application can access the contents of the SVM region specified by *svm_ptr* and *size* on the host.

If *blocking_map* is CL_FALSE i.e. map operation is non-blocking, the region specified by *svm_ptr* and *size* 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 region specified by *svm_ptr* and *size*.

map_flags is a bit-field and is described in *table 5.5*.

svm_ptr and *size* are a pointer to a memory region and size in bytes that will be updated by the host. If *svm_ptr* is allocated using **clSVMAlloc** then it must be allocated from the same context from which *command_queue* was created. Otherwise the behavior is undefined.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

clEnqueueSVMMap 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 host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_VALUE if *svm_ptr* is NULL.
- CL_INVALID_VALUE if *size* is 0 or if values specified in _map_flags_are not valid.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the map operation is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clEnqueueSVMUnmap(cl_command_queue command_queue,
                         void *svm_ptr,
                         cl_uint num_events_in_wait_list,
                         const cl_event *event_wait_list,
                         cl_event *event)
```

enqueues a command to indicate that the host has completed updating the region given by *svm_ptr* and which was specified in a previous call to **clEnqueueSVMMap**.

command_queue must be a valid host command-queue.

svm_ptr is a pointer that was specified in a previous call to **clEnqueueSVMMap**. If *svm_ptr* is allocated using **clSVMAlloc** then it must be allocated from the same context from which *command_queue* was created. Otherwise the behavior is undefined.

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

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. **clEnqueueBarrierWithWaitList** can be used instead. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

clEnqueueSVMUnmap 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 host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_VALUE if *svm_ptr* is NULL.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or if *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

clEnqueueSVMMap and **clEnqueueSVMUnmap** act as synchronization points for the region of the SVM buffer specified in these calls.

NOTE:

If a coarse-grained SVM buffer is currently mapped for writing, the application must ensure that the SVM buffer is unmapped before any enqueued kernels or commands that read from or write to this SVM buffer or any of its associated cl_mem buffer objects begin execution; otherwise the behavior is undefined.

If a coarse-grained SVM buffer is currently mapped for reading, the application must ensure that the SVM buffer is unmapped before any enqueued kernels or commands that write to this memory object or any of its associated cl_mem buffer objects begin execution; otherwise the behavior is undefined.

A SVM buffer is considered as mapped if there are one or more active mappings for the SVM buffer irrespective of whether the mapped regions span the entire SVM buffer.

The above note does not apply to fine-grained SVM buffers (fine-grained buffers allocated using **clSVMAlloc** or fine-grained system allocations).

The function

```
cl_int clEnqueueSVMMigrateMem(cl_command_queue command_queue,
                              cl_uint num_svm_pointers,
                              const void **svm_pointers,
```

```
                                const size_t *sizes,
                                cl_mem_migration_flags flags,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

enqueues a command to indicate which device a set of ranges of SVM allocations should be associated with. Once the event returned by **clEnqueueSVMMigrateMem** has become CL_COMPLETE, the ranges specified by svm pointers and sizes have been successfully migrated to the device associated with command queue.

The user is responsible for managing the event dependencies associated with this command in order to avoid overlapping access to SVM allocations. Improperly specified event dependencies passed to **clEnqueueSVMMigrateMem** could result in undefined results.

command_queue is a valid host command queue. The specified set of allocation ranges will be migrated to the OpenCL device associated with *command_queue*.

num_svm_pointers is the number of pointers in the specified *svm_pointers* array, and the number of sizes in the *sizes* array, if _sizes_is not NULL.

svm_pointers is a pointer to an array of pointers. Each pointer in this array must be within an allocation produced by a call to **clSVMAlloc**.

sizes is an array of sizes. The pair *svm_pointers*[i] and *sizes*[i] together define the starting address and number of bytes in a range to be migrated. *sizes* may be NULL indicating that every allocation containing any *svm_pointer*[i] is to be migrated. Also, if *sizes*[i] is zero, then the entire allocation containing *svm_pointer*[i] is migrated.

flags is a bit-field that is used to specify migration options. *Table 5.12* describes the possible values for *flags*.

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

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 another command that waits for this command to complete. If the *event_wait_list* and *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

clEnqueueSVMMigrateMem 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 host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_VALUE if _num_svm_pointers is zero_or_svm_pointers_is NULL.
- CL_INVALID_VALUE if *sizesis non-zero*range [*svm_pointers*[i], *svm_pointers*[i]+*sizes*[i]) is not contained within an existing **clSVMAlloc** allocation.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or if *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.6.2 Memory consistency for SVM allocations

To ensure memory consistency in SVM allocations, the program can rely on the guaranteed memory consistency at synchronization points. This consistency support already exists in OpenCL 1.x and can be used for coarse-grained SVM allocations or for fine-grained buffer SVM allocations; what SVM adds is the ability to share pointers between the host and all SVM devices.

In addition, sub-buffers can also be used to ensure that each device gets a consistent view of a SVM buffers memory when it is shared by multiple devices. For example, assume that two devices share a SVM pointer. The host can create a cl_mem buffer object using **clCreateBuffer** with CL_MEM_USE_HOST_PTR and *host_ptr* set to the SVM pointer and then create two disjoint sub-buffers with starting virtual addresses *sb1_ptr* and *sb2_ptr*. These pointers (*sb1_ptr* and *sb2_ptr*) can be passed to kernels executing on the two devices. **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject** and the existing access rules for memory objects (in *section 5.5.3*) can be used to ensure consistency for buffer regions (*sb1_ptr* and *sb2_ptr*) read and written by these kernels.

When the host and devices are able to use SVM atomic operations (i.e. CL_DEVICE_SVM_ATOMICS is set in CL_DEVICE_SVM_CAPABILITIES), these atomic operations can be used to provide memory consistency at a fine grain in a shared memory region. The effect of these operations is visible to the host and all devices with which that memory is shared.

5.7 Sampler Objects

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

5.7.1 Creating Sampler Objects

The function

```
cl_sampler clCreateSamplerWithProperties(cl_context context,
                                         const cl_sampler_properties *sampler_properties ↩
                                             ,
                                         cl_int *errcode_ret)
```

creates a sampler object.

context must be a valid OpenCL context.

sampler_properties specifies a list of sampler property names and their corresponding values. Each sampler property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties is described in *table 5.15*. If a supported property and its value is not specified in *sampler_properties*, its default value will be used. *sampler_properties* can be NULL in which case the default values for supported sampler properties will be used.

Table 5.16: *List of supported cl_sampler_properties values and description*

cl_sampler_properties enum	**Property Value**	**Description**

Table 5.16: (continued)

CL_SAMPLER_NORMALIZED_ COORDS	cl_bool	A boolean value that specifies whether the image coordinates specified are normalized or not. The default value (i.e. the value used if this property is not specified in sampler_properties) is CL_TRUE.
CL_SAMPLER_ADDRESSING_ MODE	cl_addressing_ + mode	Specifies how out-of-range image coordinates are handled when reading from an image. Valid values are: CL_ADDRESS_MIRRORED_REPEAT CL_ADDRESS_REPEAT CL_ADDRESS_CLAMP_ CL_ADDRESS_CLAMP CL_ADDRESS_NONE The default is CL_ADDRESS_CLAMP.
CL_SAMPLER_FILTER_MODE	cl_filter_mode	Specifies the type of filter that must be applied when reading an image. Valid values are: CL_FILTER_NEAREST CL_FILTER_LINEAR The default value is CL_FILTER_NEAREST.

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

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_INVALID_VALUE if the property name in *sampler_properties* is not a supported property name, if the value specified for a supported property name is not valid, or if the same property name is specified more than once.
- CL_INVALID_OPERATION if images are not supported by any device associated with *context* (i.e. CL_DEVICE_IMAGE_SUPPORT specified in *table 4.3* is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clRetainSampler(cl_sampler sampler)
```

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

- CL_INVALID_SAMPLER if *sampler* is not a valid sampler object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clReleaseSampler(cl_sampler sampler)
```

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

- CL_INVALID_SAMPLER if *sampler* is not a valid sampler object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Using this function to release a reference that was not obtained by creating the object or by calling **clRetainSampler** causes undefined behavior.

5.7.2 Sampler Object Queries

The function

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

returns information about the sampler object.

sampler specifies the sampler being queried.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetSamplerInfo** is described in *table 5.16*.

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 being queried by *param_name*. If *param_value_size_ret* is NULL, it is ignored.

Table 5.17: *clGetSamplerInfo parameter queries*

Table 5.17: (continued)

cl_sampler_info	Return Type	Info. returned in *param_value*
CL_SAMPLER_REFERENCE_ COUNT[13]:	cl_uint	Return the *sampler* reference count.
CL_SAMPLER_CONTEXT	cl_context	Return the context specified when the sampler is created.
CL_SAMPLER_NORMALIZED_ COORDS	cl_bool	Return the normalized coords value associated with *sampler*.
CL_SAMPLER_ADDRESSING_ MODE	cl_addressing_mode	Return the addressing mode value associated with *sampler*.
CL_SAMPLER_FILTER_MODE	cl_filter_mode	Return the filter mode value associated with *sampler*.

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.16_and _param_value* is not NULL.
- CL_INVALID_SAMPLER if *sampler* is a not a valid sampler object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8 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, library or compiled binary, the list of devices for which the program executable, library or compiled binary is built, the build options used and a build log.
- The number of kernel objects currently attached.

5.8.1 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)
```

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

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*. The source code specified by *strings* is either an OpenCL C program source, header or implementation-defined source for custom devices that support an online compiler. OpenCL C++ is not supported as an online-compiled kernel language through this interface.

context must be a valid OpenCL context.

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

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

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

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_INVALID_VALUE if *count* is zero or if *strings* or any entry in *strings* is NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_program clCreateProgramWithIL(cl_context context,
                                 const void *il,
                                 size_t length,
                                 cl_int *errcode_ret)
```

creates a program object for a context, and loads the IL pointed to by _il_and with length in bytes _length_into the program object.

context must be a valid OpenCL context.

il_is a pointer to a _length-byte block of memory containing SPIR-V or an implementation-defined intermediate language.

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

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_INVALID_VALUE if *il* is NULL or if _length_is zero.
- CL_INVALID_VALUE if the *length*-byte memory pointed to by *il* does not contain well-formed intermediate language input that can be consumed by the OpenCL runtime.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

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

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

context must be a valid OpenCL context.

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

num_devices_is the number of devices listed in _device_list.

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

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

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

The program binaries specified by *binaries* contain the bits that describe one of the following:

- a program executable to be run on the device(s) associated with *context*,
- a compiled program for device(s) associated with *context*, or
- a library of compiled programs for device(s) associated with *context*.

The program binary can consist of either or both:

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

binary_status returns whether the program binary for each device specified in *device_list* was loaded successfully or not. It is an array of *num_devices* entries and returns CL_SUCCESS in *binary_status[i]* if binary was successfully loaded for device specified by *device_list[i]*; otherwise returns CL_INVALID_VALUE if *lengths[i]* is zero or if *binaries[i]* is a NULL value or CL_INVALID_BINARY in *binary_status[i]* if program binary is not a valid binary for the specified device. If *binary_status* is NULL, it is ignored.

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

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_INVALID_VALUE if *device_list* is NULL or *num_devices* is zero.
- CL_INVALID_DEVICE if OpenCL devices listed in *device_list* are not in the list of devices associated with *context*.
- CL_INVALID_VALUE if *lengths* or *binaries* are NULL or if any entry in *lengths*[i] is zero or *binaries*[i] is NULL.
- CL_INVALID_BINARY if an invalid program binary was encountered for any device. *binary_status* will return specific status for each device.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.

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

OpenCL allows applications to create a program object using the program source or binary and build appropriate program executables. This can be very useful as it allows applications to load program source and then compile and link to generate a program executable online on its first instance for appropriate OpenCL devices in the system. These executables can now be queried and cached by the application. The cached executables can be read and loaded by the application, which can help significantly reduce the application initialization time.

The function

```
cl_program clCreateProgramWithBuiltInKernels(cl_context context,
                                             cl_uint num_devices,
                                             const cl_device_id *device_list,
                                             const char *kernel_names,
                                             cl_int *errcode_ret)
```

creates a program object for a context, and loads the information related to the built-in kernels into a program object.

context must be a valid OpenCL context.

num_devices_is the number of devices listed in _device_list.

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

kernel_names is a semi-colon separated list of built-in kernel names.

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_INVALID_VALUE if *device_list* is NULL or *num_devices* is zero.
- CL_INVALID_VALUE if *kernel_names* is NULL or *kernel_names* contains a kernel name that is not supported by any of the devices in *device_list*.
- CL_INVALID_DEVICE if devices listed in *device_list* are not in the list of devices associated with *context*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.2 Retaining and Releasing Program Objects

The function

```
cl_int clRetainProgram(cl_program program)
```

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

- CL_INVALID_PROGRAM if *program* is not a valid program object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.

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

The function

```
cl_int clReleaseProgram(cl_program program)
```

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

- CL_INVALID_PROGRAM if *program* is not a valid program object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Using this function to release a reference that was not obtained by creating the object or by calling **clRetainProgram** causes undefined behavior.

The function

```
cl_int clSetProgramReleaseCallback(cl_program program,
                                   void (CL_CALLBACK *pfn_notify)
                                       (cl_program prog,
                                       void *user_data),
                                   void *user_data)
```

registers a user callback function with a program object. Each call to **clSetProgramReleaseCallback** registers the specified user callback function on a callback stack associated with program. The registered user callback functions are called in the reverse order in which they were registered. The user callback functions are called after destructors (if any) for program scope global variables (if any) are called and before the program is released. This provides a mechanism for the application (and libraries) to be notified when destructors are complete.

program is a valid program object

pfn_notify is the callback function that can be registered by the application. This callback function may be called asynchronously by the OpenCL implementation. It is the applications responsibility to ensure that the callback function is thread safe. The parameters to this callback function are:

prog is the program object whose destructors are being called. When the user callback is called by the implementation, this program object is not longer valid. prog is only provided for reference purposes.

user_data is a pointer to user supplied data. *user_data* will be passed as the *user_data* argument when *pfn_notify* is called. user data can be NULL.

clSetProgramReleaseCallback 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 *pfn_notify* is NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.3 Setting SPIR-V specialization constants

The function

```
cl_int clSetProgramSpecializationConstant(cl_program program,
                                          cl_uint spec_id,
                                          size_t spec_size,
                                          const void *spec_value)
```

sets the values of a SPIR-V specialization constants.

program must be a valid OpenCL program created from a SPIR-V module.

spec id_ identifies the SPIR-V specialization constant whose value will be set.

spec_size specifies the size in bytes of the data pointed to by *spec_value*. This should be 1 for boolean constants. For all other constant types this should match the size of the specialization constant in the SPIR-V module.

spec_value is a pointer to the memory location that contains the value of the specialization constant. The data pointed to by *spec_value* are copied and can be safely reused by the application after **clSetProgramSpecializationConstant** returns. This specialization value will be used by subsequent calls to **clBuildProgram** until another call to **clSetProgramSpecializationConstant** changes it. If a specialization constant is a boolean constant, _spec value_should be a pointer to a cl_uchar value. A value of zero will set the specialization constant to false; any other value will set it to true.

Calling this function multiple times for the same specialization constant shall cause the last provided value to override any previously specified value. The values are used by a subsequent **clBuildProgram** call for the *program*.

Application is not required to provide values for every specialization constant contained in SPIR-V module. SPIR-V provides default values for all specialization constants.

clSetProgramSpecializationConstant 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 created from a SPIR-V module.
- CL_INVALID_SPEC_ID if *spec_id* is not a valid specialization constant ID
- CL_INVALID_VALUE if *spec_size* does not match the size of the specialization constant in the SPIR-V module, or if *spec_value* is NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.4 Building Program Executables

The function

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

builds (compiles & links) a program executable from the program source or binary for all the devices or a specific device(s) in the OpenCL context associated with *program*. OpenCL allows program executables to be built using the

source or the binary. **clBuildProgram** must be called for *program* created using **clCreateProgramWithSource**, **clCreateProgramWithIL** or **clCreateProgramWithBinary** to build the program executable for one or more devices associated with *program*. If *program* is created with **clCreateProgramWithBinary**, then the program binary must be an executable binary (not a compiled binary or library).

The executable binary can be queried using **clGetProgramInfo**(*program*, CL_PROGRAM_BINARIES,) and can be specified to **clCreateProgramWithBinary** to create a new program object.

program is the program object.

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

num_devices_is the number of devices listed in _device_list.

options is a pointer to a null-terminated string of characters that describes the build options to be used for building the program executable. The list of supported options is described *in section 5.8.6*. If the program was created using clCreateProgramWithBinary and *options* is a NULL pointer, the program will be built as if options were the same as when the program binary was originally built. If the program was created using clCreateProgramWithBinary and *options* string contains anything other than the same options in the same order (whitespace ignored) as when the program binary was originally built, then the behavior is implementation defined.

pfn_notify is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not NULL, **clBuildProgram** does not need to wait for the build to complete and can return immediately once the build operation can begin. The build operation can begin if the context, program whose sources are being compiled and linked, list of devices and build options specified are all valid and appropriate host and device resources needed to perform the build are available. 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 applications responsibility to ensure that the callback function is thread-safe.

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

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

- CL_INVALID_PROGRAM if *program* is not a valid program object.
- CL_INVALID_VALUE if *device_list* is NULL and *num_devices* is greater than zero, or if *device_list* is not NULL and *num_devices* is zero.
- CL_INVALID_VALUE if *pfn_notify* is NULL but *user_data* is not NULL.
- CL_INVALID_DEVICE if OpenCL devices listed in *device_list* are not in the list of devices associated with *program*
- CL_INVALID_BINARY if *program* is created with **clCreateProgramWithBinary** 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_COMPILER_NOT_AVAILABLE if *program* is created with **clCreateProgramWithSource** and a compiler is not available i.e. CL_DEVICE_COMPILER_AVAILABLE specified in *table 4.3* is set to CL_FALSE.
- CL_BUILD_PROGRAM_FAILURE if there is a failure to build the program executable. This error will be returned if **clBuildProgram** does not return until the build has completed.
- CL_INVALID_OPERATION if 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_INVALID_OPERATION if there are kernel objects attached to *program*.
- CL_INVALID_OPERATION if *program* was not created with **clCreateProgramWithSource, clCreateProgramWithIL** or **clCreateProgramWithBinary**.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.5 Separate Compilation and Linking of Programs

OpenCL programs are compiled and linked to support the following:

- Separate compilation and link stages. Program sources can be compiled to generate a compiled binary object and linked in a separate stage with other compiled program objects to the program exectuable.
- Embedded headers. In OpenCL 1.0 and 1.1, the I build option could be used to specify the list of directories to be searched for headers files that are included by a program source(s). OpenCL 1.2 extends this by allowing the header sources to come from program objects instead of just header files.
- Libraries. The linker can be used to link compiled objects and libraries into a program executable or to create a library of compiled binaries.

The function

```
cl_int clCompileProgram(cl_program program,
                        cl_uint num_devices,
                        const cl_device_id *device_list,
                        const char *options,
                        cl_uint num_input_headers,
                        const cl_program *input_headers,
                        const char **header_include_names,
                        void (CL_CALLBACK *pfn_notify)
                            (cl_program program,
                            void *user_data),
                        void *user_data)
```

compiles a programs source for all the devices or a specific device(s) in the OpenCL context associated with *program*. The pre-processor runs before the program sources are compiled. The compiled binary is built for all devices associated with *program* or the list of devices specified. The compiled binary can be queried using **clGetProgramInfo**(*program*, CL_PROGRAM_BINARIES,) and can be passed to **clCreateProgramWithBinary** to create a new program object.

program is the program object that is the compilation target.

device_list is a pointer to a list of devices associated with *program*. If *device_list* is a NULL value, the compile is performed for all devices associated with *program*. If *device_list* is a non-NULL value, the compile is performed for devices specified in this list.

num_devices_is the number of devices listed in _device_list.

options is a pointer to a null-terminated string of characters that describes the compilation options to be used for building the program executable. Certain options are ignored when program is created with IL. The list of supported options is as described *in section 5.8.4*.

num_input_headers specifies the number of programs that describe headers in the array referenced by *input_headers*.

input_headers is an array of program embedded headers created with **clCreateProgramWithSource**.

header_include_names is an array that has a one to one correspondence with *input_headers*. Each entry in *header_include_names* specifies the include name used by source in *program* that comes from an embedded header. The corresponding entry in *input_headers* identifies the program object which contains the header source to be used. The embedded headers are first searched before the headers in the list of directories specified by the I compile option (as described in *section 5.8.4.1*). If multiple entries in *header_include_names* refer to the same header name, the first one encountered will be used.

If *program* was created using clCreateProgramWithIL, then *num_input_headers*, *input_headers*, and *header_include_names* are ignored.

For example, consider the following program source:

```
#include <foo.h>
#include <mydir/myinc.h>

__kernel void

image_filter (int n, int m,

 __constant float *filter_weights,

 __read_only image2d_t src_image,

 __write_only image2d_t dst_image)

{

...

}
```

This kernel includes two headers foo.h and mydir/myinc.h. The following describes how these headers can be passed as embedded headers in program objects:

```
cl_program foo_pg = clCreateProgramWithSource(context,

1, &foo_header_src, NULL, &err);

cl_program myinc_pg = clCreateProgramWithSource(context,

1, &myinc_header_src, NULL, &err);

// lets assume the program source described above is given

// by program_A and is loaded via clCreateProgramWithSource

cl_program input_headers[2] = \{ foo_pg, myinc_pg };

char * input_header_names[2] = \{ foo.h, mydir/myinc.h };

clCompileProgram(program_A,

0, NULL, // num_devices & device_list

NULL, // compile_options

2, // num_input_headers

input_headers,

input_header_names,

NULL, NULL); // pfn_notify & user_data
```

pfn_notify is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not NULL, **clCompileProgram** does not need to wait for the compiler to complete and can return immediately once the compilation can begin. The compilation can begin if the context, program whose sources are being compiled, list of devices, input headers, programs that describe input headers and compiler options specified are all valid and appropriate host and device resources needed to perform the compile are available. If *pfn_notify* is NULL,

clCompileProgram does not return until the compiler has completed. This callback function may be called asynchronously by the OpenCL implementation. It is the applications 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.

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

- CL_INVALID_PROGRAM if *program* is not a valid program object.
- CL_INVALID_VALUE if *device_list* is NULL and *num_devices* is greater than zero, or if *device_list* is not NULL and *num_devices* is zero.
- CL_INVALID_VALUE if *num_input_headers* is zero and *header_include_names* or *input_headers* are not NULL or if *num_input_headers* is not zero and *header_include_names* or *input_headers* are NULL.
- CL_INVALID_VALUE if *pfn_notify* is NULL but *user_data* is not NULL.
- CL_INVALID_DEVICE if OpenCL devices listed in *device_list* are not in the list of devices associated with *program*
- CL_INVALID_COMPILER_OPTIONS if the compiler options specified by *options* are invalid.
- CL_INVALID_OPERATION if the compilation or build of a program executable for any of the devices listed in *device_list* by a previous call to **clCompileProgram** or **clBuildProgram** for *program* has not completed.
- 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.
- CL_COMPILE_PROGRAM_FAILURE if there is a failure to compile the program source. This error will be returned if **clCompileProgram** does not return until the compile has completed.
- CL_INVALID_OPERATION if there are kernel objects attached to *program*.
- CL_INVALID_OPERATION if *program* has no source or IL available, i.e. it has not been created with **clCreateProgramWithSource** or **clCreateProgramWithIL**.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_program clLinkProgram(cl_context context,
                         cl_uint num_devices,
                         const cl_device_id *device_list,
                         const char *options,
                         cl_uint num_input_programs,
                         const cl_program *input_programs,
                         void (CL_CALLBACK *pfn_notify)
                             (cl_program program,
                             void *user_data),
                         void *user_data,
                         cl_int *errcode_ret)
```

links a set of compiled program objects and libraries for all the devices or a specific device(s) in the OpenCL context and creates a library or executable. **clLinkProgram** creates a new program object which contains the library or executable. The library or executable binary can be queried using **clGetProgramInfo**(*program*, CL_PROGRAM_BINARIES,) and can be specified to **clCreateProgramWithBinary** to create a new program object.

The devices associated with the returned program object will be the list of devices specified by_device_list_ or if *device_list* is NULL it will be the list of devices associated with *context*.

context must be a valid OpenCL context.

device_list is a pointer to a list of devices that are in *context*. If *device_list* is a NULL value, the link is performed for all devices associated with *context* for which a compiled object is available. If *device_list* is a non-NULL value, the link is performed for devices specified in this list for which a compiled object is available.

num_devices_is the number of devices listed in _device_list.

options is a pointer to a null-terminated string of characters that describes the link options to be used for building the program executable. The list of supported options is as described *in section 5.8.7*. If the program was created using clCreateProgramWithBinary and *options* is a NULL pointer, the program will be linked as if options were the same as when the program binary was originally built. If the program was created using clCreateProgramWithBinary and *options* string contains anything other than the same options in the same order (whitespace ignored) as when the program binary was originally built, then the behavior is implementation defined.

num_input_programs specifies the number of programs in array referenced by *input_programs*.

input_programs is an array of program objects that are compiled binaries or libraries that are to be linked to create the program executable. For each device in *device_list* or if *device_list* is NULL the list of devices associated with context, the following cases occur:

- All programs specified by *input_programs* contain a compiled binary or library for the device. In this case, a link is performed to generate a program executable for this device.
- None of the programs contain a compiled binary or library for that device. In this case, no link is performed and there will be no program executable generated for this device.
- All other cases will return a CL_INVALID_OPERATION error.

pfn_notify is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be called when the program executable has been built (successfully or unsuccessfully).

If *pfn_notify* is not NULL, **clLinkProgram** does not need to wait for the linker to complete and can return immediately once the linking operation can begin. Once the linker has completed, the *pfn_notify* callback function is called which returns the program object returned by **clLinkProgram**. The application can query the link status and log for this program object. This callback function may be called asynchronously by the OpenCL implementation. It is the applications responsibility to ensure that the callback function is thread-safe.

If *pfn_notify* is NULL, **clLinkProgram** does not return until the linker has completed.

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

The linking operation can begin if the context, list of devices, input programs and linker options specified are all valid and appropriate host and device resources needed to perform the link are available. If the linking operation can begin, **clLinkProgram** returns a valid non-zero program object.

If *pfn_notify* is NULL, the *errcode_ret* will be set to CL_SUCCESS if the link operation was successful and CL_LINK_FAILURE if there is a failure to link the compiled binaries and/or libraries.

If *pfn_notify* is not NULL, **clLinkProgram** does not have to wait until the linker to complete and can return CL_SUCCESS in *errcode_ret* if the linking operation can begin. The *pfn_notify* callback function will return a CL_SUCCESS or CL_LINK_FAILURE if the linking operation was successful or not.

Otherwise **clLinkProgram** returns a NULL program object with an appropriate error in *errcode_ret*. The application should query the linker status of this program object to check if the link was successful or not. The list of errors that can be returned are:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if *device_list* is NULL and *num_devices* is greater than zero, or if *device_list* is not NULL and *num_devices* is zero.
- CL_INVALID_VALUE if *num_input_programs* is zero and *input_programs* is NULL__or if_num_input_programs_ is zero and *input_programs* is not NULL or if *num_input_programs* is not zero and *input_programs* is NULL.
- CL_INVALID_PROGRAM if programs specified in *input_programs* are not valid program objects.

- CL_INVALID_VALUE if *pfn_notify* is NULL but *user_data* is not NULL.
- CL_INVALID_DEVICE if OpenCL devices listed in *device_list* are not in the list of devices associated with *context*
- CL_INVALID_LINKER_OPTIONS if the linker options specified by *options* are invalid.
- CL_INVALID_OPERATION if the compilation or build of a program executable for any of the devices listed in *device_list* by a previous call to **clCompileProgram** or **clBuildProgram** for *program* has not completed.
- CL_INVALID_OPERATION if the rules for devices containing compiled binaries or libraries as described in *input_programs* argument above are not followed.
- CL_LINKER_NOT_AVAILABLE if a linker is not available i.e. CL_DEVICE_LINKER_AVAILABLE specified in *table 4.3* is set to CL_FALSE.
- CL_LINK_PROGRAM_FAILURE if there is a failure to link the compiled binaries and/or libraries.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.6 Compiler Options

The compiler 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 the compiler when building program executables online or offline from OpenCL C/C++ or, where relevant, from an IL. These may be extended by a set of vendor- or platform-specific options.

5.8.6.1 Preprocessor options

These options control the OpenCL C/C++ preprocessor which is run on each program source before actual compilation. These options are ignored for programs created with IL.

```
-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** or **clCompileProgram**. Note that a space is required between the -D option and the symbol it defines, otherwise behavior is implementation defined.

```
-I _dir_
```

Add the directory *dir* to the list of directories to be searched for header files. *dir* can optionally be enclosed in double quotes.

This option is not portable due to its dependency on host file system and host operating system. It is supported for backwards compatibility with previous OpenCL versions. Developers are encouraged to create and use explicit header objects by means of clCompileProgram followed by clLinkProgram.

5.8.6.2 Math Intrinsics Options

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

```
-cl-single-precision-constant
```

Treat double precision floating-point constant as single precision constant. This option is ignored for programs created with IL.

```
-cl-denorms-are-zero
```

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

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

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

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

```
-cl-fp32-correctly-rounded-divide-sqrt
```

The -cl-fp32-correctly-rounded-divide-sqrt build option to **clBuildProgram** or

clCompileProgram allows an application to specify that single precision floating-point

divide (x/y and 1/x) and sqrt used in the program source are correctly rounded. If

this build option is not specified, te minimum numerical accuracy of single precision

floating-point divide and sqrt are as defined in the SPIR-V OpenCL environment specification.

This build option can only be specified if the CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT is set in CL_DEVICE_SINGLE_FP_CONFIG (as defined in *table 4.3*) for devices that the program is being build. **clBuildProgram** or **clCompileProgram** will fail to compile the program for a device if the -cl-fp32-correctly-rounded-divide-sqrt option is specified and CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT is not set for the device.

5.8.6.3 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.
```

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

```
-cl-mad-enable
```

Allow a * b + c to be replaced by a mad. The mad computes a * b + c with

reduced accuracy. For example, some OpenCL devices implement mad as truncate the

result of a * b before adding it to c.

```
-cl-no-signed-zeros
```

Allow optimizations for floating-point arithmetic that ignore the signedness of zero.

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

```
-cl-unsafe-math-optimizations
```

Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid, (b) may violate IEEE 754 standard and (c) may violate the OpenCL numerical compliance requirements as defined in the SPIR-V OpenCL environment specification for single precision and double precision floating-point, and edge case behavior in the SPIR-V OpenCL environment specification. 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, +Inf, -Inf. This option may violate the OpenCL numerical compliance requirements for single precision and double precision floating-point, as well as edge case behavior. The original and modified values are defined in the SPIR-V OpenCL environment specification

```
-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 for single precision and double precision floating-point, as well as floating point edge case behavior. This option also relaxes the precision of commonly used math functions. This option causes the preprocessor macro *FAST_RELAXED_MATH* to be defined in the OpenCL program. The original and modified values are defined in the SPIR-V OpenCL environment specification

```
-cl-uniform-work-group-size
```

This requires that the global work-size be a multiple of the work-group size specified to **clEnqueueNDRangeKernel**. Allow optimizations that are made possible by this restriction.

```
-cl-no-subgroup-ifp
```

This indicates that kernels in this program do not require subgroups to make independent forward progress. Allows optimizations that are made possible by this restriction. This option has no effect for devices that do not support independent forward progress for subgroups.

5.8.6.4 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. These options are ignored for programs created with IL.

```
-w
```

Inhibit all warning messages.

```
-Werror
```

Make all warnings into errors.

5.8.6.5 Options Controlling the OpenCL C version

The following option controls the version of OpenCL C that the compiler accepts. These options are ignored for programs created with IL.

```
-cl-std=
```

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

- CL1.1 Support all OpenCL C programs that use the OpenCL C language features defined in *section 6* of the OpenCL 1.1 specification.
- CL1.2 Support all OpenCL C programs that use the OpenCL C language features defined in *section 6* of the OpenCL 1.2 specification.
- CL2.0 Support all OpenCL C programs that use the OpenCL C language features defined in *section 6* OpenCL C 2.0 specification.

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

Calls to **clBuildProgram** or **clCompileProgram** with the -cl-std=CL1.2 option **will fail** to compile the program for any devices with CL_DEVICE_OPENCL_C_VERSION = OpenCL C 1.0.

Calls to **clBuildProgram** or **clCompileProgram** with the -cl-std=CL2.0 option **will fail** to compile the program for any devices with CL_DEVICE_OPENCL_C_VERSION = OpenCL C 1.0, OpenCL C 1.1 or OpenCL C 1.2.

If the cl-std build option is not specified, the highest OpenCL C 1.x language version supported by each device is used when compiling the program for each device. Applications are required to specify the cl-std=CL2.0 option if they want to compile or build their programs with OpenCL C 2.0.

5.8.6.6 Options for Querying Kernel Argument Information

```
-cl-kernel-arg-info
```

This option allows the compiler to store information about the arguments of a kernel(s) in

the program executable. The argument information stored includes the argument name,

its type, the address space and access qualifiers used. Refer to description of

clGetKernelArgInfo on how to query this information.

5.8.6.7 Options for debugging your program

The following option is available.

```
-g
```

This option can currently be used to generate additional errors for the built-in functions that allow you to enqueue commands on a device (refer to OpenCL kernel languages specifications).

5.8.7 Linker Options

This specification defines a standard set of linker options that must be supported by the OpenCL C compiler when linking compiled programs online or offline. These linker options are categorized as library linking options and program linking options. These may be extended by a set of vendor- or platform-specific options.

5.8.7.1 Library Linking Options

The following options can be specified when creating a library of compiled binaries.

```
-create-library
```

Create a library of compiled binaries specified in *input_programs* argument to **clLinkProgram**.

```
-enable-link-options
```

Allows the linker to modify the library behavior based on one or more link options

(described in *section 5.8.5.2*) when this library is linked with a program executable. This

option must be specified with the create-library option.

5.8.7.2 Program Linking Options

The following options can be specified when linking a program executable.

```
-cl-denorms-are-zero
-cl-no-signed-zeroes
-cl-unsafe-math-optimizations
-cl-finite-math-only
-cl-fast-relaxed-math
-cl-no-subgroup-ifp
```

The options are described in *section 5.8.4.2* and *section 5.8.4.3*. The linker may apply these options to all compiled program objects specified to **clLinkProgram**. The linker may apply these options only to libraries which were created with the enable-link-option.

5.8.8 Unloading the OpenCL Compiler

The function

```
cl_int  clUnloadPlatformCompiler(cl_platform_id platform)
```

allows the implementation to release the resources allocated by the OpenCL compiler for *platform*. 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**, **clCompileProgram** or **clLinkProgram** after **clUnloadPlatformCompiler** will reload the compiler, if necessary, to build the appropriate program executable.

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

- CL_INVALID_PLATFORM if *platform* is not a valid platform.

5.8.9 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 5.17*.

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 5.17*.

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

Table 5.18: *clGetProgramInfo parameter queries*

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

Table 5.18: (continued)

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. If program is created using clCreateProgramWithBinary, clCreateProgramWithIL or clCreateProgramWithBuiltinKernels, a null string or the appropriate program source code is returned depending on whether or not the program source code is stored in the binary. The actual number of characters that represents the program source code including the null terminator is returned in param_value_size_ret.
CL_PROGRAM_IL	char[]	Returns the program IL for programs created with clCreateProgramWithIL. If program is created with clCreateProgramWithSource, clCreateProgramWithBinary or clCreateProgramWithBuiltinKernels the memory pointed to by param_value will be unchanged and param_value_size_retwill be set to 0.

Table 5.18: (continued)

CL_PROGRAM_BINARY_SIZES	size_t[]	Returns an array that contains the size in bytes of the program binary (could be an executable binary, compiled binary or library 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. If program is created using clCreateProgramWithBuiltinKernels, the implementation may return zero in any entries of the returned array.

Table 5.18: (continued)

CL_PROGRAM_BINARIES	unsigned char *[]	Return the program binaries (could be an executable binary, compiled binary or library binary) 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 or clLinkProgram. If program is created with clCreateProgramWithSource or clCreateProgramWithIL, the binary returned is the binary generated by clBuildProgram, clCompileProgram or clLinkProgram. The bits returned can be an implementation-specific intermediate representation (a.k.a. IR) or device specific executable bits or both. The decision on which information is returned in the binary is up to the OpenCL implementation. param_value points to an array of n pointers allocated by the caller, where n is the number of devices associated with program. The buffer sizes needed to allocate the memory that these n pointers refer to can be queried using the CL_PROGRAM_BINARY_SIZES query as described in this table. Each entry in this array is used by the implementation as the location in memory where to copy the program binary for a specific device, if there is a binary available. To find out which device the program binary in the array refers to, use the CL_PROGRAM_DEVICES query to get the list of devices. There is a one-to-one correspondence between the array of n pointers returned by CL_PROGRAM_BINARIES and array of devices returned by CL_PROGRAM_DEVICES.

Table 5.18: (continued)

CL_PROGRAM_NUM_KERNELS	size_t	Returns the number of kernels declared in *program* that can be created with **clCreateKernel**. This information is only available after a successful program executable has been built for at least one device in the list of devices associated with *program*.
CL_PROGRAM_KERNEL_ NAMES	char[]	Returns a semi-colon separated list of kernel names in *program* that can be created with **clCreateKernel**. This information is only available after a successful program executable has been built for at least one device in the list of devices associated with *program*.
CL_PROGRAM_SCOPE_ GLOBAL_CTORS_PRESENT	cl_bool	This indicates that the *program* object contains non-trivial constructor(s) that will be executed by runtime before any kernel from the program is executed.
CL_PROGRAM_SCOPE_ GLOBAL_DTORS_PRESENT	cl_bool	This indicates that the program object contains non-trivial destructor(s) that will be executed by runtime when *program* is destroyed.

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.17_and _param_value* is not NULL.
- CL_INVALID_PROGRAM if *program* is a not a valid program object.
- CL_INVALID_PROGRAM_EXECUTABLE if *param_name* is CL_PROGRAM_NUM_KERNELS or CL_PROGRAM_KERNEL_NAMES and a successful program executable has not been built for at least one device in the list of devices associated with *program*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clGetProgramBuildInfo(cl_program program,
                             cl_device_id device,
                             cl_program_build_info param_name,
```

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

```
                                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 5.18*.

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 5.18*.

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

Table 5.19: *clGetProgramBuildInfo parameter queries.*

cl_program_build_info	**Return Type**	**Info. returned in *param_value***

Table 5.19: (continued)

CL_PROGRAM_BUILD_ STATUS	cl_build_status	Returns the build, compile or link status, whichever was performed last on program for device. This can be one of the following: CL_BUILD_NONE. The build status returned if no clBuildProgram, clCompileProgram or clLinkProgram has been performed on the specified program object for device. CL_BUILD_ERROR. The build status returned if clBuildProgram, clCompileProgram or clLinkProgram whichever was performed last on the specified program object for device generated an error. CL_BUILD_SUCCESS. The build status returned if clBuildProgram, clCompileProgram or clLinkProgram whichever was performed last on the specified program object for device was successful. CL_BUILD_IN_PROGRESS. The build status returned if clBuildProgram, clCompileProgram or clLinkProgram whichever was performed last on the specified program object for device has not finished.
CL_PROGRAM_BUILD_ OPTIONS	char[]	Return the build, compile or link options specified by the options argument in clBuildProgram, clCompileProgram or clLinkProgram, whichever was performed last on program for device. If build status of program for device is CL_BUILD_NONE, an empty string is returned.

Table 5.19: (continued)

CL_PROGRAM_BUILD_ LOG	char[]	Return the build or compile log for clBuildProgram or clCompileProgram whichever was performed last on program for device. If build status of program for device is CL_BUILD_NONE, an empty string is returned.
CL_PROGRAM_BINARY_ TYPE	cl_program_ binary_type	Return the program binary type for device. This can be one of the following values: CL_PROGRAM_BINARY_TYPE_NONE – There is no binary associated with device. CL_PROGRAM_BINARY_TYPE_ COMPILED_OBJECT – A compiled binary is associated with device. This is the case if program was created using clCreateProgramWithSource and compiled using clCompileProgram or a compiled binary is loaded using clCreateProgramWithBinary. CL_PROGRAM_BINARY_TYPE_ LIBRARY – A library binary is associated with device. This is the case if program was created by clLinkProgram which is called with the – create-library link option or if a library binary is loaded using clCreateProgramWithBinary. CL_PROGRAM_BINARY_TYPE_ EXECUTABLE – An executable binary is associated with device. This is the case if program was created by clLinkProgram without the –create-library link option or program was created by clBuildProgram or an executable binary is loaded using clCreateProgramWithBinary.
CL_PROGRAM_BUILD_ GLOBAL_VARIABLE_ TOTAL_SIZE	size_t	The total amount of storage, in bytes, used by program variables in the global address space.

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

- CL_INVALID_DEVICE if *device* is not in the list of devices associated with *program*.
- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.18* and *param_value* is not NULL.
- CL_INVALID_PROGRAM if *program* is a not a valid program object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

NOTE:

A program binary (compiled binary, library binary or executable binary) built for a parent device can be used by all its sub-devices. If a program binary has not been built for a sub-device, the program binary associated with the parent device will be used.

A program binary for a device specified with **clCreateProgramWithBinary** or queried using **clGetProgramInfo** can be used as the binary for the associated root device, and all sub-devices created from the root-level device or sub-devices thereof.

5.9 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.

5.9.1 Creating Kernel Objects

To create a kernel object, use the function

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

program is a program object with a successfully built executable.

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

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

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

- CL_INVALID_PROGRAM if _program_is not a valid program object.
- CL_INVALID_PROGRAM_EXECUTABLE if there is no successfully built executable for *program*.
- CL_INVALID_KERNEL_NAME if *kernel_name* is not found in *program*.
- CL_INVALID_KERNEL_DEFINITION if the function definition for *_kernel function given by _kernel_name* such as the number of arguments, the argument types are not the same for all devices for which the *program* executable has been built.
- CL_INVALID_VALUE if *kernel_name* is NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

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

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

program is a program object with a successfully built executable.

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

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

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

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

- CL_INVALID_PROGRAM if *program* is not a valid program object.
- CL_INVALID_PROGRAM_EXECUTABLE if there is no successfully built executable for any device in *program*.
- CL_INVALID_VALUE if *kernels* is not NULL and *num_kernels* is less than the number of kernels in *program*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Kernel objects can only be created once you have a program object with a valid program source or binary loaded into the program object and the program executable has been successfully built for one or more devices associated with program. No changes to the program executable are allowed while there are kernel objects associated with a program object. This means that calls to **clBuildProgram** and **clCompileProgram** return CL_INVALID_OPERATION if there are kernel objects attached to a program object. The OpenCL context associated with *program* will be the context associated with *kernel*. The list of devices associated with *program* are the devices associated with *kernel*. Devices associated with a program object for which a valid program executable has been built can be used to execute kernels declared in the program object.

The function

```
cl_int clRetainKernel(cl_kernel kernel)
```

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

- CL_INVALID_KERNEL if *kernel* is not a valid kernel object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

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

The function

```
cl_int clReleaseKernel(cl_kernel kernel)
```

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

- CL_INVALID_KERNEL if *kernel* is not a valid kernel object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The kernel object is deleted once the number of instances that are retained to *kernel* become zero and the kernel object is no longer needed by any enqueued commands that use *kernel*. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainKernel** causes undefined behavior.

5.9.2 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**) until the argument value is changed by a call to **clSetKernelArg** for *kernel*.

If the argument is a memory object (buffer, pipe, image or image array), the *arg_value* entry will be a pointer to the appropriate buffer, pipe, image or image array object. The memory object must be created with the context associated with the kernel object. If the argument is a buffer object, the *arg_value* pointer can be NULL or point to a NULL value in which case a NULL value will be used as the value for the argument declared as a pointer to global or constant memory in the kernel. If the argument is declared with the local qualifier, the *arg_value* entry must be NULL. If the argument is of type *sampler_t*, the *arg_value* entry must be a pointer to the sampler object. If the argument is of type *queue_t*, the *arg_value* entry must be a pointer to the device queue object.

If the argument is declared to be a pointer of a built-in scalar or vector type, or a user defined structure type in the global or constant address space, the memory object specified as argument value must be a buffer object (or NULL). If the argument

is declared with the constant qualifier, the size in bytes of the memory object cannot exceed CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE and the number of arguments declared as pointers to *constant* memory cannot exceed CL_DEVICE_MAX_CONSTANT_ARGS.

The memory object specified as argument value must be a pipe object if the argument is declared with the *pipe* qualifier.

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 2D image object with image channel order = CL_DEPTH if the argument is declared to be of type *image2d_depth_t*. The memory object specified as argument value must be a 3D image object if argument is declared to be of type *image3d_t*. The memory object specified as argument value must be a 1D image object if the argument is declared to be of type *image1d_t*. The memory object specified as argument value must be a 1D image buffer object if the argument is declared to be of type *image1d_buffer_t*. The memory object specified as argument value must be a 1D image array object if argument is declared to be of type *image1d_array_t*. The memory object specified as argument value must be a 2D image array object if argument is declared to be of type *image2d_array_t*. The memory object specified as argument value must be a 2D image array object with image channel order = CL_DEPTH if argument is declared to be of type *image2d_array_depth_t*.

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

arg_size specifies the size of the argument value. If the argument is a memory object, the size is the size of the memory object. 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). If the argument is of type *queue_t*, the *arg_size* value must be equal to sizeof(cl_command_queue). For all other arguments, the size will be the size of argument type.

Note

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

---

[a] 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 safel y release user allocated resources associated with OpenCL objects such as the cl_mem backing store used with CL_MEM_USE_HOST_PTR.

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 not a valid value.
- 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_DEVICE_QUEUE for an argument declared to be of type *queue_t* when the specified *arg_value* is not a valid device queue 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).
- CL_MAX_SIZE_RESTRICTION_EXCEEDED if the size in bytes of the memory object (if the argument was declared with constant qualifier) or *arg_size* (if the argument was declared with local qualifier) exceed the maximum size restriction that was set with the optional language attribute. The optional attribute can be cl::max_size defined in OpenCL 2.2 C++ Kernel Languange specification or SpvDecorationMaxByteOffset defined in SPIR-V 1.2 Specification.

- CL_INVALID_ARG_VALUE if the argument is an image declared with the read_only qualifier and *arg_value* refers to an image object created with *cl_mem_flags* of CL_MEM_WRITE or if the image argument is declared with the write_only qualifier and *arg_value* refers to an image object created with *cl_mem_flags* of CL_MEM_READ.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int  clSetKernelArgSVMPointer(cl_kernel kernel,
                                 cl_uint arg_index,
                                 const void *arg_value)
```

is used to set a SVM pointer as 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.

arg_value is the SVM pointer that should be used as the argument value for argument specified by *arg_index*. The SVM pointer specified is the value used by all API calls that enqueue *kernel* (**clEnqueueNDRangeKernel**) until the argument value is changed by a call to **clSetKernelArgSVMPointer** for *kernel*. The SVM pointer can only be used for arguments that are declared to be a pointer to global or constant memory. The SVM pointer value must be aligned according to the arguments type. For example, if the argument is declared to be global float4 p, the SVM pointer value passed for p must be at a minimum aligned to a float4. The SVM pointer value specified as the argument value can be the pointer returned by **clSVMAlloc** or can be a pointer offset into the SVM region.

clSetKernelArgSVMPointer 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 not a valid value.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int  clSetKernelExecInfo(cl_kernel kernel,
                            cl_kernel_exec_info param_name,
                            size_t param_value_size,
                            const void *param_value)
```

can be used to pass additional information other than argument values to a kernel.

kernel specifies the kernel object being queried.

param_name specifies the information to be passed to kernel. The list of supported *param_name* types and the corresponding values passed in *param_value* is described in *table 5.19*.

param_value_size specifies the size in bytes of the memory pointed to by *param_value*.

param_value is a pointer to memory where the appropriate values determined by *param_name* are specified.

Table 5.20: *clSetKernelExecInfo parameter values.*

cl_kernel_exec_info	Type	Description
CL_KERNEL_EXEC_INFO_ SVM_PTRS	void *[]	SVM pointers must reference locations contained entirely within buffers that are passed to kernel as arguments, or that are passed through the execution information. Non-argument SVM buffers must be specified by passing pointers to those buffers via clSetKernelExecInfo for coarse-grain and fine-grain buffer SVM allocations but not for finegrain system SVM allocations.
CL_KERNEL_EXEC_INFO_ SVM_FINE_GRAIN_SYSTEM	cl_bool	This flag indicates whether the kernel uses pointers that are fine grain system SVM allocations. These fine grain system SVM pointers may be passed as arguments or defined in SVM buffers that are passed as arguments to *kernel*.

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

- CL_INVALID_KERNEL if *kernel* is a not a valid kernel object.
- CL_INVALID_VALUE if *param_name* is not valid, if *param_value* is NULL or if the size specified by *param_value_size* is not valid.
- CL_INVALID_OPERATION if *param_name* = CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM and *param_value* = CL_TRUE but no devices in context associated with *kernel* support fine-grain system SVM allocations.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

NOTES

Coarse-grain or fine-grain buffer SVM pointers used by a kernel which are not passed as a kernel arguments must be specified using **clSetKernelExecInfo** with CL_KERNEL_EXEC_INFO_SVM_PTRS. For example, if SVM buffer A contains a pointer to another SVM buffer B, and the kernel dereferences that pointer, then a pointer to B must either be passed as an argument in the call to that kernel or it must be made available to the kernel using **clSetKernelExecInfo**. For example, we might pass extra SVM pointers as follows:

```
clSetKernelExecInfo(kernel,
                    CL_KERNEL_EXEC_INFO_SVM_PTRS,
                    num_ptrs * sizeof(void *),
                    extra_svm_ptr_list);
```

Here num_ptrs specifies the number of additional SVM pointers while extra_svm_ptr_list specifies a pointer to memory containing those SVM pointers.

When calling **clSetKernelExecInfo** with CL_KERNEL_EXEC_INFO_SVM_PTRS to specify pointers to non-argument SVM buffers as extra arguments to a kernel, each of these pointers can be the SVM pointer returned by **clSVMAlloc** or can be a pointer + offset into the SVM region. It is sufficient to provide one pointer for each SVM buffer used.

CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM is used to indicate whether SVM pointers used by a kernel will refer to system allocations or not.

CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM = CL_FALSE indicates that the OpenCL implementation may assume that system pointers are not passed as kernel arguments and are not stored inside SVM allocations passed as kernel arguments.

CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM = CL_TRUE indicates that the OpenCL implementation must assume that system pointers might be passed as kernel arguments and/or stored inside SVM allocations passed as kernel arguments. In this case, if the device to which the kernel is enqueued does not support system SVM pointers, **clEnqueueNDRangeKernel** will return a CL_INVALID_OPERATION error. If none of the devices in the context associated with kernel support fine-grain system SVM allocations, **clSetKernelExecInfo** will return a CL_INVALID_OPERATION error.

If **clSetKernelExecInfo** has not been called with a value for **CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM**, the default value is used for this kernel attribute. The default value depends on whether the device on which the kernel is enqueued supports fine-grain system SVM allocations. If so, the default value used is CL_TRUE (system pointers might be passed); otherwise, the default is CL_FALSE.

A call to **clSetKernelExecInfo** for a given value of *param_name* replaces any prior value passed for that value of *param_name*. Only one *param_value* will be stored for each value of *param_name*.

5.9.3 Copying Kernel Objects

The function

```
cl_kernel clCloneKernel(cl_kernel source_kernel,
                        cl_int *errcode_ret)
```

is used to make a shallow copy of the kernel object, its arguments and any information passed to the kernel object using **clSetKernelExecInfo**. If the kernel object was ready to be enqueued before copying it, the clone of the kernel object is ready to enqueue.

source_kernel is a valid cl_kernel object that will be copied. _source_kernel_will not be modified in any way by this function.

errcode_ret will be assigned an appropriate error code. If _errcode_ret_is NULL, no error code is returned.

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

- CL_INVALID_KERNEL if *kernel* is not a valid kernel object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The returned kernel object is an exact copy of *source_kernel*, with one caveat: the reference count on the returned kernel object is set as if it had been returned by **clCreateKernel**. The reference count of *source_kernel will* not be changed.

The resulting kernel will be in the same state as if **clCreateKernel** is called to create the resultant kernel with the same arguments as those used to create *source_kernel*, the latest call to **clSetKernelArg** or **clSetKernelArgSVMPointer** for each argument index applied to kernel and the last call to **clSetKernelExecInfo** for each value of the param name parameter are applied to the new kernel object.

All arguments of the new kernel object must be intact and it may be correctly used in the same situations as kernel except those that assume a pre-existing reference count. Setting arguments on the new kernel object will not affect *source_kernel* except insofar as the argument points to a shared underlying entity and in that situation behavior is as if two kernel objects had been created and the same argument applied to each. Only the data stored in the kernel object is copied; data referenced by the kernels arguments are not copied. For example, if a buffer or pointer argument is set on a kernel object, the pointer is copied but the underlying memory allocation is not.

5.9.4 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 5.20*.

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 5.20*.

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

Table 5.21: *clGetKernelInfo parameter queries.*

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[15]:	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 5.21: (continued)

CL_KERNEL_ATTRIBUTES	char[]	Returns any attributes specified using the *attribute* OpenCL Cqualifier (or using an OpenCL C++ qualifier syntax [[]]) with the kernel function declaration in the program source. These attributes include attributes described in the earlier OpenCL C kernel language specifications and other attributes supported by an implementation. Attributes are returned as they were declared inside *attribute*..., with any surrounding whitespace and embedded newlines removed. When multiple attributes are present, they are returned as a single, space delimited string. For kernels not created from OpenCL C source and the clCreateProgramWithSource API call the string returned from this query will be empty.

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

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.20_and _param_value* is not NULL.
- CL_INVALID_KERNEL if *kernel* is a not a valid kernel object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

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

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

kernel specifies the kernel object being queried.

[15] 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.

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 5.21*.

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 5.21*.

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

Table 5.22: *clGetKernelWorkGroupInfo parameter queries.*

cl_kernel_work_group_info	**Return Type**	**Info. returned in *param_value***
CL_KERNEL_GLOBAL_ WORK_SIZE	size_t[3]	This provides a mechanism for the application to query the maximum global size that can be used to execute a kernel (i.e. global_work_size argument to clEnqueueNDRangeKernel) on a custom device given by device or a built-in kernel on an OpenCL device given by device. If device is not a custom device and kernel is not a built-in kernel, clGetKernelWorkGroupInfo returns the error CL_INVALID_VALUE.
CL_KERNEL_WORK_ GROUP_SIZE	size_t	This provides a mechanism for the application to query the maximum workgroup size that can be used to execute the 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. As a result and unlike CL_DEVICE_MAX_WORK_GROUP_ SIZE this value may vary from one kernel to another as well as one device to another. CL_KERNEL_WORK_GROUP_SIZE will be less than or equal to CL_DEVICE_MAX_WORK_GROUP_SI ZE for a given kernel object.

Table 5.22: (continued)

CL_KERNEL_COMPILE_ WORK_GROUP_SIZE	size_t[3]	Returns the work-group size specified in the kernel source or IL. If the work-group size is not specified in the kernel source or IL, (0, 0, 0) is returned.
CL_KERNEL_LOCAL_ MEM_SIZE	cl_ulong	Returns the amount of local memory in bytes being used by a kernel. This includes local memory that may be needed by an implementation to execute the kernel, variables declared inside the kernel with the *local address qualifier and local memory to be allocated for arguments to the kernel declared as pointers with the* local address qualifier and whose size is specified with clSetKernelArg. If the local memory size, for any pointer argument to the kernel declared with the __local address qualifier, is not specified, its size is assumed to be 0.
CL_KERNEL_PREFERRED_ WORK_GROUP_SIZE_MULTIPLE	size_t	Returns the preferred multiple of work-group size for launch. This is a performance hint. Specifying a work-group size that is not a multiple of the value returned by this query as the value of the local work size argument to **clEnqueueNDRangeKernel** will not fail to enqueue the kernel for execution unless the work-group size specified is larger than the device maximum.
CL_KERNEL_PRIVATE_ MEM_SIZE	cl_ulong	Returns the minimum amount of private memory, in bytes, used by each work-item in the kernel. This value may include any private memory needed by an implementation to execute the kernel, including that used by the language built-ins and variable declared inside the kernel with the __private qualifier.

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

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

more than one device associated with *kernel*.

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.21_and _param_value* is not NULL.
- CL_INVALID_VALUE if *param_name* is CL_KERNEL_GLOBAL_WORK_SIZE and *device* is not a custom device and *kernel* is not a built-in kernel.
- CL_INVALID_KERNEL if *kernel* is a not a valid kernel object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clGetKernelSubGroupInfo(cl_kernel kernel,
                               cl_device_id device,
                               cl_kernel_sub_group_info param_name,
                               size_t input_value_size,
                               const void *input_value,
                               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.

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 **clGetKernelSubGroupInfo** is described in *table 5.22*.

input_value_size is used to specify the size in bytes of memory pointed to by *input_value*. This size must be == size of input type as described in the table below.

input_value is a pointer to memory where the appropriate parameterization of the query is passed from. If *input_value* is NULL, it is ignored.

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 5.22*.

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

Table 5.23: *clGetKernelSubGroupInfo parameter queries.*

cl_kernel_sub_group_info	**Input Type**	**Return Type**	**Info. returned in *param_value***

Table 5.23: (continued)

CL_KERNEL_MAX_ SUB_GROUP_SIZE_ FOR_NDRANGE	size_t *	size_t	Returns the maximum sub-group size for this kernel. All sub-groups must be the same size, while the last subgroup in any work-group (i.e. the subgroup with the maximum index) could be the same or smaller size. The input_value must be an array of size_t values corresponding to the local work size parameter of the intended dispatch. The number of dimensions in the ND-range will be inferred from the value specified for input_value_size.
CL_KERNEL_SUB_ GROUP_COUNT_ FOR_NDRANGE	size_t *	size_t	Returns the number of sub-groups that will be present in each work-group for a given local work size. All workgroups, apart from the last work-group in each dimension in the presence of non-uniform work-group sizes, will have the same number of sub-groups. The input_value must be an array of size_t values corresponding to the local work size parameter of the intended dispatch. The number of dimensions in the ND-range will be inferred from the value specified for input_value_size.

Table 5.23: (continued)

CL_KERNEL_LOCAL_ SIZE_FOR_SUB_ GROUP_COUNT	size_t	size_t[]	Returns the local size that will generate the requested number of sub-groups for the kernel. The output array must be an array of size_t values corresponding to the local size parameter. Any returned work-group will have one dimension. Other dimensions inferred from the value specified for param_value_size will be filled with the value 1. The returned value will produce an exact number of sub-groups and result in no partial groups for an executing kernel except in the case where the last work-group in a dimension has a size different from that of the other groups. If no work-group size can accommodate the requested number of sub-groups, 0 will be returned in each element of the return array.
CL_KERNEL_MAX_ NUM_SUB_GROUPS	ignored	size_t	This provides a mechanism for the application to query the maximum number of sub-groups that may make up each work-group 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. The returned value may be used to compute a work-group size to enqueue the kernel with to give a round number of sub-groups for an enqueue.

Table 5.23: (continued)

CL_KERNEL_ COMPILE_NUM_ SUB_GROUPS	ignored	size_t	Returns the number of sub-groups specified in the kernel source or IL. If the sub-group count is not specified using the above attribute then 0 is returned.

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

- CL_INVALID_DEVICE if *device* is not in the list of devices associated with *kernel* or if *device* is NULL but there is more than one device associated with *kernel*.
- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.22* and *param_value* is not NULL.
- CL_INVALID_VALUE if *param_name* is CL_KERNEL_SUB_GROUP_SIZE_FOR_NDRANGE and the size in bytes specified by *input_value_size* is not valid or if *input_value* is NULL.
- CL_INVALID_KERNEL if *kernel* is a not a valid kernel object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clGetKernelArgInfo(cl_kernel kernel,
                          cl_uint arg_indx,
                          cl_kernel_arg_info param_name,
                          size_t param_value_size,
                          void *param_value,
                          size_t *param_value_size_ret)
```

returns information about the arguments of a kernel.

Kernel argument information is only available if the program object associated with *kernel* is created with **clCreateProgramWithSource** and the program executable was built with the -cl-kernel-arg-info option specified in options argument to **clBuildProgram** or **clCompileProgram**.

kernel specifies the kernel object being queried.

arg_indx 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.

param_name specifies the argument information to query. The list of supported *param_name* types and the information returned in *param_value_by* ***clGetKernelArgInfo*** *is described in _table__5.23*.

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__5.23*.

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

Table 5.24: *clGetKernelArgInfo parameter queries.*

cl_kernel_arg_info	Return Type	Info. returned in *param_value*
CL_KERNEL_ARG_ ADDRESS_QUALIFIER	cl_kernel_arg_ address_qualifier	Returns the address qualifier specified for the argument given by arg_indx. This can be one of the following values: CL_KERNEL_ARG_ADDRESS_ GLOBAL CL_KERNEL_ARG_ADDRESS_ LOCAL CL_KERNEL_ARG_ADDRESS_ CONSTANT CL_KERNEL_ARG_ADDRESS_ PRIVATE If no address qualifier is specified, the default address qualifier which is CL_KERNEL_ARG_ADDRESS_PRIVATE is returned.
CL_KERNEL_ARG_ ACCESS_QUALIFIER	cl_kernel_arg_ access_qualifier	Returns the access qualifier specified for the argument given by arg_indx. This can be one of the following values: CL_KERNEL_ARG_ACCESS_ READ_ONLY CL_KERNEL_ARG_ACCESS_ WRITE_ONLY CL_KERNEL_ARG_ACCESS_ READ_WRITE CL_KERNEL_ARG_ACCESS_ NONE If argument is not an image type and is not declared with the pipe qualifier, CL_KERNEL_ARG_ACCESS_NONE is returned. If argument is an image type, the access qualifier specified or the default access qualifier is returned.
CL_KERNEL_ARG_TYPE_ NAME	char[]	Returns the type name specified for the argument given by *arg_indx*. The type name returned will be the argument type name as it was declared with any whitespace removed. If argument type name is an unsigned scalar type (i.e. unsigned char, unsigned short, unsigned int, unsigned long), uchar, ushort, uint and ulong will be returned. The argument type name returned does not include any type qualifiers.

Table 5.24: (continued)

CL_KERNEL_ARG_TYPE_ QUALIFIER	cl_kernel_arg_ type_qualifier	Returns the type qualifier specified for the argument given by arg_indx. The returned value can be: CL_KERNEL_ARG_TYPE_ CONST CL_KERNEL_ARG_TYPE_ RESTRICT CL_KERNEL_ARG_TYPE_ VOLATILE, a combination of the above enums, CL_KERNEL_ARG_TYPE_PIPE or CL_KERNEL_ARG_TYPE_NONE NOTE: CL_KERNEL_ARG_ TYPE_VOLATILE is returned if the argument is a pointer and the referenced type is declared with the volatile qualifier. For example, a kernel argument declared as global int volatile *x returns CL_KERNEL_ARG_TYPE_ VOLATILE but a kernel argument declared as global int * volatile x does not. Similarly, CL_KERNEL_ARG_TYPE_CONST is returned if the argument is a pointer and the referenced type is declared with the restrict or const qualifier. For example, a kernel argument declared as global int const *x returns CL_KERNEL_ARG_TYPE_CONST but a kernel argument declared as global int * const x does not. CL_KERNEL_ARG_TYPE_ RESTRICT will be returned if the pointer type is marked restrict. For example, global int * restrict x returns CL_KERNEL_ARG_TYPE_ RESTRICT. If the argument is declared with the constant address space qualifier, the CL_KERNEL_ARG_TYPE_CONST type qualifier will be set. CL_KERNEL_ARG_TYPE_NONE is returned for all parameters passed by value.
CL_KERNEL_ARG_NAME	char[]	Returns the name specified for the argument given by *arg_indx*.

clGetKernelArgInfo returns CL SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_ARG_INDEX if *arg_indx* is not a valid argument index.
- 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 5.23_ and *param_value* is not NULL.
- CL_KERNEL_ARG_INFO_NOT_AVAILABLE if the argument information is not available for kernel.
- CL_INVALID_KERNEL if *kernel* is a not a valid kernel object.

5.10 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 host command-queue. The kernel will be queued for execution on the device associated with *command_queue*.

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

work_dim is the number of dimensions used to specify the global work-items and work-items in the work-group. *work_dim* must be greater than zero and less than or equal to CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS. If *global_work_size_is NULL, or the value in any passed dimension is 0 then the kernel command will trivially succeed after its event dependencies are satisfied and subsequently update its completion event. The behavior in this situation is similar to that of an enqueued marker, except that unlike a marker, an enqueued kernel with no events passed to _event_wait_list* may run at any time.

global_work_offset can be used to specify an array of *work_dim* unsigned values that describe the offset used to calculate the global ID of a work-item. If *global_work_offset* is NULL, the global IDs start at offset (0, 0, 0).

global_work_size 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_KERNEL_WORK_GROUP_SIZE value specified in *table 5.21* 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.

Enabling non-uniform work-groups requires the *kernel*'s program to be compiled without the -cl-uniform-work-group-size flag. If the program was created with clCreateProgramWithSource, non-uniform work-groups are enabled only if the

program was compiled with the -cl-std=CL2.0 flag and without the -cl-uniform-work-group-size flag. If the program was created using clLinkProgram and any of the linked programs were compiled in a way that only supports uniform work-group sizes, the linked program only supports uniform work group sizes. If *local_work_size* is specified and the OpenCL *kernel* is compiled without non-uniform work-groups enabled, the values specified in *global_work_size*[0], . . . , *global_work_size*[*work_dim* - 1] must be evenly divisible by the corresponding values specified in *local_work_size*[0], . . . , *local_work_size*[*work_dim* - 1].

If non-uniform work-groups are enabled for the kernel, any single dimension for which the global size is not divisible by the local size will be partitioned into two regions. One region will have work-groups that have the same number of work items as was specified by the local size parameter in that dimension. The other region will have work-groups with less than the number of work items specified by the local size parameter in that dimension. The global IDs and group IDs of the work items in the first region will be numerically lower than those in the second, and the second region will be at most one work-group wide in that dimension. Work-group sizes could be non-uniform in multiple dimensions, potentially producing work-groups of up to 4 different sizes in a 2D range and 8 different sizes in a 3D range.

If *local_work_size_is NULL and the kernel is compiled without support for non-uniform work-groups, the OpenCL runtime will implement the ND-range with uniform work-group sizes. If _local_work_size* is NULL and non-uniform-work-groups are enabled, the OpenCL runtime is free to implement the ND-range using uniform or non-uniform work-group sizes, regardless of the divisibility of the global work size. If the ND-range is implemented using non-uniform work-group sizes, the work-group sizes, global IDs and group IDs will follow the same pattern as described in above paragraph.

The work-group size to be used for *kernel* can also be specified in the program source or intermediate language. In this case the size of work group specified by *local_work_size* must match the value specified in the program source.

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

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

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

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

clEnqueueNDRangeKernel returns CL_SUCCESS if the kernel-instance 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 host command-queue.
- CL_INVALID_KERNEL if *kernel* is not a valid kernel object.
- CL_INVALID_CONTEXT if context associated with *command_queue* and *kernel* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_KERNEL_ARGS if the kernel argument values have not been specified or if a kernel argument declared to be a pointer to a type does not point to a named address space.
- CL_INVALID_WORK_DIMENSION if *work_dim* is not a valid value (i.e. a value between 1 and 3).

- CL_INVALID_GLOBAL_WORK_SIZE if any of the values specified in *global_work_size*[0], *global_work_size*[*work_dim* 1] exceed the maximum value representable by size_t on the device on which the kernel-instance will be enqueued.
- CL_INVALID_GLOBAL_OFFSET if the value specified in *global_work_size* + the corresponding values in *global_work_offset* for any dimensions is greater than the maximum value representable by size t on the device on which the kernel-instance will be enqueued.
- CL_INVALID_WORK_GROUP_SIZE if *local_work_size* is specified and does not match the required work-group size for *kernel* in the program source.
- CL_INVALID_WORK_GROUP_SIZE if *local_work_size* is specified and is not consistent with the required number of sub-groups for *kernel* in the program source.
- CL_INVALID_WORK_GROUP_SIZE if *local_work_size* is specified and the total number of work-items in the work-group computed as *local_work_size*[0] * *local_work_size*[*work_dim* 1] is greater than the value specified by CL_KERNEL_WORK_GROUP_SIZE in *table 5.21*.
- CL_INVALID_WORK_GROUP_SIZE if the program was compiled with cl-uniform-work-group-size and the number of work-items specified by *global_work_size* is not evenly divisible by size of work-group given by *local_work_size* or by the required work-group size specified in the kernel source.
- CL_INVALID_WORK_ITEM_SIZE if the number of work-items specified in any of *local_work_size*[0], *local_work_size*[*work_dim* 1] is greater than the corresponding values specified by CL_DEVICE_MAX_WORK_ITEM_SIZES[0], . CL_DEVICE_MAX_WORK_ITEM_SIZES[*work_dim* 1].
- CL_MISALIGNED_SUB_BUFFER_OFFSET if a sub-buffer object is specified as the value for an argument that is a buffer object and the *offset* specified when the sub-buffer object is created is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN value for device associated with *queue*.
- CL_INVALID_IMAGE_SIZE if an image object is specified as an argument value and the image dimensions (image width, height, specified or compute row and/or slice pitch) are not supported by device associated with *queue*.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if an image object is specified as an argument value and the image format (image channel order and data type) is not supported by device associated with *queue*.
- CL_OUT_OF_RESOURCES if there is a failure to queue the execution instance of *kernel* on the command-queue because of insufficient resources needed to execute the kernel. For example, the explicitly specified *local_work_size* causes a failure to execute the kernel because of insufficient resources such as registers or local memory. Another example would be the number of read-only image args used in *kernel* exceed the CL_DEVICE_MAX_READ_IMAGE_ARGS value for device or the number of write-only and read-write image args used in *kernel* exceed the CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS value for device or the number of samplers used in *kernel* exceed CL_DEVICE_MAX_SAMPLERS for device.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with image or buffer objects specified as arguments to *kernel*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_INVALID_OPERATION if SVM pointers are passed as arguments to a kernel and the device does not support SVM or if system pointers are passed as arguments to a kernel and/or stored inside SVM allocations passed as kernel arguments and the device does not support fine grain system SVM allocations.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clEnqueueNativeKernel(cl_command_queue command_queue,
                             void (CL_CALLBACK *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 host 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 host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_VALUE if *user_func* is NULL.
- CL_INVALID_VALUE if *args* is a NULL value and *cb_args* > 0, or if *args* is a NULL value and *num_mem_objects* > 0.
- CL_INVALID_VALUE if *args* is not NULL and *cb_args* is 0.
- CL_INVALID_VALUE if *num_mem_objects* > 0 and *mem_list* or *args_mem_loc* are NULL.
- CL_INVALID_VALUE if *num_mem_objects* = 0 and *mem_list* or *args_mem_loc* are not NULL.
- CL_INVALID_OPERATION if the device associated with *command_queue* cannot execute the native kernel.
- CL_INVALID_MEM_OBJECT if one or more memory objects specified in *mem_list* are not valid or are not buffer objects.
- CL_OUT_OF_RESOURCES if there is a failure to queue the execution instance of *kernel* on the command-queue because of insufficient resources needed to execute the kernel.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with buffer objects specified as arguments to *kernel*.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.

- CL_INVALID_OPERATION if SVM pointers are passed as arguments to a kernel and the device does not support SVM or if system pointers are passed as arguments to a kernel and/or stored inside SVM allocations passed as kernel arguments and the device does not support fine grain system SVM allocations.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

NOTE:

The total number of read-only images specified as arguments to a kernel cannot exceed CL_DEVICE_MAX_READ_IMAGE_ARGS. Each image array argument to a kernel declared with the read_only qualifier counts as one image. The total number of write-only images specified as arguments to a kernel cannot exceed CL_DEVICE_MAX_WRITE_IMAGE_ARGS. Each image array argument to a kernel declared with the write_only qualifier counts as one image.

The total number of read-write images specified as arguments to a kernel cannot exceed CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS. Each image array argument to a kernel declared with the read_write qualifier counts as one image.

5.11 Event Objects

Event objects can be used to refer to a kernel-instance command (**clEnqueueNDRangeKernel, clEnqueueNativeKernel**), read, write, map and copy commands on memory objects (**clEnqueue{Read|Write|Map}Buffer, clEnqueueUnmapMemObject**, **clEnqueue{Read|Write}BufferRect**, **clEnqueue{Read|Write|Map}Image**, **clEnqueueCopy{Buffer|Image}**, **clEnqueueCopyBufferRect**, **clEnqueueCopyBufferToImage**, **clEnqueueCopyImageToBuffer), clEnqueueSVMMemcpy**, **clEnqueueSVMMemFill**, **clEnqueueSVMMap**, **clEnqueueSVMUnmap**, **clEnqueueSVMFree**, **clEnqueueMarkerWithWaitList**,**clEnqueueBarrierWithWaitList**(refer to *section 5.12*) or user events.

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

The execution status of an enqueued command at any given point in time can be one of the following:

- CL_QUEUED This indicates that the command has been enqueued in a command-queue. This is the initial state of all events except user events.
- CL_SUBMITTED This is the initial state for all user events. For all other events, this indicates that the command has been submitted by the host to the device.
- CL_RUNNING This indicates that the device has started executing this command. In order for the execution status of an enqueued command to change from CL_SUBMITTED to CL_RUNNING, all events that this command is waiting on must have completed successfully i.e. their execution status must be CL_COMPLETE.
- CL_COMPLETE This indicates that the command has successfully completed.
- Error code The error code is a negative integer value and indicates that the command was abnormally terminated. Abnormal termination may occur for a number of reasons such as a bad memory access.

Note
A command is considered to be complete if its execution status is CL_COMPLETE or is a negative integer value.

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

The function

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

creates a user event object. User events allow applications to enqueue commands that wait on a user event to finish before the command is executed by the device.

context must be a valid OpenCL context.

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

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

- CL_INVALID_CONTEXT if _context_is not a valid context.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

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

The function

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

sets the execution status of a user event object.

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

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

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

- CL_INVALID_EVENT if *event* is not a valid user event object.
- CL_INVALID_VALUE if the *execution_status* is not CL_COMPLETE or a negative integer value.
- CL_INVALID_OPERATION if the *execution_status* for *event* has already been changed by a previous call to **clSetUserEventStatus**.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Note

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

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

```
ev1 = clCreateUserEvent(ctx, NULL);

clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ..., 1, &ev1, NULL);

clEnqueueWriteBuffer(cq, buf2, CL_FALSE,...);

clReleaseMemObject(buf2);

clSetUserEventStatus(ev1, CL_COMPLETE);
```

The following code sequence, however, works correctly.

```
ev1 = clCreateUserEvent(ctx, NULL);

clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ...,1, &ev1, NULL);

clEnqueueWriteBuffer(cq, buf2, CL_FALSE,...);

clSetUserEventStatus(ev1, CL_COMPLETE);

clReleaseMemObject(buf2);
```

The function

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

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

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

- CL_INVALID_VALUE if *num_events* is zero or *event_list* is NULL.
- CL_INVALID_CONTEXT if events specified in *event_list* do not belong to the same context.
- CL_INVALID_EVENT if event objects specified in *event_list* are not valid event objects.
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST if the execution status of any of the events in *event_list* is a negative integer value.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

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

returns information about the event object.

event specifies the event object being queried.

param_name specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetEventInfo** is described in *table 5.24*.

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 5.24*.

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

Table 5.25: *clGetEventInfo parameter queries.*

cl_event_info	**Return Type**	**Info. returned in *param_value***
CL_EVENT_COMMAND_ QUEUE	cl_command_ queue	Return the command-queue associated with *event*. For user event objects, a NULL value is returned.
CL_EVENT_CONTEXT	cl_context	Return the context associated with *event*.

Table 5.25: (continued)

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_NATIVE_KERNEL CL_COMMAND_READ_BUFFER CL_COMMAND_WRITE_BUFFER CL_COMMAND_COPY_BUFFER CL_COMMAND_READ_IMAGE CL_COMMAND_WRITE_IMAGE CL_COMMAND_COPY_IMAGE CL_COMMAND_COPY_BUFFER_ TO_IMAGE CL_COMMAND_COPY_IMAGE_ TO_BUFFER CL_COMMAND_MAP_BUFFER CL_COMMAND_MAP_IMAGE CL_COMMAND_UNMAP_MEM_ OBJECT CL_COMMAND_MARKER CL_COMMAND_ACQUIRE_ GL_OBJECTS CL_COMMAND_RELEASE_ GL_OBJECTS CL_COMMAND_READ_ BUFFER_RECT CL_COMMAND_WRITE_ BUFFER_RECT CL_COMMAND_COPY_ BUFFER_RECT CL_COMMAND_USER CL_COMMAND_BARRIER CL_COMMAND_MIGRATE_ MEM_OBJECTS CL_COMMAND_FILL_BUFFER CL_COMMAND_FILL_IMAGE CL_COMMAND_SVM_FREE CL_COMMAND_SVM_MEMCPY CL_COMMAND_SVM_MEMFILL CL_COMMAND_SVM_MAP CL_COMMAND_SVM_UNMAP

Table 5.25: (continued)

CL_EVENT_COMMAND_ EXECUTION_STATUS[16]:	cl_int	Return the execution status of the command identified by event. Valid values are: CL_QUEUED (command has been enqueued n the command-queue), CL_SUBMITTED (enqueued command has been submitted by the host to the device associated with the command-queue), CL_RUNNING (device is currently executing this command), CL_COMPLETE (the command has completed), or Error code given by a negative integer value. (command was abnormally terminated – this may be caused by a bad memory access etc.). These error codes come from the same set of error codes that are returned from the platform or runtime API calls as return values or errcode_ret values.
CL_EVENT_REFERENCE_ COUNT[17]:	cl_uint	Return the *event* reference count.

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. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.23* and *param_value* is not NULL.
- CL_INVALID_VALUE if information to query given in *param_name* cannot be queried for *event*.
- CL_INVALID_EVENT if *event* is a not a valid event object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

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

[17] 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.

The function

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

registers a user callback function for a specific command execution status. The registered callback function will be called when the execution status of command associated with *event* changes to an execution status equal to or past the status specified by *command_exec_status*.

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

event is a valid event object.

command_exec_callback_type specifies the command execution status for which the callback is registered. The command execution callback values for which a callback can be registered are: CL_SUBMITTED, CL_RUNNING or CL_COMPLETE[18]:. There is no guarantee that the callback functions registered for various execution status values for an event will be called in the exact order that the execution status of a command changes. Furthermore, it should be noted that receiving a call back for an event with a status other than CL_COMPLETE, in no way implies that the memory model or execution model as defined by the OpenCL specification has changed. For example, it is not valid to assume that a corresponding memory transfer has completed unless the event is in a state CL_COMPLETE.

pfn_event_notify is the event callback function that can be registered by the application. This callback function may be called asynchronously by the OpenCL implementation. It is the applications responsibility to ensure that the callback function is thread-safe. The parameters to this callback function are:

- *event* is the event object for which the callback function is invoked.
- *event_command_ _exec_status* is equal to the *command_exec_callback_type* used while registering the callback. Refer to table 5.23 for the command execution status values. If the callback is called as the result of the command associated with event being abnormally terminated, an appropriate error code for the error that caused the termination will be passed to *event_command_exec_status* instead.
- *user_data* is a pointer to user supplied data.

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

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

- **clFinish**,
- **clWaitForEvents**,
- blocking calls to **clEnqueueReadBuffer**, **clEnqueueReadBufferRect**,
- **clEnqueueWriteBuffer**, **clEnqueueWriteBufferRect**,
- blocking calls to **clEnqueueReadImage** and **clEnqueueWriteImage**,
- blocking calls to **clEnqueueMapBuffer** and **clEnqueueMapImage**,
- blocking calls to **clBuildProgram**, **clCompileProgram** or **clLinkProgram**,
- blocking calls to **clEnqueueSVMMemcpy** or **clEnqueueSVMMap**

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

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

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

- CL_INVALID_EVENT if *event* is not a valid event object.
- CL_INVALID_VALUE if *pfn_event_notify* is NULL or if *command_exec_callback_type* is not CL_SUBMITTED, CL_RUNNING or CL_COMPLETE.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clRetainEvent(cl_event event)
```

increments the *event* reference count. The OpenCL commands that return an event perform an implicit retain.

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

- CL_INVALID_EVENT if *event* is not a valid event object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

To release an event, use the following function

```
cl_int clReleaseEvent(cl_event event)
```

decrements the *event* reference count.

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

- CL_INVALID_EVENT if *event* is not a valid event object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The event object is deleted once the reference count becomes zero, the specific command identified by this event has completed (or terminated) and there are no commands in the command-queues of a context that require a wait for this event to complete. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainEvent** causes undefined behavior.

Note
Developers should be careful when releasing their last reference count on events created by **clCreateUserEvent** that have not yet been set to status of CL_COMPLETE or an error. If the user event was used in the event_wait_list argument passed to a **clEnqueue** API or another application host thread is waiting for it in **clWaitForEvents**, those commands and host threads will continue to wait for the event status to reach CL_COMPLETE or error, even after the application has released the object. Since in this scenario the application has released its last reference count to the user event, it would be in principle no longer valid for the application to change the status of the event to unblock all the other machinery. As a result the waiting tasks will wait forever, and associated events, cl_mem objects, command queues and contexts are likely to leak. In-order command queues caught up in this deadlock may cease to do any work.

5.12 Markers, Barriers and Waiting for Events

The function

```
cl_int clEnqueueMarkerWithWaitList(cl_command_queue command_queue,
                                   cl_uint num_events_in_wait_list,
                                   const cl_event *event_wait_list,
                                   cl_event *event)
```

enqueues a marker command which waits for either a list of events to complete, or if the list is empty it waits for all commands previously enqueued in *command_queue* to complete before it completes. This command returns an *event* which can be waited on, i.e. this event can be waited on to insure that all events either in the *event_wait_list* or all previously enqueued commands, queued before this command to *command_queue*, have completed.

command_queue is a valid host command-queue.

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, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

If *event_wait_list* is NULL, then this particular command waits until all previous enqueued commands to *command_queue* have completed.

event returns an event object that identifies this particular command. Event objects are unique and can be used to identify this marker command later on. *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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The function

```
cl_int clEnqueueBarrierWithWaitList(cl_command_queue command_queue,
                                    cl_uint num_events_in_wait_list,
                                    const cl_event *event_wait_list,
                                    cl_event *event)
```

enqueues a barrier command which waits for either a list of events to complete, or if the list is empty it waits for all commands previously enqueued in *command_queue* to complete before it completes. This command blocks command execution, that is, any following commands enqueued after it do not execute until it completes. This command returns an *event* which can be waited on, i.e. this event can be waited on to insure that all events either in the *event_wait_list* or all previously enqueued commands, queued before this command to *command_queue*, have completed

command_queue is a valid host command-queue.

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, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

If *event_wait_list* is NULL, then this particular command waits until all previous enqueued commands to *command_queue* have completed.

event returns an event object that identifies this particular command. Event objects are unique and can be used to identify this barrier command later on. *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. If the *event_wait_list* and the *event* arguments are not NULL, the *event* argument should not refer to an element of the *event_wait_list* array.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

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

The OpenCL functions that are submitted to a command-queue are enqueued in the order the calls are made but can be configured to execute in-order or out-of-order. The _properties_argument in **clCreateCommandQueueWithProperties** can be used to specify the execution order.

If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is not set, the commands enqueued to a command-queue execute in order. For example, if an application calls **clEnqueueNDRangeKernel** to execute kernel A followed by a **clEnqueueNDRangeKernel** to execute kernel B, the application can assume that kernel A finishes first and then kernel B is executed. If the memory objects output by kernel A are inputs to kernel B then kernel B will see the correct data in memory objects produced by execution of kernel A. If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is set, then there is no guarantee that kernel A will finish before kernel B starts execution.

Applications can configure the commands enqueued to a command-queue to execute out-of-order by setting the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of the command-queue. This can be specified when the command-queue is created. In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued. As there is no guarantee that kernels will be executed in order, i.e. based on when the **clEnqueueNDRangeKernel** calls are made within a command-queue, it is therefore possible that an earlier **clEnqueueNDRangeKernel** call to execute kernel A identified by event A may execute and/or finish later than a **clEnqueueNDRangeKernel** call to execute kernel B which was called by the application at a later point in time. To guarantee a specific order of execution of kernels, a wait on a particular event (in this case event A) can be used. The wait for event A can be specified in the *event_wait_list* argument to **clEnqueueNDRangeKernel** for kernel B.

In addition, a marker (**clEnqueueMarkerWithWaitList**) or a barrier (**clEnqueueBarrierWithWaitList**) command can be enqueued to the command-queue. The marker command ensures that previously enqueued commands identified by the list of events to wait for (or all previous commands) have finished. A barrier command is similar to a marker command, but additionally guarantees that no later-enqueued commands will execute until the waited-for commands have executed.

Similarly, commands to read, write, copy or map memory objects that are enqueued after **clEnqueueNDRangeKernel** 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** or **clEnqueueNativeKernel** can be used to enqueue a wait for event or a barrier command can be enqueued that must complete before reads or writes to the memory object(s) occur.

5.14 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[19]: being referred to are: **clEnqueue{Read|Write|Map}Buffer, clEnqueue{Read|Write}BufferRect, clEnqueue{Read|Write|Map}Image, clEnqueueUnmapMemObject**,**clEnqueueSVMMemcpy**, **clEnqueueSVMMemFill**, **clEnqueueSVMMap**, **clEnqueueSVMUnmap**, **clEnqueueSVMFree**,**clEnqueueCopyBuffer, clEnqueueCopyBufferRect, clEnqueueCopyImage, clEnqueueCopyImageToBuffer, clEnqueueCopyBufferToImage, clEnqueueNDRangeKernel** and **clEnqueueNativeKernel**. These enqueued commands are identified by unique event objects.

Event objects can be used to capture profiling information that measure execution time of a command. Profiling of OpenCL commands can be enabled either by using a command-queue created with CL_QUEUE_PROFILING_ENABLE flag set in _properties_argument to **clCreateCommandQueueWithProperties**.

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 5.25*

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 5.25*.

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

[19] clEnqueueAcquireGLObjects and clEnqueueReleaseGLObjects defined in section 9.6.6 of the OpenCL 2.0 Extension Specification are also included.

Table 5.26: *clGetEventProfilingInfo parameter queries.*

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.
CL_PROFILING_COMMAND_COMPLETE	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event and any child commands enqueued by this command on the device have finished execution.

The unsigned 64-bit values returned can be used to measure the time in nano-seconds consumed by OpenCL commands.

OpenCL devices are required to correctly track time across changes in device frequency and power states. The CL_DEVICE_PROFILING_TIMER_RESOLUTION specifies the resolution of the timer i.e. the number of nanoseconds elapsed before the timer is incremented.

clGetEventProfilingInfo returns CL_SUCCESS if the function is executed successfully and the profiling information has been recorded. Otherwise, it returns one of the following errors:

- CL_PROFILING_INFO_NOT_AVAILABLE if the CL_QUEUE_PROFILING_ENABLE flag is not set for the command-queue, if the execution status of the command identified by *event* is not CL_COMPLETE or if *event* refers to the **clEnqueueSVMFree** command or is a user event object.
- CL_INVALID_VALUE if *param_name* is not valid, or if size in bytes specified by *param_value_size* is < size of return type as described in *table 5.25* and *param_value* is not NULL.
- CL_INVALID_EVENT if *event* is a not a valid event object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.15 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* will eventually be submitted to the appropriate device. There is no guarantee that they will be complete after **clFlush** returns.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid host command-queue.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Any blocking commands queued in a command-queue and **clReleaseCommandQueue** perform an implicit flush of the command-queue. These blocking commands are **clEnqueueReadBuffer**, **clEnqueueReadBufferRect**, **clEnqueueReadImage**, with *blocking_read* set to CL_TRUE; **clEnqueueWriteBuffer**,**clEnqueueWriteBufferRect**,**clEnqueueWriteImage** with *blocking_write* set to CL_TRUE; **clEnqueueMapBuffer**, **clEnqueueMapImage** with *blocking_map* set to CL_TRUE; **clEnqueueSVMMemcpy** with *blocking_copy* set to CL_TRUE; **clEnqueueSVMMap** with *blocking_map* set to CL_TRUE or **clWaitForEvents**.

To use event objects that refer to commands enqueued in a command-queue as event objects to wait on by commands enqueued in a different command-queue, the application must call a **clFlush** or any blocking commands that perform an implicit flush of the command-queue where the commands that refer to these event objects are enqueued.

The function

```
cl_int clFinish(cl_command_queue command_queue)
```

blocks until all previously queued OpenCL commands in *command_queue* are issued to the associated device and have completed. **clFinish** does not return until all previously queued commands in *command_queue* have been processed and completed. **clFinish** is also a synchronization point.

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

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid host command-queue.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

Chapter 6

Associated OpenCL specification

6.1 SPIR-V Intermediate language

The OpenCL 2.2 specification requires support for the SPIR-V intermediate language that allows offline, or linked online, compilation to a binary format that may be consumed by the **clCreateProgramWithIL** interface.

The OpenCL specification includes a specification for the SPIR-V 1.2 intermediate language as a cross-platform input language. In addition, platform vendors may support their own IL if this is appropriate. The OpenCL runtime will return a list of supported IL versions using the **CL_DEVICE_IL_VERSION** parameter to the *clGetDeviceInfo*query.

6.2 Extensions to OpenCL

In addition to the specification of core features, OpenCL provides a number of extensions to the API, kernel language or intermediate representation. These features are defined in the OpenCL 2.2 extensions specification document.

Extensions defined against earlier versions of the OpenCL specifications, whether the API or language specification, are defined in the matching versions of the extension specification document.

6.3 Support for earlier OpenCL C kernel languages

The OpenCL C kernel language is not defined in the OpenCL 2.2 specification. New language features are described in the OpenCL C++ specification as well as the SPIR-V 1.2 specification and in kernel languages that target it. A kernel language defined by any of the OpenCL 1.0, OpenCL 1.1, OpenCL 1.2 and OpenCL 2.0 kernel language specifications as well as kernels language extensions defined by the matching versions of OpenCL extension specifications are valid to pass to **clCreateProgramWithSource** executing against an OpenCL 2.2 runtime.

Chapter 7

OpenCL Embedded Profile

The OpenCL 2.2specification describes the feature requirements for desktop platforms. This section describes the OpenCL 2.2embedded profile that allows us to target a subset of the OpenCL 2.2specification for handheld and embedded platforms. The optional extensions defined in the OpenCL 2.2Extension Specification apply to both profiles.

The OpenCL 2.2 embedded profile has the following restrictions:

1. 64 bit integers i.e. long, ulong including the appropriate vector data types and operations on 64-bit integers are optional. The *cles_khr_int64[1]: extension string will be reported if the embedded profile implementation supports 64-bit integers.
2. [line-through] Support for 3D images is optional.
3. If [line-through] **CL_DEVICE_IMAGE3D_MAX_WIDTH**, [line-through] **CL_DEVICE_IMAGE3D_MAX_HEIGHT** and [line-through] **CL_DEVICE_IMAGE3D_MAX_DEPTH** are zero, the call to **clCreateImage** in the embedded profile will fail to create the 3D image. The *errcode_ret* argument in **clCreateImage** returns [line-through] CL_INVALID_OPERATION. Declaring arguments of type*[line-through]*image3d_t in a kernel will result in a compilation error.

If [line-through] **CL_DEVICE_IMAGE3D_MAX_WIDTH**, [line-through] **CL_DEVICE_IMAGE3D_HEIGHT** and [line-through] CL_DEVICE_IMAGE3D_MAX_DEPTH > 0, 3D images are supported by the OpenCL embedded profile implementation. [line-through] **clCreateImage** will work as defined by the OpenCL specification. The [line-through] image3d_t data type can be used in a kernel(s).

1. [line-through] Support for 2D image array writes is optional. If the **cles_khr_2d_image_array_writes** extension is supported by the [line-through] embedded profile, writes to 2D image arrays are supported.
2. [line-through] Image and image arrays created with an [line-through] image_channel_data_type value of [line-through] CL_FLOAT or [line-through] CL_HALF_FLOAT can only be used with samplers that use a filter mode of [line-through] CL_FILTER_NEAREST. The values returned by **read_imagef** and *read_imageh*[2]: for 2D and 3D images if [line-through] image_channel_data_type value is [line-through] CL_FLOAT or [line-through] CL_HALF_FLOAT and sampler with [line-through] filter_mode =CL_FILTER_LINEAR are undefined.
3. The mandated minimum single precision floating-point capability given by CL_DEVICE_SINGLE_FP_CONFIG is CL_FP_ROUND_TO_ZERO or CL_FP_ROUND_TO_NEAREST. If CL_FP_ROUND_TO_NEAREST is supported, the default rounding mode will be round to nearest even; otherwise the default rounding mode will be round to zero.
4. The single precision floating-point operations (addition, subtraction and multiplication) shall be correctly rounded. Zero results may always be positive 0.0. The accuracy of division and sqrt are given in the SPIR-V OpenCL environment specification.

[1] Note that the performance of 64-bit integer arithmetic can vary significantly between embedded devices.
[2] If cl_khr_fp16 extension is supported.

If CL_FP_INF_NAN is not set in CL_DEVICE_SINGLE_FP_CONFIG, and one of the operands or the result of addition, subtraction, multiplication or division would signal the overflow or invalid exception (see IEEE 754 specification), the value of the result is implementation-defined. Likewise, single precision comparison operators (<, >, ⇐, >=, ==, !=) return implementation-defined values when one or more operands is a NaN.

In all cases, conversions (see the SPIR-V OpenCL environment specification) shall be correctly rounded as described for the FULL_PROFILE, including those that consume or produce an INF or NaN. The built-in math functions shall behave as described for the FULL_PROFILE, including edge case behavior but with slightly different accuracy rules. Edge case behavior and accuracy rules are described in the SPIR-V OpenCL environment specification.

NOTE

If addition, subtraction and multiplication have default round to zero rounding mode, then **fract**, **fma** and **fdim** shall produce the correctly rounded result for round to zero rounding mode.

This relaxation of the requirement to adhere to IEEE 754 requirements for basic floating-point operations, though extremely undesirable, is to provide flexibility for embedded devices that have lot stricter requirements on hardware area budgets.

1. 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. The SPIR-V environment specification for details.
2. The precision of conversions from CL_UNORM_INT8, CL_SNORM_INT8, CL_UNORM_INT16, CL_SNORM_INT16, CL_UNORM_INT_101010 and CL_UNORM_INT_101010_2 to float is ⇐ 2 ulp for the embedded profile instead of ⇐ 1.5 ulp as defined in the full profile. The exception cases described in the full profile and given below apply to the embedded profile.

For CL_UNORM_INT8

- 0 must convert to 0.0f and
- 255 must convert to 1.0f

For CL_UNORM_INT16

- 0 must convert to 0.0f and
- 65535 must convert to 1.0f

For CL_SNORM_INT8

- -128 and -127 must convert to -1.0f,
- 0 must convert to 0.0f and
- 127 must convert to 1.0f

For CL_SNORM_INT16

- -32768 and -32767 must convert to -1.0f,
- 0 must convert to 0.0f and
- 32767 must convert to 1.0f

For CL_UNORM_INT_101010

- 0 must convert to 0.0f and

- 1023 must convert to 1.0f

For CL_UNORM_INT_101010_2

- 0 must convert to 0.0f and
- 1023 must convert to 1.0f (for RGB)
- 3 must convert to 1.0f (for A)

The following optional extensions defined in the OpenCL 2.2 Extension Specification are available to the embedded profile:

cl_khr_int64_base_atomics

cl_khr_int64_extended_atomics

cl_khr_fp16

cles_khr_int64.

If double precision is supported i.e. CL_DEVICE_DOUBLE_FP_CONFIG is not zero, then cles_khr_int64 must also be supported.

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* that have been modified for the OpenCL embedded profile are listed below:

cl_device_info	Return Type	Description
CL_DEVICE_MAX_READ_IMAGE_ARGS	cl_uint	Max number of image objects arguments of a kernel declared with the read_only qualifier. The minimum value is 8 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	cl_uint	Max number of image objects arguments of a kernel declared with the write_only qualifier. The minimum value is 8 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS	cl_uint	Max number of image objects arguments of a kernel declared with the write_only or read_write qualifier. 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 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 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.
CL_DEVICE_IMAGE3D_ MAX_DEPTH	size_t	Max depth of 3D image in pixels. The minimum value is 2048.
CL_DEVICE_IMAGE_ MAX_BUFFER_SIZE	size_t	Max number of pixels for a 1D image created from a buffer object. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_IMAGE_ MAX_ARRAY_SIZE	size_t	Max number of images in a 1D or 2D image array. The minimum value is 256 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. 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 all arguments that can be passed to a kernel. The minimum value is 256 bytes for devices that are not of type CL_DEVICE_TYPE_CUSTOM.

CL_DEVICE_SINGLE_ FP_CONFIG	cl_device_ fp_config	Describes single precision floatingpoint 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 positive and negative infinity rounding modes supported CL_FP_FMA – IEEE754-2008 fused multiply-add is supported. CL_FP_CORRECTLY_ ROUNDED_DIVIDE _SQRT – divide and sqrt are correctly rounded as defined by the IEEE754 specification. CL_FP_SOFT_FLOAT – Basic floatingpoint operations (such as addition, subtraction, multiplication) are implemented in software. The mandated minimum floating-point capability is: CL_FP_ROUND_TO_ZERO or CL_FP_ROUND_TO_NEAREST for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_MAX_CONSTANT_ BUFFER_SIZE	cl_ulong	Max size in bytes of a constant buffer allocation. The minimum value is 1 KB for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_MAX_CONSTANT_ ARGS	cl_uint	Max number of arguments declared with the __constant qualifier in a kernel. The minimum value is 4 for devices that are not of type CL_DEVICE_TYPE_CUSTOM.

CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong	Size of local memory arena in bytes. The minimum value is 1 KB for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_COMPILER_AVAILABLE	cl_bool	Is CL_FALSE if the implementation does not have a compiler available to compile the program source. Is CL_TRUE if the compiler is available. This can be CL_FALSE for the embedded platform profile only.
CL_DEVICE_LINKER_AVAILABLE	cl_bool	Is CL_FALSE if the implementation does not have a linker available. Is CL_TRUE if the linker is available. This can be CL_FALSE for the embedded platform profile only. This must be CL_TRUE if CL_DEVICE_COMPILER_AVAILABLE is CL_TRUE.
CL_DEVICE_QUEUE_ON_DEVICE_MAX_SIZE	cl_uint	The max. size of the device queue in bytes. The minimum value is 64 KB for the embedded profile
CL_DEVICE_PRINTF_BUFFER_SIZE	size_t	Maximum size in bytes of the internal buffer that holds the output of printf calls from a kernel. The minimum value for the EMBEDDED profile is 1 KB.

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.

Chapter 8

Appendix A

8.1 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.

8.2 A.2 Multiple Host Threads

All OpenCL API calls are thread-safe[1]: except those that modify the state of cl_kernel objects: **clSetKernelArg, clSetKernelArgSVMPointer, clSetKernelExecInfo*and *clCloneKernel**. **clSetKernelArg** ,

[1] Please refer to the OpenCL glossary for the OpenCL definition of thread -safe. This definition may be different from usage of the term in other contexts.

clSetKernelArgSVMPointer, **clSetKernelExecInfo** and **clCloneKernel*are safe to call from any host thread, and safe to call re-entrantly so long as concurrent calls to any combination of these API calls operate on different cl_kernel objects. The state of the cl_kernel object is undefined if *clSetKernelArg**, **clSetKernelArgSVMPointer**, **clSetKernelExecInfo** or *clCloneKernel*are called from multiple host threads on the same cl_kernel object at the same time[2]:. Please note that there are additional limitations as to which OpenCL APIs may be called from OpenCL callback functions — please see *section 5.11*.

The behavior of OpenCL APIs called from an interrupt or signal handler is implementation-defined

The OpenCL implementation should be able to create multiple command-queues for a given OpenCL context and multiple OpenCL contexts in an application running on the host processor.

[2] There is an inherent race condition in the design of OpenCL that occurs between setting a kernel argument and using the kernel with clEnqueueNDRangeKernel. Another host thread might change the kernel arguments between when a host thread sets the kernel arguments and then enqueues the kernel, causing the wrong kernel arguments to be enqueued. Rather than attempt to share cl_kernel objects among multiple host threads, applications are strongly encouraged to make additional cl_kernel objects for kernel functions for each host thread.

Chapter 9

Appendix B Portability

OpenCL is designed to be portable to other architectures and hardware designs. OpenCL has used at its core a C99 based programming language and follows rules based on that heritage. 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 whereas the same calls on the GPU will likely fail for now. 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 SPIR-V IL 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[1]: 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

In register (little-endian):

[1] 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.

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.

When the host and device have different endianness, the developer must ensure that kernel argument values are processed correctly. The implementation may or may not automatically convert endianness of kernel arguments. Developers should consult vendor documentation for guidance on how to handle kernel arguments in these situations.

OpenCL provides a consistent programming model across architectures by numbering elements according to their order in memory. Concepts such as even/odd and high/low follow accordingly. Once the data is loaded into registers, we find that element 0 is at the left of the big-endian vector and element 0 is at the right of the little-endian vector:

```
float x[4];

float4 v = vload4( 0, x );
```

Big-endian:

```
v contains {~x[0],~x[1],~x[2],~x[3] }
```

Little-endian:

```
v~contains~{~x[3],~x[2],~x[1],~x[0] }
```

The compiler is aware that this swap occurs and references elements accordingly. So long as we refer to them by a numeric index such as .s0123456789abcdef or by descriptors such as .xyzw, .hi, .lo, .even and .odd, everything works transparently. Any ordering reversal is undone when the data is stored back to memory. The developer should be able to work with a big endian programming model and ignore the element ordering problem in the vector . . . for most problems. This mechanism relies on the fact that we can rely on a consistent element numbering. Once we change numbering system, for example by conversion-free casting (using as_type_n_) a vector to another vector of the same size but a different number of elements, then we get different results on different implementations depending on whether the system is big- endian, or little-endian or indeed has no vector unit at all. (Thus, the behavior of bitcasts to vectors of different numbers of elements is implementation-defined, see section 1.2.4 of OpenCL 2.0C specification)

An example follows:

```
float x[4] = { 0.0f, 1.0f, 2.0f, 3.0f };

float4 v = vload4( 0, x );

uint4 y = as_uint4(v);  legal, portable~

ushort8 z = as_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)
- Any bitcast that changes the number of elements in the vector. (Operations on the new type are non-portable.)

* Swizzle operations that change the order of data using chunk sizes that are not the same as the element size

Examples of operations that are portable:

- Combining two uint8's to make a uchar16 using .even and .odd operators. For example to interleave left and right audio streams.
- Any bitcast that does not change the number of elements (e.g. (float4) unit4 — we define the storage format for floating-point types)
- Swizzle operations that swizzle elements of the same size as the elements of the vector.

OpenCL has made some additions to C to make application behavior more dependable than C. Most notably in a few cases OpenCL defines the behavior of some operations that are undefined in C99:

- OpenCL provides convert_ operators for conversion between all types. C99 does not define what happens when a floating-point type is converted to integer type and the floating-point value lies outside the representable range of the integer type after rounding. When the *sat variant of the conversion is used, the float shall be converted to the nearest representable integer value. Similarly, OpenCL also makes recommendations about what should happen with NaN. Hardware manufacturers that provide the saturated conversion in hardware may use the saturated conversion hardware for both the saturated and non-saturated versions of the OpenCL convert* operator. OpenCL does not define what happens for the non-saturated conversions when floating-point operands are outside the range representable integers after rounding.
- The format of half, float, and double types is defined to be the binary16, binary32 and binary64 formats in the draft IEEE-754 standard. (The latter two are identical to the existing IEEE-754 standard.) You may depend on the positioning and meaning of the bits in these types.
- OpenCL defines behavior for oversized shift values. Shift operations that shift greater than or equal to the number of bits in the first operand reduce the shift value modulo the number of bits in the element. For example, if we shift an int4 left by 33 bits, OpenCL treats this as shift left by 33%32 = 1 bit.
- A number of edge cases for math library functions are more rigorously defined than in C99. Please see _section 3.5_of the OpenCL 2.0 C specification.

Chapter 10

Appendix C Application Data Types

This section documents the provided host application types and constant definitions. The documented material describes the commonly defined data structures, types and constant values available to all platforms and architectures. The addition of these details demonstrates our commitment to maintaining a portable programming environment and potentially deters changes to the supplied headers.

10.1 C.1 Shared Application Scalar Data Types

The following application scalar types are provided for application convenience.

cl_char

cl_uchar

cl_short

cl_ushort

cl_int

cl_uint

cl_long

cl_ulong

cl_half

cl_float

cl_double

10.2 C.2 Supported Application Vector Data Types

Application vector types are unions used to create vectors of the above application scalar types. The following application vector types are provided for application convenience.

cl_char_n_

cl_uchar_n_

cl_short_n_

cl_ushort_n_

cl_int_n_

cl_uint_n_

cl_long_n_

cl_ulong_n_

cl_half_n_

cl_float_n_

cl_double_n_

n can be 2, 3, 4, 8 or 16.

The application scalar and vector data types are defined in the **cl_platform.h** header file.

10.3 C.3 Alignment of Application Data Types

The user is responsible for ensuring that pointers passed into and out of OpenCL kernels are natively aligned relative to the data type of the parameter as defined in the kernel language and SPIR-V specifications. This implies that OpenCL buffers created with CL_MEM_USE_HOST_PTR need to provide an appropriately aligned host memory pointer that is aligned to the data types used to access these buffers in a kernel(s), that SVM allocations must correctly align and that pointers into SVM allocations must also be correctly aligned. The user is also responsible for ensuring image data passed is aligned to the granularity of the data representing a single pixel (e.g. image_num_channels * sizeof(image_channel_data_type)) except for CL_RGB and CL_RGBx images where the data must be aligned to the granularity of a single channel in a pixel (i.e. sizeof(image_channel_data_type)). This implies that OpenCL images created with CL_MEM_USE_HOST_PTR must align correctly. The image alignment value can be queried using the CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT query. In addition,

source pointers for clEnqueueWriteImage and other operations that copy to the OpenCL runtime, as well as destination pointers for clEnqueueReadImage and other operations that copy from the OpenCL runtime must follow the same alignment rules.

OpenCL makes no requirement about the alignment of OpenCL application defined data types outside of buffers and images, except that the underlying vector primitives (e.g. *_cl_float4) where defined shall be directly accessible as such using appropriate named fields in the cl_type union (see _section C.5*). Nevertheless, it is recommended that the **cl_platform.h** header should attempt to naturally align OpenCL defined application data types (e.g. cl_float4) according to their type.

10.4 C.4 Vector Literals

Application vector literals may be used in assignments of individual vector components. Literal usage follows the convention of the underlying application compiler.

```
cl_float2 foo = { .s[1] = 2.0f };

cl_int8 bar = {{ 2, 4, 6, 8, 10, 12, 14, 16 }};
```

10.5 C.5 Vector Components

The components of application vector types can be addressed using the <vector_name>.s[<index>] notation.

For example:

```
foo.s[0] = 1.0f; // Sets the 1st vector component of foo

pos.s[6] = 2; // Sets the 7th vector component of bar
```

In some cases vector components may also be accessed using the following notations. These notations are not guaranteed to be supported on all implementations, so their use should be accompanied by a check of the corresponding preprocessor symbol.

10.5.1 C.5.1 Named vector components notation

Vector data type components may be accessed using the .sN, .sn or .xyzw field naming convention, similar to how they are used within the OpenCL language. Use of the .xyzw field naming convention only allows accessing of the first 4 component fields. Support of these notations is identified by the CL_HAS_NAMED_VECTOR_FIELDS preprocessor symbol. For example:

```
#ifdef CL_HAS_NAMED_VECTOR_FIELDS

cl_float4 foo;

cl_int16 bar;

foo.x = 1.0f; // Set first component

foo.s0 = 1.0f; // Same as above

bar.z = 3;  // Set third component

bar.se = 11;  // Same as bar.s[0xe]

bar.sD = 12;  // Same as bar.s[0xd]

#endif
```

Unlike the OpenCL language type usage of named vector fields, only one component field may be accessed at a time. This restriction prevents the ability to swizzle or replicate components as is possible with the OpenCL language types. Attempting to access beyond the number of components for a type also results in a failure.

```
foo.xy  // illegal - illegal field name combination

bar.s1234 // illegal - illegal field name combination

foo.s7  // illegal - no component s7
```

10.5.2 C.5.2 High/Low vector component notation

Vector data type components may be accessed using the .hi and .lo notation similar to that supported within the language types. Support of this notation is identified by the CL_HAS_HI_LO_VECTOR_FIELDS preprocessor symbol. For example:

```
#ifdef CL_HAS_HI_LO_VECTOR_FIELDS

cl_float4 foo;

cl_float2 new_hi = 2.0f, new_lo = 4.0f;

foo.hi = new_hi;

foo.lo = new_lo;

#endif
```

10.5.3 C.5.3 Native vector type notation

Certain native vector types are defined for providing a mapping of vector types to architecturally builtin vector types. Unlike the above described application vector types, these native types are supported on a limited basis depending on the supporting architecture and compiler.

These types are not unions, but rather convenience mappings to the underlying architectures' builtin vector types. The native types share the name of their application counterparts but are preceded by a double underscore "__".

For example, *cl_float4 is the native builtin vector type equivalent of the cl_float4 application vector type. The* cl_float4 type may provide direct access to the architectural builtin __m128 or vector float type, whereas the cl_float4 is treated as a union.

In addition, the above described application data types may have native vector data type members for access convenience. The native components are accessed using the .vN sub-vector notation, where N is the number of elements in the sub-vector. In cases where the native type is a subset of a larger type (more components), the notation becomes an index based array of the sub-vector type.

Support of the native vector types is identified by a *CL_TYPEN* preprocessor symbol matching the native type name. For example:

```
#ifdef __CL_FLOAT4__ // Check for native cl_float4 type

cl_float8 foo;

__cl_float4 bar; // Use of native type

bar = foo.v4[1]; // Access the second native float4 vector

#endif
```

10.6 C.6 Implicit Conversions

Implicit conversions between application vector types are not supported.

10.7 C.7 Explicit Casts

Explicit casting of application vector types (cl_typen) is not supported. Explicit casting of native vector types (__cl_typen) is defined by the external compiler.

10.8 C.8 Other operators and functions

The behavior of standard operators and function on both application vector types (cl_typen) and native vector types (__cl_typen) is defined by the external compiler.

10.9 C.9 Application constant definitions

In addition to the above application type definitions, the following literal defintions are also available.

CL_CHAR_BIT	Bit width of a character
CL_SCHAR_MAX	Maximum value of a type cl_char
CL_SCHAR_MIN	Minimum value of a type cl_char
CL_CHAR_MAX	Maximum value of a type cl_char
CL_CHAR_MIN	Minimum value of a type cl_char
CL_UCHAR_MAX	Maximum value of a type cl_uchar
CL_SHORT_MAX	Maximum value of a type cl_short
CL_SHORT_MIN	Minimum value of a type cl_short
CL_USHORT_MAX	Maximum value of a type cl_ushort
CL_INT_MAX	Maximum value of a type cl_int
CL_INT_MIN	Minimum value of a type cl_int
CL_UINT_MAX	Maximum value of a type cl_uint
CL_LONG_MAX	Maximum value of a type cl_long
CL_LONG_MIN	Minimum value of a type cl_long
CL_ULONG_MAX	Maximum value of a type cl_ulong
CL_FLT_DIAG	Number of decimal digits of precision for the type cl_float
CL_FLT_MANT_DIG	Number of digits in the mantissa of type cl_float

CL_FLT_MAX_10_EXP	Maximum positive integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type cl_float
CL_FLT_MAX_EXP	Maximum exponent value of type cl_float
CL_FLT_MIN_10_EXP	Minimum negative integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type cl_float
CL_FLT_MIN_EXP	Minimum exponent value of type cl_float
CL_FLT_RADIX	Base value of type cl_float
CL_FLT_MAX	Maximum value of type cl_float
CL_FLT_MIN	Minimum value of type cl_float
CL_FLT_EPSILON	Minimum positive floating-point number of type cl_float such that 1.0 + CL_FLT_EPSILON != 1 is true.
CL_DBL_DIG	Number of decimal digits of precision for the type cl_double
CL_DBL_MANT_DIG	Number of digits in the mantissa of type cl_double
CL_DBL_MAX_10_EXP	Maximum positive integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type cl_double
CL_DBL_MAX_EXP	Maximum exponent value of type cl_double
CL_DBL_MIN_10_EXP	Minimum negative integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type cl_double
CL_DBL_MIN_EXP	Minimum exponent value of type cl_double
CL_DBL_RADIX	Base value of type cl_double
CL_DBL_MAX	Maximum value of type cl_double
CL_DBL_MIN	Minimum value of type cl_double
CL_DBL_EPSILON	Minimum positive floating-point number of type cl_double such that 1.0 + CL_DBL_EPSILON != 1 is true.
CL_NAN	Macro expanding to a value representing NaN
CL_HUGE_VALF	Largest representative value of type cl_float
CL_HUGE_VAL	Largest representative value of type cl_double
CL_MAXFLOAT	Maximum value of type cl_float
CL_INFINITY	Macro expanding to a value represnting infinity

These literal definitions are defined in the **cl_platform.h** header.

Chapter 11

Appendix D CL_MEM_COPY_OVERLAP

The following code describes how to determine if there is overlap between the source and destination rectangles specified to **clEnqueueCopyBufferRect** provided the source and destination buffers refer to the same buffer object.

```
unsigned int

check_copy_overlap(const size_t src_origin[],
                   const size_t dst_origin[],
                   const size_t region[],
                   const size_t row_pitch,
                   const size_t slice_pitch )
{

  const size_t slice_size = (region[1] - 1) * row_pitch + region[0];
  const size_t block_size = (region[2] - 1) * slice_pitch + slice_size;
  const size_t src_start = src_origin[2] * slice_pitch +
  src_origin[1] * row_pitch +
  src_origin[0];
  const size_t src_end = src_start + block_size;
  const size_t dst_start = dst_origin[2] * slice_pitch
                           + dst_origin[1] * row_pitch
                           +  dst_origin[0];

  F  const size_t dst_end = dst_start + block_size;

  /* No overlap if dst ends before src starts or if src ends

   * before dst starts.

   */

  if( (dst_end <= src_start) || (src_end <= dst_start) ){
    return 0;
  }

  /* No overlap if region[0] for dst or src fits in the gap

   * between region[0] and row_pitch.

   */

  {
```

```
    const size_t src_dx = src_origin[0] % row_pitch;
    const size_t dst_dx = dst_origin[0] % row_pitch;

    if( ((dst_dx >= src_dx + region[0]) &&

        (dst_dx + region[0] <= src_dx + row_pitch)) ||

        ((src_dx >= dst_dx + region[0]) &&

       (src_dx + region[0] <= dst_dx + row_pitch)) )

      {
        return 0;
      }

  }

~
  /* No overlap if region[1] for dst or src fits in the gap

   * between region[1] and slice_pitch.

   */

   {

      const size_t src_dy =

        (src_origin[1] * row_pitch + src_origin[0]) % slice_pitch;

      const size_t dst_dy =

        (dst_origin[1] * row_pitch + dst_origin[0]) % slice_pitch;


      if( ((dst_dy >= src_dy + slice_size) &&

          (dst_dy + slice_size <= src_dy + slice_pitch)) ||

          ((src_dy >= dst_dy + slice_size) &&

          (src_dy + slice_size <= dst_dy + slice_pitch)) ) {
        return 0;

      }

   }

~
 /* Otherwise src and dst overlap. */

 return 1;

}
```

Chapter 12

Appendix E Changes

12.1 E.1 Summary of changes from OpenCL 1.0

The following features are added to the OpenCL 1.1 platform layer and runtime (*sections 4 and 5*):

- Following queries to *table 4.3*

o CL_DEVICE_NATIVE_VECTOR_WIDTH_{CHAR | SHORT | INT | LONG | FLOAT | DOUBLE | HALF}

o CL_DEVICE_HOST_UNIFIED_MEMORY

o CL_DEVICE_OPENCL_C_VERSION

- CL_CONTEXT_NUM_DEVICES to the list of queries specified to **clGetContextInfo**.
- Optional image formats: CL_Rx, CL_RGx and CL_RGBx.

Support for sub-buffer objects ability to create a buffer object that refers to a specific region in another buffer object using *clCreateSubBuffer.

- **clEnqueueReadBufferRect**, **clEnqueueWriteBufferRect** and **clEnqueueCopyBufferRect** APIs to read from, write to and copy a rectangular region of a buffer object respectively.
- **clSetMemObjectDestructorCallback** API to allow a user to register a callback function that will be called when the memory object is deleted and its resources freed.
- Options that control the OpenCL C version used when building a program executable. These are described in *section 5.8.4.5*.
- CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE to the list of queries specified to **clGetKernelWorkGroupInfo**.
- Support for user events. User events allow applications to enqueue commands that wait on a user event to finish before the command is executed by the device. Following new APIs are added*- clCreateUserEvent*and **clSetUserEventStatus**.
- **clSetEventCallback** API to register a callback function for a specific command execution status.

The following modifications are made to the OpenCL 1.1 platform layer and runtime (*sections 4 and 5*):

- Following queries in *table 4.3*

o CL_DEVICE_MAX_PARAMETER_SIZE from 256 to 1024 bytes

o CL_DEVICE_LOCAL_MEM_SIZE from 16 KB to 32 KB.

- The *global_work_offset* argument in **clEnqueueNDRangeKernel** can be a non-NULL value.
- All API calls except **clSetKernelArg** are thread-safe.

The following features are added to the OpenCL C programming language (*section 6*) in OpenCL 1.1:

- 3-component vector data types.
- New built-in functions

o **get_global_offset** work-item function defined in section *6.12.1*.

o **minmag**, **maxmag** math functions defined in section *6.12.2*.

o **clamp** integer function defined in *section 6.12.3*.

o (vector, scalar) variant of integer functions **min** and **max** in *section 6.12.3*.

o **async_work_group_strided_copy** defined in section *6.12.10*.

o **vec_step**, **shuffle** and **shuffle2** defined in section *6.12.12*.

- **cl_khr_byte_addressable_store** extension is a core feature.

***cl_khr_global_int32_base_atomics**, **cl_khr_global_int32_extended_atomics**, **cl_khr_local_int32_base_atomics** and **cl_khr_local_int32_extended_atomics*extensions are core features. The built-in atomic function names are changed to use the *atomic_** prefix instead of **atom_**.

- Macros CL_VERSION_1_0 and CL_VERSION_1_1.

The following features in OpenCL 1.0 are deprecated (see glossary) in OpenCL 1.1:

- The **clSetCommandQueueProperty** API is no longer supported in OpenCL 1.1.
- The *ROUNDING_MODE* macro is no longer supported in OpenCL C 1.1.

The cl-strict-aliasing option that can be specified in *options* argument to *clBuildProgram is no longer supported in OpenCL 1.1.

The following new extensions are added to *section 9* in OpenCL 1.1:

- **cl_khr_gl_event** for creating a CL event object from a GL sync object.
- **cl_khr_d3d10_sharing** for sharing memory objects with Direct3D 10.

The following modifications are made to the OpenCL ES Profile described in *section 10* in OpenCL 1.1:

- 64-bit integer support is optional.

12.2 E.2 Summary of changes from OpenCL 1.1

The following features are added to the OpenCL 1.2 platform layer and runtime (*sections 4 and 5*):

- Custom devices and built-in kernels are supported.
- Device partitioning that allows a device to be partitioned based on a number of partitioning schemes supported by the device.

* Extend *cl_mem_flags* to describe how the host accesses the data in a cl_mem object.

***clEnqueueFillBuffer** and **clEnqueueFillImage** to support filling a buffer with a pattern or an image with a color.

- Add CL_MAP_WRITE_INVALIDATE_REGION to *cl_map_flags*. Appropriate clarification to the behavior of CL_MAP_WRITE has been added to the spec.
- New image types: 1D image, 1D image from a buffer object, 1D image array and 2D image arrays.
- **clCreateImage** to create an image object.

***clEnqueueMigrateMemObjects** API that allows a developer to have explicit control over the location of memory objects or to migrate a memory object from one device to another.

* Support separate compilation and linking of programs.

- Additional queries to get the number of kernels and kernel names in a program have been added to **clGetProgramInfo**.
- Additiional queries to get the compile and link status and options have been added to **clGetProgramBuildInfo**.
- **clGetKernelArgInfo** API that returns information about the arguments of a kernel.
- **clEnqueueMarkerWithWaitList** and **clEnqueueBarrierWithWaitList** APIs.

The following features are added to the OpenCL C programming language (*section 6*) in OpenCL 1.2:

- Double-precision is now an optional core feature instead of an extension.
- New built in image types: **image1d_t** , **image1d_array_t** and **image2d_array_t** .
- New built-in functions

o Functions to read from and write to a 1D image, 1D and 2D image arrays described in *sections 6.12.14.2*, *6.12.14.3* and *6.12.14.4*.

o Sampler-less image read functions described in *section 6.12.14.3*.

o **popcount** integer function described in *section 6.12.3*.

o **printf** function described in *section 6.12.13*.

- Storage class specifiers extern and static as described in *section 6.8*.
- Macros CL_VERSION_1_2 and *OPENCL_C_VERSION*.

The following APIs in OpenCL 1.1 are deprecated (see glossary) in OpenCL 1.2:

- **clEnqueueMarker**, **clEnqueueBarrier** and **clEnqueueWaitForEvents**
- **clCreateImage2D** and **clCreateImage3D**
- **clUnloadCompiler*and*clGetExtensionFunctionAddress**

***clCreateFromGLTexture2D** and **clCreateFromGLTexture3D**

The following queries are deprecated (see glossary) in OpenCL 1.2:

***CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE** in *table 4.3* queried using **clGetDeviceInfo**.

12.3 E.3 Summary of changes from OpenCL 1.2

The following features are added to the OpenCL 2.0 platform layer and runtime (*sections 4 and 5*):

- Shared virtual memory.
- Device queues used to enqueue kernels on the device.
- Pipes.
- Images support for 2D image from buffer, depth images and sRGB images.

The following modifications are made to the OpenCL 2.0 platform layer and runtime (sections 4 and 5): **All API calls except *clSetKernelArg**, **clSetKernelArgSVMPointer** and **clSetKernelExecInfo** are thread-safe.

The following features are added to the OpenCL C programming language (*section 6*) in OpenCL 2.0:

- Clang Blocks.
- Kernels enqueing kernels to a device queue.
- Program scope variables in global address space.
- Generic address space.
- C1x atomics.
- New built-in functions (sections 6.13.9, 6.13.11, 6.13.15 and 6.14).
- Support images with the read_write qualifier.
- 3D image writes are a core feature.
- The CL_VERSION_2_0 macro.

The following APIs are deprecated (see glossary) in OpenCL 2.0:

- **clCreateCommandQueue** , **clCreateSampler** and **clEnqueueTask**

The following queries are deprecated (see glossary) in OpenCL 2.0:

- **CL_DEVICE_HOST_UNIFIED_MEMORY** in *table 4.3* queried using **clGetDeviceInfo**.
- **CL_IMAGE_BUFFER** in *table 5.10* is deprecated.
- **CL_DEVICE_QUEUE_PROPERTIES*is replaced by *CL_DEVICE_QUEUE_ON_HOST_PROPERTIES**.
- The explicit memory fence functions defined in section 6.12.9 of the OpenCL 1.2 specification.
- The OpenCL 1.2 atomic built-in functions for 32-bit integer and floating-point data types defined in section 6.12.11 of the OpenCL 1.2 specification.

12.4 E.4 Summary of changes from OpenCL 2.0

The following features are added to the OpenCL 2.1 platform layer and runtime (*sections 4 and 5*):

- **clGetKernelSubGroupInfo** API call.
- **CL_KERNEL_MAX_NUM_SUB_GROUPS**, **CL_KERNEL_COMPILE_NUM_SUB_GROUPS** additions to table 5.21 of the API specification.
- **clCreateProgramWithIL** API call.
- **clGetHostTimer** and **clGetDeviceAndHostTimer** API calls.
- **clEnqueueSVMMigrateMem** API call.

- **clCloneKernel** API call.
- **clSetDefaultDeviceCommandQueue** API call.
- **CL_PLATFORM_HOST_TIMER_RESOLUTION** added to table 4.1 of the API specification.
- **CL_DEVICE_IL_VERSION**, **CL_DEVICE_MAX_NUM_SUB_GROUPS**, **CL_DEVICE_SUB_GROUP_INDEPENDENT_FORWARD_PROGRESS** added to table 4.3 of the API specification.
- *CL_PROGRAM_IL*to table 5.17 of the API specification.
- **CL_QUEUE_DEVICE_DEFAULT** added to table 5.2 of the API specification.
- Added table 5.22 to the API specification with the enums: **CL_KERNEL_MAX_SUB_GROUP_SIZE_FOR_NDRANGE** , **CL_KERNEL_SUB_GROUP_COUNT_FOR_NDRANGE** and **CL_KERNEL_LOCAL_SIZE_FOR_SUB_GROUP_COUNT**

The following modifications are made to the OpenCL 2.1 platform layer and runtime (sections 4 and 5):

- All API calls except **clSetKernelArg** , **clSetKernelArgSVMPointer** , **clSetKernelExecInfo** and *clCloneKernel*are thread-safe.

The OpenCL C kernel language is no longer chapter 6. The OpenCL C kernel language is not updated for OpenCL 2.1. The OpenCL 2.0 kernel language will still be consumed by OpenCL 2.1 runtimes.

The SPIR-V IL specification has been added.

12.5 E.5 Summary of changes from OpenCL 2.1

The following changes have been made to the OpenCL 2.2 execution model (section 3)

- Added the third prerequisite (executing non-trivial constructors for program scope global variables).

The following features are added to the OpenCL 2.2 platform layer and runtime (*sections 4 and 5*):

- **clSetProgramSpecializationConstant** API call
- **clSetProgramReleaseCallback** API call
- Queries for CL_PROGRAM_SCOPE_GLOBAL_CTORS_PRESENT, CL_PROGRAM_SCOPE_GLOBAL_DTORS_PRESENT

The following modifications are made to the OpenCL 2.2 platform layer and runtime (section 4 and 5):

- Modified description of CL_DEVICE_MAX_CLOCK_FREQUENCY query.
- Added a new error code CL_MAX_SIZE_RESTRICTION_EXCEEDED to **clSetKernelArg** API call

Added definition of Deprecation and Specialization constants to the glossary.