IBM®

OpenCL Development Kit for Linux on Power

Installation and User's Guide

Version 0.3

OpenCL

March 15, 2011

IBM Systems and Technology Group

2070 Route 52, Bldg. 330

Hopewell Junction, NY 12533-6351

The IBM home page can be found at http://www.ibm.com

March 15, 2011

Table of Contents

1 Overview

The OpenCL™ Development Kit for Linux® on Power and accompanying XL C for OpenCL compiler is a technology preview of IBM's implementation of the OpenCL standard. OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. The IBM implementation supports Power Architecture® processors with the Vector/SIMD Multimedia Extension (Power/VMX) or the Vector Scalar eXtension (VSX), and Cell Broadband Engine™ Architecture (CBEA) compliant processors. CBEA processors include both the Cell Broadband Engine (Cell/B.E.) processor and the PowerXCell™ 8i processor.

This OpenCL implementation is fully conformant to the OpenCL 1.1 specification as certified by the successful execution of the OpenCL 1.1 conformance test suite.

1.1 Tested Configurations

The OpenCL Development Kit for Linux on Power has been tested on the PowerXCell™ 8i, Power6™, and Power7™ processor systems.

Tested Processor Systems	Example Products	Linux Operating Systems
PowerXCell 8i	IBM BladeCenter QS22	Red Hat Enterprise Linux 5.5 and 5.6
POWER6	IBM BladeCenter JS22/JS23/JS43	Red Hat Enterprise Linux 5.5, 5.6 and 6.0
POWER7	IBM Power 750 server, IBM Power 755 server, and IBM BladeCenter PS700, PS701, and PS702 Express	Red Hat Enterprise Linux 6.0 SUSE® Linux Enterprise Server 11 SP 1

While the use of this software on other Linux operating systems will likely be fully operational, it is untested and is not guaranteed functional or compliant.

1.2 A Word About Licenses

The OpenCL Development Kit runtime and XL C for OpenCL compiler are licensed under the International License Agreement for Early Release of Programs, or ILAR. After installation, a copy of the license can be found in `/usr/share/doc/OpenCL-0.3-ibm/documentation/license`. The 'LA_' prefixed files are the language specific License Agreement files and the 'LI_' prefixed files are the language specific License Information files.

The OpenCL samples are licensed under the International License Agreement for Non-Warranted Programs, or ILAN. A copy of the license can be found in the samples zipfile.

1.3 Getting Support

The OpenCL Development Kit for Linux on Power is not formally supported. Problems, issues, suggestions, and comments can be posted to its technology forum on the IBM alphaWorks website (http://www.alphaworks.ibm.com/tech/opencl/forum).

Note: *The Development Kit is provided on an "as-is" basis. Wherever possible, workarounds to problems will be provided on the forum.*

Useful information and discussion on all IBM OpenCL projects can be found in the *OpenCL Lounge* developerWorks group. A complete list of public groups can be found at https://www.ibm.com/developerworks/mydeveloperworks/groups.

1.4 Related Documentation

1. *Khronos API Implementers Guide*, Second Edition, Edited by Mark Callow, HI Corporation, Copyright © 2009 The Khronos Group Inc., http://www.khronos.org/registry/implementers_guide.pdf.
2. *Khronos OpenCL API registry* which includes the OpenCL 1.1 Core API Specification, Headers, and Extension Documentation. http://www.khronos.org/registry/cl.
3. *OpenCL 1.1 Reference Pages*, Khronos Group, http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml
4. *OpenCL API 1.1 Quick Reference Card*, Revision 0610, Copyright © 2010 Khronos Group, http://www.khronos.org/files/opencl-1-1-quick-reference-card.pdf.
5. *Power Instruction Set Architecture Version 2.06, http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf*
6. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology, Programming Environments Manual*, Version 2.07c, https://www.ibm.com/chips/techlib/techlib.nsf/techdocs/C40E4C6133B31EE8872570B500791108
7. *Synergistic Processor Unit Instruction Set Architecture*, Version 1.2, https://www.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44

2 Installing the OpenCL Development Kit

2.1 Prerequisites

Prerequisite software required for execution on CBEA systems like the IBM BladeCenter QS22 consists of the following packages which can be found on the Barcelona Supercomputing Center site:

```
ppu-binutils-2.18.50-21.ppc.rpm
ppu-gcc-4.1.1-166.ppc.rpm
spu-binutils-2.18.50-21.ppc.rpm
spu-gcc-4.1.1-166.ppc.rpm
spu-newlib-1.16.0-17.ppc.rpm
```

Prerequisite software required for execution on both CBEA and Power (VMX/VSX) systems consists of the following package which can be installed using your Linux distribution's installation tool – either `yum` for Red Hat Enterprise Linux or `zypper` (SUSE Linux Enterprise Server):

```
numactl
```

2.2 Development Kit Install / Update Procedure

The OpenCL Development Kit consists of two components, the OpenCL runtime and the XL C for OpenCL compiler, archived in a single ISO image. The installation of both components consists of the following 3 simple steps.

Step 1 – Install Prerequisite Software

As root, yum install the prerequisite numactl packages. On Red Hat Enterprise Linux enter:

```
yum install numactl.ppc numactl.ppc64
```

On SUSE Linux enter:

```
zypper install numactl libnuma1 libnuma1-32bit
```

For SUSE Linux Enterprise 11 Service Pack 1 users, ensure that your kernel is at least version 2.6.32.19-0.2.1. If not, install the latest Novell kernel patches. The kernel version can be obtained by entering "`uname -r`".

Step 2 – Install CBEA-specific Prerequisite Software

This step may be skipped when updating a previous install of the OpenCL Development Kit version (0.1 or 0.2) or installing the development kit on a non-CBEA system.

Download the following RPMs from the Barcelona Supercomputing Center site: http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.1/CellSDK-Open-RHEL/cbea/[1]

```
ppu-binutils-2.18.50-21.ppc.rpm
ppu-gcc-4.1.1-166.ppc.rpm
spu-binutils-2.18.50-21.ppc.rpm
spu-gcc-4.1.1-166.ppc.rpm
spu-newlib-1.16.0-17.ppc.rpm
```

As root, install the downloaded RPMs:

```
rpm –Uvh *.rpm
```

Step 3 – Install Development Kit Software

Download the OpenCL Development Kit ISO image, `OpenCL-0.3.linux.ppc.iso`, from alphaWorks (http://www.alphaworks.ibm.com/tech/opencl/download).

As root, mount the ISO image, install the software, and un-mount the image:

```
mount –o loop OpenCL-0.3.linux.ppc.iso /mnt
/mnt/install.sh
umount /mnt
```

2.3 Development Kit Uninstall Procedure

The section documents the procedure for uninstalling the OpenCL Development Kit for each of the tested systems. The differences in the procedure are the number of packages to be removed.

2.3.1 Uninstalling on a CBEA System

As root uninstall the IBM XL C for OpenCL compiler packages:

```
rpm –e opencl-xlc-help opencl-xlc-man opencl-xlc-lib opencl-xlc-rte     \
opencl-xlc-rte-lnk opencl-xlc-cmp opencl-cell-xlc-cmp opencl-cell-xlc-lib
```

As root uninstall the OpenCL runtime and developer packages:

```
rpm –e OpenCL-32bit OpenCL-devel-32bit OpenCL-cell OpenCL-cell-devel
```

2.3.2 Uninstalling on a Power System

As root uninstall the IBM XL C for OpenCL compiler packages:

```
rpm –e opencl-xlc-help opencl-xlc-man opencl-xlc-lib opencl-xlc-rte     \
opencl-xlc-rte-lnk opencl-xlc-cmp
```

[1] The prerequisite software is also available on the NCSA mirror site http://mirror.ncsa.illinois.edu/sdkma/sdk3.1/CellSDK-Open-RHEL/cbea/

As root uninstall the OpenCL runtime and developer packages:

```
rpm –e OpenCL-32bit OpenCL-devel-32bit OpenCL-64bit OpenCL-devel-64bit
```

2.4 OpenCL Samples Install Procedure

OpenCL programming samples have been provided in a zip file archive so that they may be installed on a wide variety of platforms. To install the samples, download the OpenCL Samples zip file, `OpenCL-samples-0.3-0.zip`, from alphaWorks (http://www.alphaworks.ibm.com/tech/opencl/download).

On Linux, unzip the package into the directory of your choosing by cd'ing to the directory and invoking:

```
unzip <src-path>/OpenCL-samples-0.3-0.zip
```

2.5 OpenCL GDB Debugger Install Procedure

Download the debugger's RPM, `ocl-gdb-7.2-21.ppc64.rpm`[2], from either the Barcelona Super Computing site (http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/opencl) or the NCSA mirror site (http://mirror.ncsa.illinois.edu/opencl).

As root, install the downloaded RPM:

```
rpm –Uvh ocl-gdb-7.2-21.ppc64.rpm
```

[2] The OpenCL Samples and GDB debugger may be released on a different schedule than the OpenCL Development kit. As such, the revision of the latest packages may differ from what is documented in this installation procedure.

3 Implementation Details

3.1 OpenCL Devices

The OpenCL Development Kit for Linux on Power provides a full profile Power (VMX/VSX) CPU device as well as an embedded profile SPU accelerator device. Both device types support execution of device kernels with up to 3 dimensions. On CPU devices, a local work-group size of up to 1024 work-items is supported. On SPU accelerator devices, a local work-group size of up to 256 work-items is supported.

The following OpenCL features are supported on both device types:

- Kernel compilation, and
- Out-of-order command execution.

The following optional OpenCL features are unsupported on both device types:

- OpenCL image objects, and
- OpenCL sampler objects.

The global memory size and maximum memory allocation size of both device types is dependent on the overall size and availability of system memory.

On the QS22, both the CPU (`CL_DEVICE_TYPE_CPU`) and accelerator (`CL_DEVICE_TYPE_ACCELERATOR`) devices exist and may be used simultaneously. Special care should be taken during simultaneous use, as both memory and compute resources are shared. Global memory is shared between the devices, which means memory consumption on one device effects the memory availability on both devices. In addition, the QS22 Power Processing Elements (PPEs) are used by both devices during execution.

Both 32-bit and 64-bit applications are supported on Power6 and Power7 Systems. Only 32-bit applications are supported on CBEA systems (e.g., QS22).

3.1.1 Power (VMX/VSX) CPU Device

The Power7 processor, Power6 processor, and the QS22's PPEs are all Power (VMX/VSX) CPU devices. The Power CPU device is of the `CL_DEVICE_TYPE_CPU` OpenCL device type. The OpenCL full profile is supported on this device type.

The number of OpenCL compute units on a Power (VMX/VSX) CPU device depends on the computer system, its number of cores, and the current simultaneous multithreading (SMT) setting. For example, the sample systems have the following number of available compute units:

		Number of Compute Units		
System	**Cores**	**SMT=1 (off)**	**SMT=2**	**SMT=4**
Cell QS22	2	n/a	4	n/a
Power6 JS22/JS23	4	4	8	n/a

Power6 JS43	8	8	16	n/a
Power7 PS700	4	4	8	16
Power7 PS701	8	8	16	32
Power7 PS702	16	16	32	64
Power7 755 server	32	32	64	128[3]

The SMT mode of Power6 and Power7 system can be set using the `ppc64_cpu` utility. See http://www.ibm.com/developerworks/wikis/display/LinuxP/powerpc-utils for details on use and features of the `ppc64_cpu` utility. As a rule of thumb, memory bound applications will generally perform better at higher SMT modes (for example, SMT=4); whereas compute bound applications will often perform best when SMT is disabled (off).

The CPU device global and local memory both map to system memory and therefore application memory consumption affects the availability of both.

Some other notable attributes of the CPU device include:

- native kernel execution support,
- floating-point denorm support, and
- floating-point IEEE 754-2008 fused multiply-add support.

3.1.2 SPU Accelerator Device

The QS22's 16 Synergistic Processor Units (SPU's) are consolidated into a single OpenCL accelerator device. The SPU accelerator device is of the CL_DEVICE_TYPE_ACCELERATOR device type. The OpenCL embedded profile is supported on this device type.

The maximum number of compute units on an SPU accelerator device is 16. The SPU accelerator device has a maximum local memory size of 256KB.

Memory buffer objects used in conjunction with an SPU accelerator device should be aligned at 128 bytes for best performance.

Some other notable attributes of the SPU accelerator device include:

- default floating-point rounding mode of round toward zero (rtz),
- floating-point IEEE754-2008 fused multiply-add support,
- no infinity and NaN support (except as specified in section 10, list item 6 of the OpenCL specification), and
- the native built-in functions (see Table 6.8 of the OpenCL specification) support the SPU's single-precision, extended number range. Infinities and NaNs are treated as large numbers and computation overflow result in ±0x1.FFFFEp+127 instead of ∞. The results are undefined for functions in which the specification mandates a NaN. For additional details on native built-ins, see the Native Built-

[3] For large systems like the IBM Power 755, 770, 785, and 795 servers, the number of compute units made available to 32-bit applications is software limited to 64. It is recommended for systems with more than 64 compute units, applications be compiled for 64-bit addressing and the CPU device be fissioned along NUMA domains.

ins section.

Note: *OpenCL utilizes all available SPUs to compose the SPU accelerator device. Any external usage of SPUs, whether within the same application or not, will result in a `CL_DEVICE_NOT_AVAILABLE` failure at context creation. Direct use or allocation of SPU resources after OpenCL context creation may result in undefined behavior.*

3.2 Optional Extensions

The OpenCL Development Kit for Linux on Power currently supports the following extensions:

- **Byte Addressable Store** (`cl_khr_byte_addressable_store`).
  Enables pointer writes to types of less than 32-bits. The `cl_khr_byte_addressable_store` extension is a core feature of OpenCL 1.1.

- **Device Fission** (`cl_ext_device_fission`).
  Extends OpenCL with the ability to split a device into multiple sub-devices. The only partition style supported by this OpenCL implementation is `{CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT, CL_AFFINITY_DOMAIN_NUMA_EXT}`. See Device Fission in section 5.1 for complete extension details.

- **Double Precision Floating-Point** (`cl_khr_fp64`).
  Adds support for double precision floating-point types, operations, and built-ins. This extension is provided on Power7 CPU devices and QS22 SPU accelerator devices. See section 9.3 of the OpenCL 1.1 specification for details.

- **Embedded 64-bit Integer** (`cles_khr_int64`).
  Adds support for 64-bit integer types (`long`, `ulong`, `long`*n*, and `ulong`*n*) on the SPU accelerator embedded device.

- **IBM Debug** (`cl_ibm_debug`).
  Provides a debug facility for obtaining the compute unit's identifier. This extension is designed to be used only during debug and as such is only available when using the OpenCL debug library (section 4.3.1). See IBM Debug in section 5.3 for extension details.

- **Memory Object Migration** (`cl_ext_migrate_memobject`).
  Provides a mechanism for assigning the device where an OpenCL memory object resides. See Migrate Memory Object in section 5.2 for extension details.

- **32-bit Atomics** (`cl_khr_global_int32_base_atomics`, `cl_khr_global_int32_extended_atomics`, `cl_khr_local_int32_base_atomics`, `cl_khr_local_int32_extended_atomics`).
  Provides both local and global atomic operations on 32-bit signed, unsigned integer, and single-precision floating-point memory locations. These extensions are core features of OpenCL. For details, see section 6.11.11 of the OpenCL specification.

- **64-bit Atomics** (`cl_khr_int64_base_atomics`, `cl_khr_int64_extended_atomics`).
  Provides both local and global atomic operations on 64-bit signed, unsigned integer memory

locations. See section 9.4 of the OpenCL 1.1 specification. The 64-bit atomics are not supported for 32-bit applications on CPU devices.

Requests for additional OpenCL extensions are welcome and should be expressed via the OpenCL alphaWorks forum (http://www.alphaworks.ibm.com/tech/opencl/forum).

3.3 Native Built-ins

A subset of the math functions are provided with a `native_` prefix. These functions generally have better performance compared to the non-native functions; however, they may also have reduced accuracy or device defined characteristics. This section documents the attributes of these built-ins for both device implementations. Applications using these built-ins should review the implementation characteristics to ensure any non-standard attributes are acceptable.

All native built-ins on the SPU accelerator device treat all IEEE 754-2008 infinities and NaN as extended precision floating-point numbers. (See the SPU Instruction Set Architecture for complete details on the SPU's extended precision numbers).

Native Built-in	Implementation Characteristics
`native_cos`	The `native_cos` function utilizes a single constant Cody & Waite range reduction method. This method achieves "good" results for inputs in the range -2π to 2π and has an absolute accuracy of less than 1.3e-07 over this input range. Relative accuracy is sacrificed near inputs ±π/2 and ±3π/2, where the cosine is small.
`native_divide`	The `native_divide` function is equivalent to the divide operator "/" and is therefore fully OpenCL compliant.
`native_exp` `native_exp10`	The `native_exp` and `native_exp10` functions are equivalent[4] to the OpenCL conformant `exp` and `exp10` functions, respectively.
`native_exp2`	The CPU device's `native_exp2` function is mapped to the VMX exponential estimate instruction, `vexptefp`, for both scalar and vector inputs. The estimate has a relative error no greater than one part in 16. The SPU accelerator's `native_exp2` function is equivalent to the OpenCL conformant `exp2` function.
`native_log` `native_log10`	The `native_log` and `native_log10` functions are equivalent to the OpenCL conformant `log` and `log10` functions, respectively.
`native_log2`	The CPU device's `native_log2` function is mapped to the VMX $\log_2$ estimate instruction, `vlogefp`, for both scalar and vector inputs. The estimate has an absolute error no greater than 2^{-5} and a relative error no greater than one part in 8. The SPU accelerator's `native_log2` function is equivalent to the OpenCL conformant `log2` function.
`native_powr`	The `native_powr` function approximates the powr function as `exp2(y * log2(x))`. The error is significant for large values of y*log2(x).

[4] Except where the SPU accelerator treats infinities and NaNs as extended precision, floating-point numbers.

`native_recip`	For scalar inputs on the CPU device, the `native_recip` function is mapped to the single precision floating reciprocal estimate instruction, `fres`. This estimate has a relative error no greater than one part in 256. Two iterations of Newton-Raphson are required to achieve single precision accuracy. For vector inputs on the CPU device, the `native_recip` function is mapped to the VMX vector reciprocal estimate instruction, `vrefp`. This estimate has a relative error no greater than one part in 4096. One iteration of Newton-Raphson is required to achieve single precision accuracy. On the SPU accelerator device, the `native_recip` function is mapped to a reciprocal estimate instruction (`frest`) followed by a floating interpolate instruction (`fi`). This estimate has a relative error of approximately one part in 4096.
`native_rsqrt`	For scalar inputs on the CPU device, the `native_rsqrt` function is mapped to the reciprocal estimate instruction, `frsqrte`. This estimate has a relative error of no greater than one part in 32. Three iterations of Newton-Raphson are required to achieve single precision accuracy. For vector inputs on the CPU device, the `native_rsqrt` function is mapped to the vector reciprocal estimate instruction, `vrsqrtefp`. This estimate has a relative error no greater than one part in 4096. One iteration of Newton-Raphson is required to achieve single precision accuracy. On the SPU accelerator device, the `native_rsqrt` function is mapped to a reciprocal square root estimate instruction (`frsqest`) followed by a floating interpolate instruction (`fi`). This estimate has a relative error of approximately one part in 4096.
`native_sin`	The `native_sin` function utilizes a single constant Cody & Waite range reduction method. This method achieves “good” results for inputs in the range -2π to 2π and has an absolute accuracy of less than 1.85e-07 over this input range. Relative accuracy is sacrificed near inputs ±π, where the sine is small.
`native_sqrt`	For scalar inputs on the CPU device, the `native_sqrt` function is mapped to the Power `fsqrts` instruction and is fully IEEE 754 compliant. For vector inputs on the CPU device, the `native_sqrt` function is mapped to the x*`native_rsqrt`(x) plus one Newton-Raphson iteration, where x is the input. This produces a result that is conforming to OpenCL’s accuracy requirement of <= 3 ulp. However, special values +0.0, -0.0, and ∞ incorrectly produce a NaN instead of +0.0, -0.0, and +0.0 respectively. On the SPU accelerator device, the `native_sqrt` function is mapped to the x*`native_rsqrt`(x) plus one Newton-Raphson iteration, where x is the input.
`native_tan`	The `native_tan` function utilizes a single constant Cody & Waite range reduction method. This method achieves reasonably “good” results for inputs in the range -π/2 to π/2.

Since the native built-ins are implementation dependent, using compiler conditionals to control their use may be required. The IBM OpenCL 1.1 kernel compiler’s predefined architecture macros can be used for this purpose. All macros are predefined to a value of 1.

Processor Architecture	Predefine Macros
Cell PPU	`__PPU__`
Cell SPU	`__SPU__`
Power6	`_ARCH_PWR6`
Power7	`_ARCH_PWR6`, `_ARCH_PWR7`

3.4 Implementation Details with Programmer Implications

This section documents other OpenCL implementation details which can affect application programs and how OpenCL programs are written.

3.4.1 Synchronous Memory Destructor Callbacks

The OpenCL specification states callbacks "may be asynchronous". Most of callback functions in IBM's OpenCL implementation are asynchronous except for the memory object destructor callback, which is synchronous. Therefore, programmers should not call `clReleaseMemObject()` and assume it will return before the callback is called. If a common mutex is used by both the caller and the callback function, then a deadlock will occur.

4 OpenCL Programming

4.1 OpenCL Samples

Several OpenCL programming samples have been provided with the Development Kit. These samples exemplify some of the best practices discussed in section Best Practices, and demonstrate various techniques of using the OpenCL framework and its extensions to leverage multiple compute units. The samples utilize IBM's OpenCL Utility Library (CLU). CLU consists of an abstracted set of OpenCL complementary utility functions that simplify the task of developing simple OpenCL applications.

Details on the programming samples, as well as on CLU, can be found in the samples installation package. See the section OpenCL Samples Install Procedure for instructions on downloading and installing the OpenCL programming samples.

4.2 Compilation

This section provides an overview of how to build OpenCL applications and the compiler technology that is involved.

4.2.1 Compiling Host Applications

Host applications can be built on a Power System™ using `xlc` (see IBM XL C/C++ for Linux), or `gcc`. On a CBEA system the host application can be built using `ppuxlc` (see IBM XL C/C++ for Multicore Acceleration for Linux) or `ppu-gcc`. The application will need to include the OpenCL header file (`#include <CL/opencl.h>`), and be linked with the OpenCL library (`-lOpenCL`) and the C++ standard library (`-lstdc++`).

4.2.2 Compiling OpenCL Kernels

Building of the application's kernel code happens when the application calls the `clBuildProgram()` OpenCL API. The API library in turn invokes the IBM XL C for OpenCL compiler to build a kernel binary.

Note: *The `/tmp` directory must be accessible for writes as it is used by the kernel build process for temporary file storage. These files should be cleaned up by the OpenCL API library upon exit of the application; but if not, any `/tmp/opencl_kernel_*` files and `/tmp/opencl_build_*` directories can manually be deleted if the application has completed.*

External Kernel Compilation

The `opencl_build_program` stand alone utility is provided for performing external compilation of OpenCL kernels. There are two forms of this utility. `opencl_build_program` compiles kernels for 32-bit addressing and `opencl_build_program_64` compiles kernels for 64-bit addressing. This utility is installed into `/usr/bin` as part of the base OpenCL runtime packages. Its options can be displayed with the --help flag. Binaries built with the utility can be passed to the OpenCL `clCreateProgramWithBinary()` API for the compiled device type.

Source to the utility is available in the `OpenCL-devel` packages (32-bit and 64-bit) and installed in the `/usr/share/doc/OpenCL-0.3-ibm/utilities` directory.

4.3 Application Debugging

OpenCL applications can encounter the following types of errors:

- OpenCL host API library errors
- OpenCL kernel compilation errors
- OpenCL kernel runtime errors

IBM's implementation of OpenCL provides assistance in identifying the source of these problems so they may be quickly and efficiently resolved.

4.3.1 Debugging OpenCL Host API Library Calls Errors

The OpenCL library returns an error code when it is unable to successfully complete a host API call. The OpenCL specification provides a description for these return codes; however, there are cases in which more than one underlying error condition can result in the same failing return code. To provide additional guidance, a debug OpenCL library is available that validates object pointers and by default prints additional information on errors to `stderr`. To use the debug library simply add the '`/usr/lib/CL/debug`' or `` ` ``/usr/lib64/CL/debug' directory to the `LD_LIBRARY_PATH` environment variable and run the application. For example:

```
export LD_LIBRARY_PATH=/usr/lib/CL/debug:/usr/lib64/CL/debug:$LD_LIBRARY_PATH
```

Note: *To assist in catching errors early, the debug library's OpenCL API argument validation is stricter. This stricter validation may result in different errors than those seen using the non-debug library.*

The diagnostic error messages output by the debug library are in the following standardized format:

```
[Severity] [OpenCL API] [Error Type<:OpenCL Return Code>] <Error Message>
```

Sample errors include:

```
[ERROR] [clGetPlatformIDs] [RETURN_FAULT_CLRC:CL_INVALID_VALUE] Arguments 'devices'
and 'num_devices' are both NULL

[ERROR] [clEnqueueMapBuffer] [RETURN_FAULT_CLRC:CL_INVALID_EVENT_WAIT_LIST] event
wait list is not NULL, but the number of events provided is 0
```

The debug library's diagnostic error messages can be disabled by setting the environment variable `IBM_OPENCL_TRACE_ENABLE` to '0' or 'false'. For example:

```
export IBM_OPENCL_TRACE_ENABLE=0
```

4.3.2 Debugging OpenCL Kernel Compilation Errors

The `clBuildProgram()` API enables the host application to build the computation kernel for the compute devices. Kernel build errors may be debugged by printing the build log obtained by using the OpenCL `clGetProgramBuildInfo()` API. See the provided OpenCL Samples for examples of how to extract a build log.

In addition, the OpenCL specification allows various build options to be passed to the `clBuildProgram()` API call. Section 5.4.3 has the complete list, but the following ones may especially be useful for debugging build failures.

- **`-Werror`** – make all warnings into errors
- **`-cl-opt-disable`** – disable compiler optimization

Note: *The `–cl-opt-disable` compiler flag can not be used with an optional `reqd_work_group_size` attribute value other than 1,1,1. Non-unity sizes will be flagged with a compilation error. Likewise, `-cl-opt-disable` will result in a `CL_KERNEL_WORK_GROUP_SIZE` of 1.*

4.3.3 Debugging OpenCL Kernel Runtime Errors

Debugging an OpenCL computation kernel can be more challenging than a standard standalone C program as the OpenCL runtime is responsible for building, scheduling and executing the kernels.

Debug Library

Currently, a fatal error in a computation kernel results in the OpenCL application aborting. As a result, calls to the OpenCL `clGetEventInfo()` API with the `CL_EVENT_COMMAND_EXECUTION_STATUS` can never return a negative status. In this situation, it is best to use the debug library as the fatal error will result in a signal that can be trapped by the OpenCL debugger, `ocl-gdb`.

Debugging with printf

The OpenCL compute devices support '`printf`' statements. Use vector component notation to print vector component values. For example:

```
int4 var;
printf(“var = %x %x %x %x\n”,var.x, var.y, var.z, var.w);
```

Note: *When using the '`printf`' function, '`stdio.h`' must **not** be included. Inclusion of the header file may result in compilation errors.*

As defined by the OpenCL Specification, long variable types are 64-bit and thus must be printed using the 'll' (two lower case L's) printf format specifier to ensure the entire 64-bits are printed.

Use the values from `get_global_id()`, `get_local_id()`, and `get_group_id()` kernel built-in function calls to associate computation results with work units. Section 6.11.1 of the OpenCL specification provides a description of these routines. The IBM Debug extension's `get_unit_id()` kernel built-in function can also be used to associate computation results with specific compute units.

Debugging with GDB

IBM has enhanced gdb to enable the debug of OpenCL applications. This enhanced debugger, called 'ocl-gdb', supports the debug of both OpenCL host applications as well as either CPU or SPU accelerator kernel programs. The ocl-gdb application is available as a separate install. See OpenCL GDB Debugger Install Procedure for instructions on installing ocl-gdb.

Running the OpenCL host application within ocl-gdb on a Power or CBEA system is helpful for diagnosing segmentation faults, setting breakpoints, single stepping code, displaying variables, etc. It is recommended that developers use the OpenCL debug runtime library when using ocl-gdb (see Debugging OpenCL Host API Library Calls Errors for further details). OpenCL computation kernels built by applications using the debug library are automatically compiled with the '-g' flag so that debug information is included. In addition, compiled kernel programs result in the creation of a kernel source file called 'IBM_OpenCL_kernel.cl' in the current working directory. This file is used to do source level debugging.

Warning: *Programmers should avoid using the file name 'IBM_OpenCL_kernel.cl' to ensure that their file is not inadvertently overwritten during debug program build.*

Once ocl-gdb has started, it is possible to set breakpoints within the OpenCL computation kernel prior to it being loaded. These are referred to as 'pending' break points. For example, the following sample output is a debug session of the Perlin noise sample (see OpenCL Samples). Since the host application has functions with identical names as the kernel, the breakpoint must be prefixed by the source filename, 'IBM_OpenCL_kernel.cl'. Some output text has been removed to improve readability.

```
$ export LD_LIBRARY_PATH=/usr/lib/CL/debug:$LD_LIBRARY_PATH
$ ocl-gdb ./perlin
(gdb) break IBM_OpenCL_kernel.cl:compute_perlin_noise
Function "compute_perlin_noise" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (IBM_OpenCL_kernel.cl:compute_perlin_noise) pending.
(gdb) run
Starting program: perlin_noise/ppc/perlin
[Thread debugging using libthread_db enabled]
[New Thread 0xf7bdf4b0 (LWP 393)]
[New Thread 0xf714f4b0 (LWP 394)]
[New Thread 0xf674f4b0 (LWP 395)]
Compiling and Creating kernel...
2D Work Group Size = NULL
Global Work Group Size 256 x 1024
Compute Device Data
[Switching to Thread 0xf7bdf4b0 (LWP 393)]

Breakpoint 1, compute_perlin_noise (output=0xe0000080, time=0, rowstride=1024)
at IBM_OpenCL_kernel.cl:200
200       float4 vt = (float4) time;
(gdb)
```

Printing vector types within ocl-gdb is also possible. For example, a float4 is displayed as a float array with 4 values. Here's an example of printing the float4 parameter variable 't' of the subroutine fade:

```
(gdb) break fade
(gdb) continue
Continuing.

Breakpoint 2, fade (t=...) at IBM_OpenCL_kernel.cl:76
76        return t * t * t * (t * (t * 6.0f - 15.0f) + 10.0f);
(gdb) p t
```

```
$1 = {0, 0.0625, 0.125, 0.1875}
(gdb)
```

Likewise, `ocl-gdb` supports OpenCL C's component access syntax. For example, the odd elements of the float4 vector '`t`' can be displayed using:

```
(gdb) p t.odd
$2 = {0.0625, 0.1875}
```

Often it is easier to debug OpenCL applications when they are compiled without optimization and execute on a single compute unit. To disable optimization, include the option string '`-cl-opt-disable`' in the options parameter of `clBuildProgram()` or `cluCreateKernel()`, if using CLU. For example:

```
rc = clBuildProgram(program, 0, NULL, "-cl-opt-disable", NULL, NULL);
```

or

```
kernel = cluCreateKernel(clu, commands, filename, kernel_name, "-cl-opt-disable",
                         CLU_SOURCE);
```

To restrict the number of compute units to 1, set the environment variable `IBM_OPENCL_DEBUG_SINGLE_UNIT`. For example:

```
export IBM_OPENCL_DEBUG_SINGLE_UNIT=Y
```

Note: *The `IBM_OPENCL_DEBUG_SINGLE_UNIT` environment variable is only observed when using the OpenCL debug library. See section 4.3.1.*

A subset of the built-in functions can be invoked from the debugger. These minimally include the following commonly used functions.

Type of Built-In Functions	As Specified in Section	Functions
Explicit Conversions	6.2.3	`convert_`<destType>`()`
Reinterpretations	6.2.4.2	`as_`<destType>`()`
Work-Item	6.11.1	`get_work_dim()`, `get_num_groups()`, `get_group_id()`, `get_global_offset()`, `get_global_size()`, `get_global_id()`, `get_local_size()`, `get_local_id()`,

[5] The get_unit_id built-in is available only when using the OpenCL debug library with devices that support the cl_ibm_debug extension. See section 5.3.

		and `get_unit_id()`[5].
Math[6]	6.11.2	`ceil()`, `cos()`, `exp()`, `exp10()`, `exp2()`, `fabs()`, `floor()`, `fmax()`, `fmin()`, `log()`, `log10()`, `log2()`, `pow()`, `powr()`, `round()`, `rsqrt()`, `sin()`, `sqrt()`, `tan()`, and `trunc()`.

Some of the functions (for example., the work-item functions) are implemented as asynchronous inferior function calls. When the inferior function is running, all the threads are also running which means one of the other threads may run into a kernel break before the inferior function completes. There are several methods of working around this behavior. They include:

- Run single threaded using the `IBM_OPENCL_DEBUG_SINGLE_UNIT` environment variable.
- Clear break points from the other threads prior to invoking the inferior function.
- Instruct `gdb` to run only a single thread instead of all threads using '`set scheduler-locking on`' before making the call.

4.4 Best Practices

The OpenCL Development Kit for Linux on Power provides the OpenCL programmer with a compiler and runtime to execute OpenCL applications on accelerated and non-accelerated Power Systems. OpenCL is a portable environment that allows applications to perform well on a wide range of hardware. This section outlines what an OpenCL application programmer should consider when targeting IBM Power Systems. This section assumes that the reader is familiar with the OpenCL specification and its terminology.

The section Implementation Details documents attributes of the supported hardware devices. It is important that the OpenCL programmer structures their application to map to the strengths of the system it will be deployed on. The IBM provided samples (see OpenCL Samples) demonstrate many of the programming best practices outlined in this section.

4.4.1 Programming Models

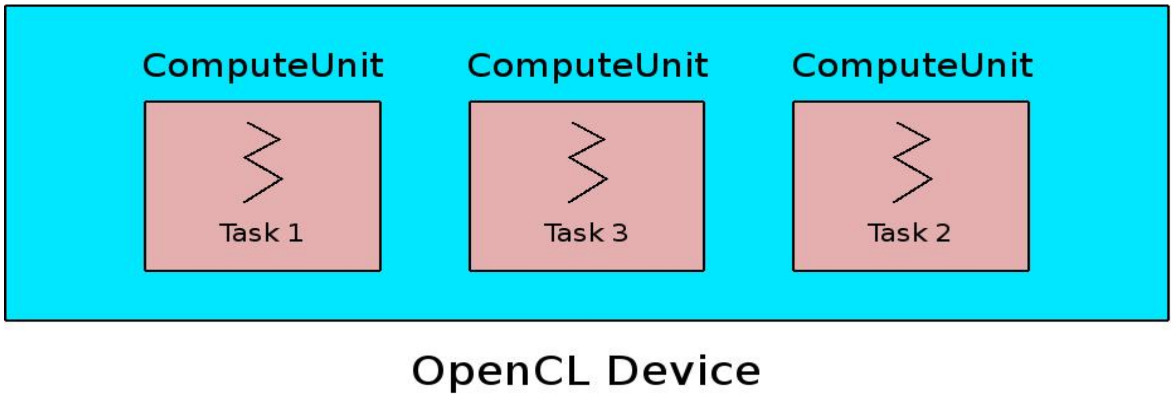
OpenCL defines data and task parallel programming models. Each model offers the OpenCL programmer different levels of control of the flow of data through the application's kernels. This section defines the two parallel programming models, their strengths and their weaknesses. Sections Targeting CBEA Systems and Targeting Power Systems outline how to map these models effectively to each type of IBM system.

Task Parallelism

The OpenCL task parallel model is similar to the POSIX threads (pthreads) programming model. Each compute unit serves as an available execution unit, much like a CPU, that the runtime can dispatch a task onto, much like a POSIX thread. Unlike pthreads, tasks are not preempted. The OpenCL runtime will dispatch each task on a single Compute Unit. The application programmer must execute multiple

[6] Math built-ins invoked by the debugger should be conformant to the accuracy specified by the OpenCL standard; however, they may not produce results that are binarily equal to the kernel's implementation of the built-ins.

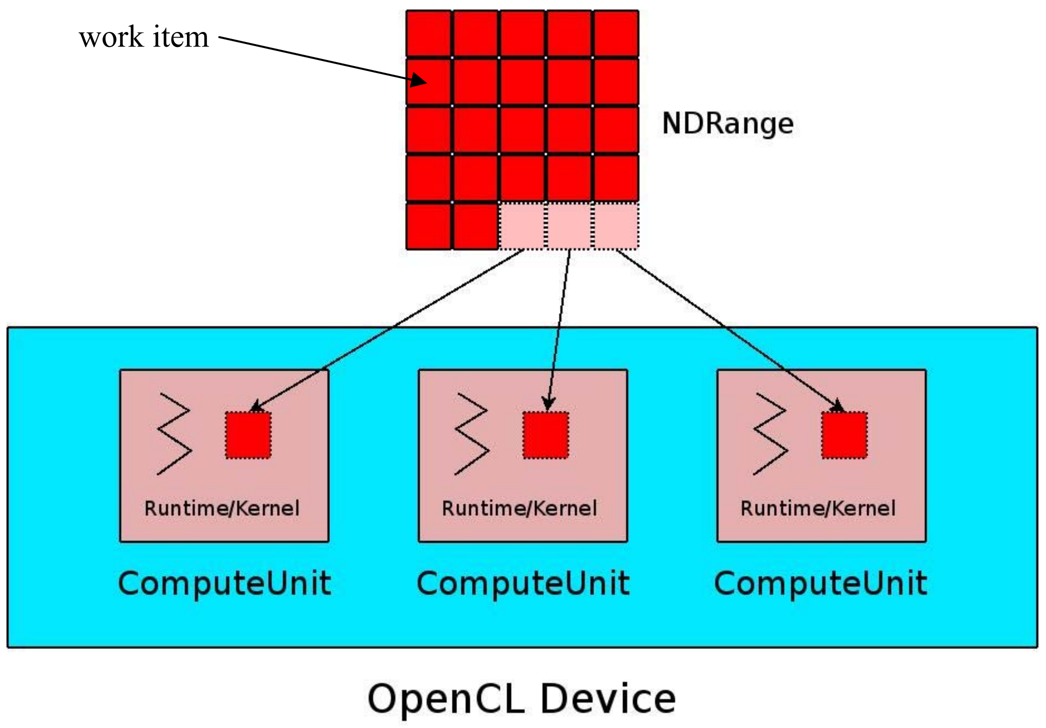
tasks on an out-of-order command queue, or on multiple in-order command queues, to use more than one compute unit per device. See the Command Queues section for more information. Due to the overhead needed to dispatch a task, applications should avoid short running tasks. Tasks are enqueued to a command queue through the OpenCL `clEnqueueTask()` API call.



Tasks shift the burden of scheduling work off of the OpenCL runtime and onto the application's kernels. Tasks should be used when the OpenCL programmer is comfortable explicitly managing the flow of data. This control allows the OpenCL programmer to extract the maximum performance from the target hardware. The OpenCL language and runtime provide a set of APIs to help OpenCL kernels manage data flow through multi-buffering or staging data. Sections Targeting CBEA Systems and Targeting Power Systems discuss these APIs and how they work on each platform.

Data Parallelism

In the OpenCL data parallel model, the OpenCL application describes its work as an N-Dimensional Range (NDRange). The OpenCL runtime iterates over this range of work-groups and dispatches each work-group onto a compute unit. Multiple compute units may execute an NDRange concurrently, provided the range comprises more than one work-group. A compute unit may be able to maximize concurrency within a work-group by scheduling multiple work-items from that work-group on different vector lanes in the compute unit's vector engine or by interleaving their execution to reduce instruction and/or memory latency stalls. The OpenCL application should, therefore, try to maximize the work-group size to ensure that the runtime can fully saturate the compute unit. Data parallel operations are enqueued through the OpenCL `clEnqueueNDRange()` API call.



The OpenCL runtime is responsible for scheduling data parallel work across the compute units. The OpenCL data parallel model is a more convenient model for an application whose work is well structured. Some devices, such as the SPU accelerator, have lower latency memory where `__local` variables are stored (i.e., local storage). These systems can explicitly stage `__global` data into `__local` variables to help maximize performance. All work-items in a work-group share `__local` variables and can use them to stage loads and stores from or to `__global` memory, or intermediate data common to all work-items. The section Targeting CBEA Systems describes the OpenCL language and runtime features that allow an application to directly manage caching within a work-group.

4.4.2 Memory Model

The OpenCL Development Kit for Linux on Power uses a memory model where the application, OpenCL accelerator devices, and OpenCL CPU device share a common memory bus. On systems, where host and device memory are on separate buses, there is a performance penalty incurred when sharing a memory object between the application and the device. This requires that the cost of transferring a memory object across the bus must be weighed against the performance advantage of running on that device. The common bus architecture found on IBM Power Systems eliminates this performance penalty and allows the application more flexibility in choosing which device it uses for each stage in its computation pipeline.

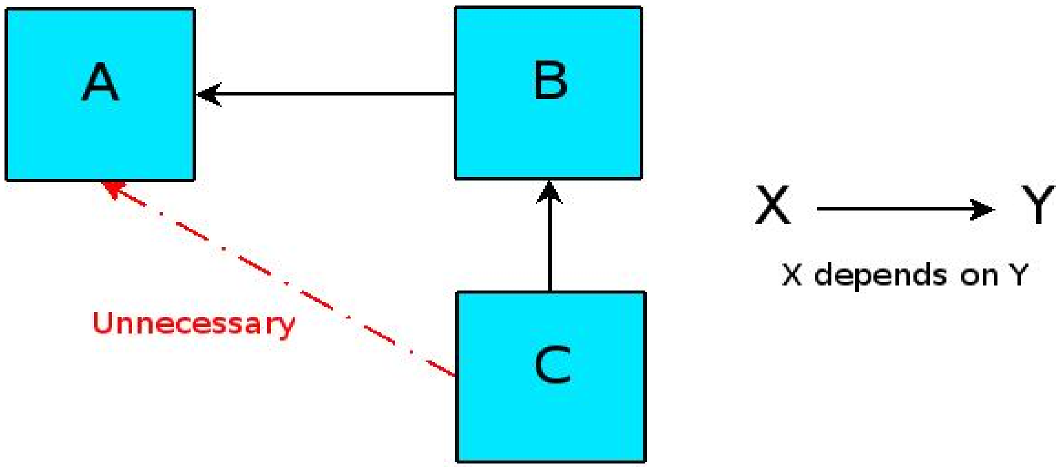
The OpenCL application must use `clEnqueueMapBuffer()`, `clEnqueueReadBuffer()`, `clEnqueueReadBufferRect()`, `clEnqueueWriteBuffer()`, `clEnqueueWriteBufferRect()`, or `clEnqueueNativeKernel()` to examine or modify memory buffers. The `clEnqueueReadBuffer()`, `clEnqueueReadBufferRect()`, `clEnqueueWriteBufferRect()`, and `clEnqueueWriteBuffer()` APIs require the OpenCL runtime to copy the contents of a memory buffer into a region of memory provided by the OpenCL application. This extra copy is inefficient and can be eliminated by mapping the memory buffer (`clEnqueueMapBuffer()`). Since the application, CPU and accelerator devices share a memory bus, it is counter-productive to try to implement a double-buffering scheme with read and write buffer routines.

It is recommended that buffer objects are allocated using `CL_MEM_ALLOC_HOST_PTR` instead of `CL_MEM_USE_HOST_PTR`. Doing so allows the OpenCL implementation to exploit huge pages and NUMA.

IBM's OpenCL runtime implementation supports both 32-bit and 64-bit[7] applications. For 32-bit applications, the total address space for the OpenCL application and runtime is limited to 4GB. These applications should be careful in managing its memory and memory objects so it does not exhaust its address space. IBM's OpenCL runtime implementation does not benefit from using multiple contexts, so the application should use a single context and create all memory buffers and command queues in that context.

4.4.3 Event Dependencies

OpenCL models command dependencies with event wait lists. Events are optionally returned from `clEnqueue*()` calls. These events can be grouped together into dependency lists and used as arguments to other `clEnqueue*()` calls to describe the order in which an OpenCL application's commands should be executed. Events are a managed resource and should be released when they are no longer needed to describe further dependencies.



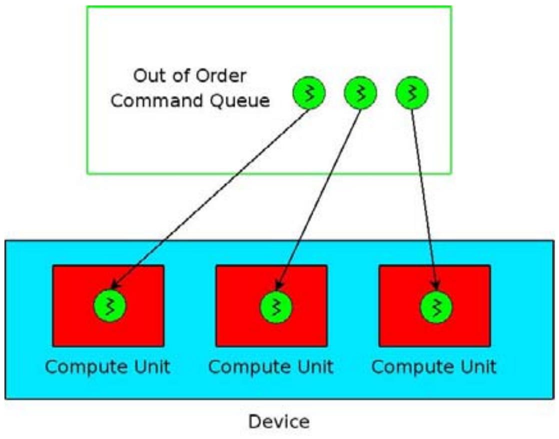
The application can reduce the amount of work required by the runtime to process event dependencies by eliminating redundant dependencies. For example, given three events A, B and C where B depends on A and C depends on A and B. It is unnecessary for C to list both event A and B as dependencies because B is already dependent on A. Instead, the application should list B as dependent on A and C as dependent on B. If the application knows the last event that will finish because it is the last event in the dependency chain, it should issue a `clWaitForEvents()` on that event instead of issuing a `clFinish()`.

[7] 64-bit applications are supported on Power6 and Power7 CPU devices only.

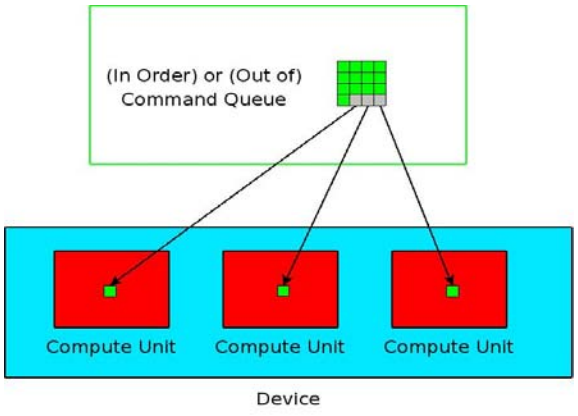
4.4.4 Command Queues

Application programmers should seek to ensure maximum concurrency for their applications to perform well. The application can schedule work on multiple compute units in three different ways – using an out-of-order command queue, using an NDRange, or using multiple in-order command queues.

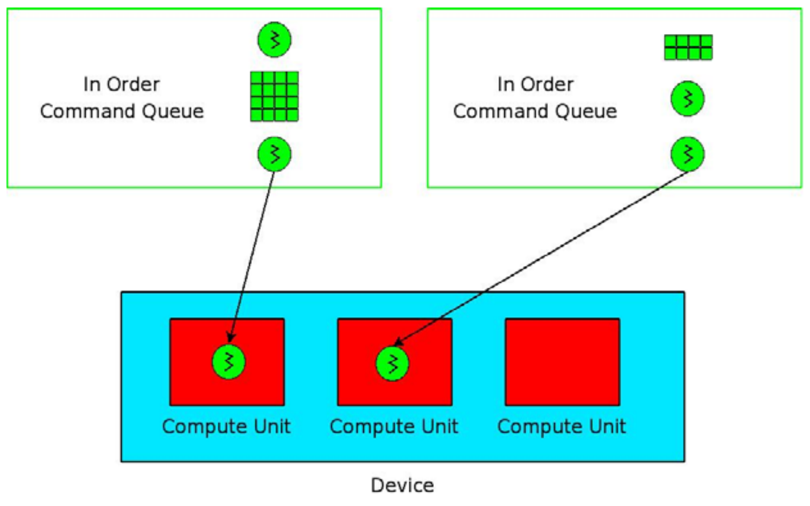
Out-of-Order Command Queue



ND-Range



Multiple In-Order Command Queues



The easiest way to execute work on an OpenCL device is to use an in-order command queue. OpenCL applications that have a serial chain of execution will perform equally on an in-order and out-of-order command queue. However, if multiple tasks or ND-Ranges can execute concurrently, the application needs to either create more than one in-order command queue, or use an out-of-order command queue. When using a single out-of-order queue, it is important for the application to avoid using synchronization points, when possible, as they may block other runnable commands.

An OpenCL application should reduce its use of routines that cause it to synchronize with the command queue (e.g. `clFlush()` and `clFinish()`). Calls to these routines will cause the application to wait for the operation on the command queue to complete and may prevent other, unrelated, work on the command queue from executing until the routine has returned. This may introduce false dependencies on commands that are available for execution, causing them to wait unnecessarily. An application may use `clEnqueueNativeKernel()` if it has native code that it wants to execute on the result buffer of a kernel. This allows the execution of that code to be dispatched as if it were an OpenCL command, avoiding an unnecessary synchronization point. Native kernels are dispatched by the command queue when their event dependencies have cleared. They may affect the CPU device's performance because they share compute resources.

If the application needs to conditionally enqueue additional work onto the command queue, it will need to synchronize with the command queue. This is because native kernels are not allowed to enqueue new commands on the command queue.

4.4.5 Kernel Runtime Considerations

The OpenCL runtime manages a kernel's arguments. The application should therefore try to reduce the number of arguments that a kernel takes as parameters. Parameters that are changed for each invocation should be folded into a single structure. `__local` arguments should only be used if their sizes must be dynamically allocated. `__local` variables that are statically sized should be defined in the program's text. A kernel will retain its parameters across calls to `clEnqueueNDRange()` and `clEnqueueTask()`. The application should only call `clSetKernelArg()` for parameters that have changed since the previous invocation.

The OpenCL runtime supports work-group sizes that are larger than one. If the kernel has a required workgroup size, the programmer can specify the size using the `reqd_work_group_size` kernel attribute. If the kernel does not have a required workgroup size, an optimal workgroup size is chosen by the compiler during compilation. When the kernel is enqueued using `clEnqueueNDRangeKernel()`, the program can use the compiler chosen size by specifying a `NULL` local_work_size parameter, or choose it own size as long as its no larger than the maximum work group size (`CL_KERNEL_WORK_GROUP_SIZE`). The OpenCL application should group work-items with high spatial coherency (data locality). These work-items will be scheduled as a work-group and may benefit from more efficient cache use.

In-order to avoid casting issues between C99 host types and OpenCL kernel types, it is preferable for the application host code to use the OpenCL `cl_*` types provided. This is especially important when the host type differ in size from the OpenCL kernel type. For example, the long type may be either 32 or 64 bits on the host and is always 64 bits in the kernel. The OpenCL application programmer should also pay special attention to `char` types, where the signed or unsigned nature of the native type may be ambiguous. To ensure portability, all char types should be attributed with either signed or unsigned.

OpenCL provides a convenient way for application programmers to write OpenCL vector code that is portable across all IBM processors supported by the OpenCL Development Kit. In addition to any vector code within an application's kernels, the OpenCL compiler may auto-vectorize[8] code within a work-item, and across work-items within a work-group.

An OpenCL application competes for resources with all other applications on the system. When running OpenCL applications, it is important to reduce over commitment of system resources. The OpenCL runtime will try to run on all available CPUs and accelerators (SPUs). The OpenCL runtime will not release these resources until the contexts associated with them are released.

4.4.6 Build Considerations

OpenCL provides a set of APIs to create programs from source or pre-built binaries. Building programs from source requires the OpenCL runtime to invoke the compiler during application execution. For applications or kernels that will be run repeatedly, it is more efficient to build the program from source once, use `clGetProgramInfo()` to retrieve the binary, and then save it to a file. On subsequent runs, the application can use `clCreateProgramWithBinary()` instead of `clCreateProgramWithSource()` to reduce the run-time. The `opencl_build_program` utility is also provided to aid in creating a program binary. See Compiling OpenCL Kernels for additional information.

If an application does not require strict IEEE mathematical compliance, the OpenCL kernels can be built with the `-cl-fast-relaxed-math` compile option. This will allow the compiler to include performance optimizing, code transformations including:

- Floating-point conditionals may be transformed such that strict compare ordering in the presence of NaNs may not be preserved.
- Floating point divides may be transformed into a reciprocal-multiply.

[8] The auto-vectorizing feature of the XL C for OpenCL kernel compiler is currently disabled. It is expected auto-vectorization may in the future be supported.

- Software support of infinites and NaNs is omitted for `half_divide` and `half_recip` built-ins (SPE accelerator device only).

4.4.7 Targeting Power Systems

An OpenCL application may use the system's CPUs as a compute device. Each compute unit is mapped to a compute thread; the CPU device comprises the collection of all CPU compute threads in the system. CBEA and IBM Power systems each have a single CPU device that comprises all CPUs on the system.

Workloads that have scattered memory access patterns or complex control logic map well to OpenCL Tasks on IBM Power Systems like the IBM BladeCenter PS701. Applications that walk through irregular data structures or have a lot of conditional branches will benefit from the hardware branch prediction and caching that IBM Power processors provide.

On Power processors, the same hardware cache is used for `__private`, `__local`, `__constant` and `__global` variables. It is counterproductive to create `__local` variables to stage `__global` memory. Instead, the application should take special care to layout `__global` memory so that the data for work-items in a work-group is cache friendly. The kernel API call `async_work_group_copy()` is implemented as a copy and should be avoided on the CPU device. The OpenCL application will not benefit from the `prefetch()` call as it is currently a no-op (i.e., does nothing) on the CPU device.

For large CPU systems like the IBM Power 780 server or IBM Power 795 server, it is recommended that the CPU device be partitioned into multiple sub-devices along NUMA boundaries to ensure good scaling as the number of compute units reaches 128 and beyond. See section 4.4.11 NUMA Affinity for additional details.

4.4.8 Targeting CBEA Systems

The OpenCL Development Kit for Linux on Power provides an accelerator device on CBEA systems like the QS22. This device exposes all 16 SPUs compute units.

Workloads that have simple control logic or high bandwidth regular access patterns map well to SPUs. These workloads map well to NDRanges because their access patterns are well defined. In addition to NDRanges, the OpenCL application programmer can create an OpenCL task that implements a double-buffering scheme by managing local store directly with `async_work_group_copy()` and `__local` variables. The SPU's ability to efficiently execute the OpenCL task and data parallel programming models allows it to run a wide range of workloads.

On CBEA systems, OpenCL applications should maximize the amount of work done on the accelerator device. Each SPU is a compute unit in the OpenCL accelerator device. The OpenCL runtime will schedule work-groups across and execute work items on the SPUs. The section entitled Command Queues describes how best to structure an application for maximum concurrency.

4.4.9 Targeting SPUs

Each SPU has 256K of local storage that will be divided among the OpenCL kernel runtime, an 8KB global data cache, and the OpenCL program's text, `__local`, `__constant`, and `__private` variables. An OpenCL program may contain one or more OpenCL kernels that share local storage. Kernels that require large amounts of local storage for `__local`, `__constant`, and `__private` variables may have to reduce their work-group size because of the lack of local storage. These kernels should, instead, be separated into their own program so other kernels' resources do not limit their work-group size. However, kernels that do not require large amounts of local storage should be grouped together into the same program. OpenCL applications that group kernels together into a program may avoid unnecessary context-switching because all kernels in a program are loaded together.

Proper management of data flow into and out of the SPU is crucial to maximize performance. This includes managing local storage effectively by staging data whenever possible. The OpenCL runtime utilizes a software data cache that caches accesses to `__global` memory in local storage. When possible, it is preferable for a kernel to aggregate all loads for a work-group into a single `async_work_group_copy()` to a `__local` variable. This will improve performance by grouping all of the work-items load latencies into one common load or store. The load will also be larger, making more efficient use of the DMA engine. If all accesses to `__global` memory are issued with `async_work_group_copy()` instead of direct access through the `__global` pointer, the software data cache will not be included, saving roughly 16KB of local storage.

An OpenCL task may also implement a double-buffering scheme. Two or more __local variables can be used as buffers to stage data. The kernel can then initiate an `async_work_group_copy()` into one buffer, then compute the results on the second buffer. The `async_work_group_copy()` built-in will use the SPU's DMA engine to copy data while the SPU's vector engine is free to operate on the second buffer. Double-buffering maximizes performance by keeping the compute engine busy by eliminating the need to wait on data transfers.

The OpenCL application should use `__global` memory buffers whose type's size is a multiple of a quad-word (16 bytes). For example, a kernel that operates sequentially on an array of floats should instead aggregate four floats together and operate on a float4 vector. This will allow the OpenCL compiler to map vector operations to the SPU's native vector types, and optimize its use of the SPU's DMA engine by eliminating alignment checks. Code that uses large vectors (that are a multiple of a quad-word, e.g. float16) is easier to read than hand unrolling loops. The large vectors will be automatically unrolled by the compiler to operate efficiently on the SPU's vector engines.

4.4.10 Huge Page Support

By allocating data buffers from huge, 16MB pages, some applications can a see significant performance benefit as a result of a reduction in page table or TLB (translation lookaside buffer) misses. OpenCL will preferentially allocate buffer objects from huge pages if the system has been configured with a pool of hugepages and the buffer objects are created **not** using the `CL_MEM_USE_HOST_PTR` flag.

If your system has not been configured with a hugetlbfs, perform the following commands as root:

```
mkdir -p /huge
mount -t hugetlbfs nodev /huge
```

Hugepages can be allocated using the `/proc/sys/vm/nr_hugepages` entry or by using the Linux `sysctl` command. For example, to view the currently hugepage setting using the `sysctl` command:

```
/sbin/sysctl vm.nr_hugepages
```

To set the number of hugepages to 10:

```
/sysbin/sysctl –w vm.nr_hugepages=10
```

A reboot may be necessary to be able to allocate all the requested hugepages because hugepages require large areas of contiguous physical memory. Over time, physical memory can become fragmented due to repeated mapping and unmapping. If hugepages are allocated early during the boot process, fragmentation is unlikely. Therefore, is it recommended that the `/etc/syscl.conf` file be used to allocate hugpages at boot time. For example, to allocate 10 16MB pages at boot time, add the following line to `/etc/sysctl.conf`:

```
vm.nr_hugepages = 10
```

4.4.11 NUMA Affinity

Applications that are memory bound can benefit greatly by ensuring good memory locality in systems with non-uniform memory access (NUMA). OpenCL provides programmers the ability to manage the memory locality through two extensions – device fission and migrate memory object. The device fission extension provides programmers the ability to partition a device into multiple sub-devices along NUMA boundaries. The migrate memory object extension gives programmers the ability to control the NUMA memory domain of memory objects.

The following code snippets demonstrate how to optimize memory locality on NUMA boundaries. For the sake of simplicity, these code snippets ignore error and return value checking. A well written program will check return values and handle errors accordingly. Complete code examples can be found in the IBM OpenCL Samples.

First the program must get the new API function entry points using the OpenCL `clGetExtensionFunctionAddress()` API.

```
clCreateSubDevicesEXT_fn clCreateSubDevicesEXT;
clReleaseDeviceEXT_fn clReleaseDeviceEXT;
clEnqueueMigrateMemObjectEXT_fn clEnqueueMigrateMemObjectEXT;

clCreateSubDevicesEXT = clGetExtensionFunctionAddress(“clCreateSubDevicesEXT”);
clReleaseDeviceEXT = clGetExtensionFunctionAddress(“clReleaseDeviceEXT”);
clEnqueueMigrateMemObjectEXT = clGetExtensionFunctionAddress(“clEnqueueMigrateMemObjectEXT”);
```

Next, use the device fission extension to create sub-devices and command queues along NUMA domains. It is good programming practice to first query the device using `clGetDeviceInfo()` to determine if the implementation supports the NUMA partitioning scheme.

```
cl_uint num_devices;
cl_device_id root_device, *sub_devices;
cl_context context;
cl_command_queue *cmd_queues;

cl_partition_properties_ext properties[3] = {
   CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT, CL_AFFINITY_DOMAIN_NUMA_EXT,
   CL_PROPERTIES_LIST_END_EXT};

clGetDeviceIDs(NULL, CL_DEVICE_TYPE_DEFAULT, 1, &root_device, NULL);

clCreateSubDevicesEXT(root_device, properties, 0, NUM, &num_devices);
sub_devices = (cl_device_id *)malloc(num_devices * sizeof(cl_device_id));
clCreateSubDevicesEXT(root_device, properties, num_devices, sub_devices, NULL);

context = clCreateContext(NULL, num_devices, sub_devices, NULL, NULL, NULL);
cmd_queues = (cl_command_queue *)malloc(num_devices * sizeof(cl_command_queue));
```

```
for (i=0; i<num_devices; i++) {
    cmd_queues[i] =  clCreateCommandQueue(context, sub_devices[i], 0, NULL);
}
```

Allocate memory buffers for each NUMA domain and migrate the buffers to their correct NUMA domain by enqueueing the migrate command on the command queue corresponding to the NUMA domain.

```
cl_mem *buffers;

buffers = (cl_mem *)malloc(num_devices * sizeof(cl_mem));
for (i=0; i<num_devices; i++) {
    buffers[i] = clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR, size, NULL, NULL);
    clEnqueueMigrateMemObjectEXT(cmd_queue[i], buffers[i], 0, 0, NULL, NULL);
}
```

Kernel commands that access the `buffers[i]` can now be enqueued on `cmd_queues[i]`.

5 OpenCL Extension Specifications

This section documents the supported extensions that are not documented in The OpenCL Specification.

5.1 Device Fission (`cl_ext_device_fission`)

The device fission extension provides OpenCL programmers the ability to split an OpenCL device into multiple sub-devices. There are a number of cases in which a typical user would like to subdivide a device:

- To reserve part of the device for use for high priority / latency-sensitive tasks,
- To more directly control the assignment of work to individual compute units, or
- To subdivide compute devices along some shared hardware feature like a cache.

Typically these are areas where some level of additional control is required to get optimal performance beyond that provided by standard OpenCL APIs. Proper use of this interface assumes some detailed knowledge of the devices in question.

It is a design goal that the device be fissionable post `cl_context` creation. This will enable software layers to avoid having to create a new context in order to fission a device. A new context would force the user to copy all the data to the new context in order to do the operation. Inter-context synchronization would also be awkward because `cl_event` objects don't work across context boundaries.

5.1.1 Procedures and Functions

clCreateSubDevicesEXT

```
cl_int clCreateSubDevicesEXT(cl_device_id in_device,
                             const cl_device_partition_property_ext *properties,
                             cl_uint num_entries,
                             cl_device_id *out_devices,
                             cl_uint *num_devices)
```

`clCreateSubDevicesEXT` 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 the `cl_device_partition_property_ext` list. The output sub-devices may be used in every way that the root device can be used, including building programs, further calls to **`clCreateSubDevicesEXT`** and creating command queues. They may also be used within any context created using the *`in_device`* or parent/ancestor thereof. When a command queue is created against a sub-device, the commands enqueued on that queue are executed only on the sub-device.

`in_device` – The device to be partitioned.

`num_entries` – The number of `cl_device_id`s that will fit in the array pointed to by *`out_devices`*. If `out_devices` is not NULL, *`num_entries`* must be greater than zero.

`out_devices` – On output, the array pointed to by *`out_devices`* will contain up to *`num_entries`* sub-devices. If the *`out_devices`* argument is NULL, it is ignored. The number of `cl_device_id`'s returned is the minimum of *`num_entries`* and the number of devices created by the partition scheme.

`num_devices` – On output, the number of devices that *`in_device`* may be partitioned into according to the partitioning scheme given by *`properties`*. If *`num_devices`* is NULL, it is ignored.

`properties` – A `CL_PROPERTIES_LIST_END_EXT` terminated list of device fission `{property-value, cl int[]}` pairs that describe how to partition the device into sub-devices. It may not be `NULL`. Only one of `CL_DEVICE_PARTITION_EQUALLY_EXT`, `CL_DEVICE_PARTITION_BY_COUNTS_EXT`, `CL_DEVICE_PARTITION_BY_NAMES_EXT`, or `CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT` may be used.

Available properties include:

`CL_DEVICE_PARTITION_EQUALLY_EXT` – 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_MAX_COMPUTE_UNITS` then the remaining compute units are not used.

Example: To divide a device containing 16 compute units into two sub-devices, each containing 8 compute units, pass:

```
{CL_DEVICE_PARTITION_EQUALLY_EXT, 8, CL_PROPERTIES_LIST_END_EXT}
```

`CL_DEVICE_PARTITION_BY_COUNTS_EXT` – This property is followed by a `CL_PARTITION_BY_COUNTS_LIST_END_EXT` terminated list of compute unit counts. For each non-zero count M in the list, a sub-device is created with M compute units. `CL_PARTITION_BY_COUNTS_LIST_END_EXT` is defined to be 0.

Example: To split a four compute unit device into two sub-devices, each containing two compute units, pass:

```
{CL_DEVICE_PARTITION_BY_COUNTS_EXT, 2, 2,
CL_PARTITION_BY_COUNTS_LIST_END_EXT, CL_PROPERTIES_LIST_END_EXT}
```

The first 2 means put two compute units in the first sub-device. The second 2 means put two compute units in the second sub-device. `CL_PARTITION_BY_COUNTS_LIST_END_EXT` terminates the list of sub-devices. `CL_PROPERTIES_LIST_END_EXT` terminates the list of properties.

The total number of compute units specified may not exceed the number of compute units in the device.

`CL_DEVICE_PARTITION_BY_NAMES_EXT` – This property is followed by a list of compute unit names. Each list starts with a `CL_PARTITION_BY_NAMES_LIST_END_EXT` terminated list of compute unit names. Compute unit names are numbers that count up from zero to the number of compute units less one. `CL_PARTITION_BY_NAMES_LIST_END_EXT` is defined to be -1. Only one sub-device may be created at a time with this selector. An individual compute unit name may not appear more than once in the sub-device description.

Example: To create a three compute unit sub-device using compute units 0, 1, and 3, pass:

```
{CL_DEVICE_PARTITION_BY_NAMES_EXT, 0, 1, 3,
CL_PARTITION_BY_NAMES_LIST_END_EXT, CL_PROPERTIES_LIST_END_EXT}
```

`CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT` – 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 `CL_AFFINITY_DOMAIN` list:

`CL_AFFINITY_DOMAIN_NUMA_EXT` – Split the device into sub-devices comprised of compute units that share a NUMA band.

`CL_AFFINITY_DOMAIN_L4_CACHE_EXT` – Split the device into sub-devices comprised of compute units that share a level 4 data cache.

`CL_AFFINITY_DOMAIN_L3_CACHE_EXT` – Split the device into sub-devices comprised of compute units that share a level 3 data cache.

`CL_AFFINITY_DOMAIN_L2_CACHE_EXT` – Split the device into sub-devices comprised of compute units that share a level 2 data cache.

`CL_AFFINITY_DOMAIN_L1_CACHE_EXT` – Split the device into sub-devices comprised of compute units that share a level 1 data cache.

`CL_AFFINITY_DOMAIN_NEXT_FISSIONABLE_EXT` – Split the device along the next fissionable `CL_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 fission the device into sub-devices comprised of compute units that share memory subsystems at this level. The user may determine the resulting partition style by calling `clGetDeviceInfo(CL_DEVICE_PARTITION_STYLE_EXT)` on the sub-devices.

Example: To split a non-NUMA device along the outermost cache level (if any) pass:

```
{CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT,
CL_AFFINITY_DOMAIN_NEXT_FISSIONANLE_EXT, CL_PROPERTIES_LIST_END_EXT}
```

The following values may be returned by **`clCreateSubDevicesEXT`**:

- **`CL_SUCCESS`** – The command succeeded.

- **`CL_INVALID_VALUE`** – The *properties* key is unknown, or the indicated partition style (`CL_DEVICE_PARTITION_EQUALLY_EXT`, `CL_DEVICE_PARTITION_BY_COUNTS_EXT`, `CL_DEVICE_PARTITION_BY_NAMES_EXT`, or `CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT`) is not supported for this device by the implementation. On an OpenCL 1.1 implementation, these cases return `CL_INVALID_PROPERTY` instead, to be consistent with **`clCreateContext`** behavior.

- **`CL_INVALID_VALUE`** – *`num_entries`* is zero and *`out_devices`* is not NULL, or both *`out_devices`* and *`num_devices`* are `NULL`.

- **`CL_DEVICE_PARTITION_FAILED_EXT`** – The indicated partition scheme is supported by the implementation, but the implementation can not further partition the device in this way. For example, `CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT` was requested, but all compute units in *`in_device`* share the same cache at the level requested.

- **`CL_INVALID_PARTITION_COUNT_EXT`** – The total number of compute units requested exceeds `CL_DEVICE_MAX_COMPUTE_UNITS`, or the number of compute units for any one sub-device is less than 1, or the number of sub-devices requested exceeds `CL_DEVICE_MAX_COMPUTE_UNITS`.

- **`CL_INVALID_PARTITION_NAME_EXT`** – A compute unit name appearing in a name list following `CL_DEVICE_PARTITION_BY_NAMES_EXT` is not in the range [-1, number of compute units – 1].

- **`CL INVALID DEVICE`** – The *`in_device`* is not a valid device. The *`in_device`* is not a device in *`context`*.

clReleaseDeviceEXT

```
cl_int clReleaseDeviceEXT(cl_device_id device)
```

`clReleaseDeviceEXT` decrements the *`device`* reference count. After the reference count reaches zero, the object is destroyed and associated resources release for reuse by the system. It has no effect on root level devices. **`clReleaseDeviceEXT`** returns `CL_SUCCESS` if the function is executed successfully or the device is a root level device. It returns `CL_INVALID_DEVICE` if the device is not a valid device.[9]

clRetainDeviceEXT

```
cl_int clRetainDeviceEXT(cl_device_id device)
```

`clRetainDeviceEXT` increments the *`device`* reference count. It has no effect on root level devices. **`clRetainDeviceEXT`** returns `CL_SUCCESS` if the function is executed successfully or the device is a root level device. It returns `CL_INVALID_DEVICE` if the device is not a valid device.

[9] CAUTION: Since root level devices are generally returned by a `clGet*` call (e.g. **`clGetDevicesIDs`**) and not a `clCreate` call, the user generally does not own a reference count for root level devices. The reference count attached to a device returned from **`clGetDeviceIDs`** is owned by the implementation. Developers need to be careful when releasing `cl_device_ids` to always balance **`clCreateSubDevicesEXT`** or **`clRetainDeviceEXT`** with each call to **`clReleaseDeviceEXT`** for the device. By convention, software layers that own a reference count should themselves be responsible for releasing it.

5.1.2 Clarification of Behavior of Existing Functions

`clBuildProgram` – This call will return `CL_INVALID_DEVICE` if the device or one of its ancestors was not used to create the program's context. **`clBuildProgram`** must be called for each device or sub-device that will be used with the program. This function will return `CL_INVALID_DEVICE` if the device was not used to create the program and the program was created with **`clCreateProgramWithBinary`**. Implementations should eliminate redundant compilation where devices and sub-devices can share a common binary.

`clCreateCommandQueue` – This call will return `CL_INVALID_CONTEXT` if the device or one of its ancestors was not used to create the context. On success, the `cl_command_queue` that is created for a sub-device will attempt to execute any work enqueued on it on just the sub-device.

`clCreateContext` – The new `cl_context` only owns a reference to the device explicitly passed to it at creation. If does not reference sub-devices of those devices unless they were also passed in the device list at creation time.

`clCreateContextFromType` – Only root level devices are used to create the context. The context does not reference sub-devices created from those devices.

`clCreateProgramWithBinary` – This call will return `CL_INVALID_DEVICE` if a device or one of its ancestors was not used to create the program's context.

`clGetCommandQueueInfo` – If the command queue was created with a sub-device, the sub-device is returned for the `CL_QUEUE_DEVICE` selector.

`clGetContextInfo` – Only the device list passed into **`clCreateContext`** is returned by `CL_CONTEXT_DEVICES`. Additional sub-devices usable in the context, because their parent devices were used to create the devices, are not returned here.

`clGetDeviceIDs` – Only root level devices shall be returned by this function. Devices returned by this function should not contain compute units that alias compute units on another returned in the same function call. Compute units in the root level devices shall not alias compute units in other root level devices.

`clGetDeviceInfo` – If the device is a sub-device created by **`clCreateSubDevicesEXT`**, then the value returned for `CL_DEVICE_MAX_COMPUTE_UNITS` is the number of compute units in the sub-device. The `CL_DEVICE_VENDOR_ID` may be different from the parent device `CL_DEVICE_VENDOR_ID`, but should be the same for all devices and sub-devices that can share a binary executable, such as that returned from **`clGetProgramInfo`**`(CL_PROGRAM_BINARIES)`. Other selectors such as `CL_DEVICE_GLOBAL_MEM_CACHE_SIZE` may optionally change value to better reflect the behavior of the sub-device in an implementation defined manner.

The following selectors are added for **`clGetDeviceInfo`**:

`CL_DEVICE_PARENT_DEVICE_EXT` - a selector to get the `cl_device_id` for the parent `cl_device_id` to which the sub-device belongs. (Sub-division can be multi-level.) If the device is a root level device, then it will return `NULL`.

`CL_DEVICE_PARTITION_TYPES_EXT` – a selector to get a list of supported partition types for partitioning a device. The return type is an array of *`cl_device_partition_property_ext`* values drawn from the following list:

```
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT
CL_DEVICE_PARTITION_BY_COUNTS_EXT
CL_DEVICE_PARTITION_BY_NAMES_EXT
CL_DEVICE_PARTITION_EQUALLY_EXT
```

The implementation shall return at least on property from the above list. However, when a partition style is found within this list, the partition style is not required to work in every case. For example, a device might support partitioning by affinity domain, but not along NUMA domains.

`CL_DEVICE_AFFINITY_DOMAINS_EXT` – a selector to get a list of supported affinity domains for partitioning the device using `CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT` partition style. The return type is an array of *`cl_device_partition_property_ext`* values. The values shall come from the list:

```
CL_AFFINITY_DOMAIN_L1_CACHE_EXT
CL_AFFINITY_DOMAIN_L2_CACHE_EXT
CL_AFFINITY_DOMAIN_L3_CACHE_EXT
CL_AFFINITY_DOMAIN_L4_CACHE_EXT
CL_AFFINITY_DOMAIN_NUMA_EXT
```

If no partition style is supported, then the size of the returned array is zero. Even though a device has a NUMA, or a particular cache level, an implementation may elect to not provide fissioning at that level.

`CL_DEVICE_REFERENCE_COUNT_EXT` – returns the device reference count. The return type is *`cl_uint`*. Return the device reference count. If the device is a root level device, a reference count of 1 is returned.

`CL_DEVICE_PARTITION_STYLE_EXT` - a selector to get the `cl_device_partition_property_ext` list used to create the sub-device. If the device is a root level device then a list consisting of {`CL_PROPERTIES_LIST_END_EXT`} is returned. If the property on device creation was (`CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT`, `CL_AFFINITY_DOMAIN_NEXT_FISSIONABLE_EXT`) then `CL_AFFINITY_DOMAIN_NEXT_FISSIONABLE_EXT` will be replaced by the symbol representing the actual `CL_AFFINITY_DOMAIN` used (e.g. `CL_AFFINITY_DOMAIN_NUMA_EXT`). The returned value is an array of *`cl_device_partition_property_ext`*. The length of the array is obtained from the size returned by the *`param_size_value_ret`* parameter to the function.

`clGetProgramInfo` – Behavior depends on how the program was created:

`clCreateProgramWithSource` – `CL_PROGRAM_DEVICES` returns the union of the set of devices used to create the `cl_context` to which the program belongs and the set of devices for which **`clBuildProgram`** has succeeded on this program.

`clCreateProgramWithBinary` – `CL_PROGRAM_DEVICES` returns the union of the set of devices used to create the program and the set of devices for which **`clBuildProgram`** has succeeded on this program.

In each case, `CL_PROGRAM_BINARIES` and `CL_PROGRAM_BINARY_SIZES` return the same list in the same order, except that pointers to compiled binaries or their sizes are returned instead, respectively. The user is responsible for preventing the obvious race condition here, in which **`clBuildProgram`** succeeds on more devices between when `CL_PROGRAM_DEVICES`, `CL_PROGRAM_BINARIES` and `CL_PROGRAM_BINARY_SIZES` are requested.

5.1.3 Glossary Entries

sub-device – An OpenCL device can be subdivided into multiple sub-devices using **`clCreateSubDevicesEXT`**. The new sub-devices alias specific collections of compute units within the parent device, according to a partition scheme. Provided that programs are rebuilt for them, the sub-devices may be used in any situation that their parent device may be used. Subdividing a device does not destroy the parent device, which may continue to be used along side and intermingled with its child sub-devices.

parent device – The cl device id in device used with **`clCreateSubDevicesEXT`** to produce a sub-device. 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 root devices.

root device – A root device is a device that has not been sub-divided using **`clCreateSubDevicesEXT`**. See also device.

5.2 Migrate Memory Object (`cl_ext_migrate_memobject`)

The Migrate Memory Object extension defines 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 or sub-device prior to usage.
- Preemptively migrate an object from one device to another.

Some implementations may choose to defer the allocation of memory objects until after they are first used. This may lead users to needlessly copying data to the buffer using **`clEnqueueWriteBuffer`** to affect allocation policies. The use of **`clEnqueueMigrateMemObjectEXT`** avoids this extra copy while allowing the user to direct memory object placement.

An application may use the memory object migration command to move a memory object to a specific device. This usage allows the transfer to occur before the memory object is needed, potentially hiding transfer latencies.

The device fission extension allows an application to divide a device into sub-devices along affinity domains. If each sub-device describes an affinity domain, this command can be used to influence the association of the memory object with the specific domain.

The memory object migration command 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. The user host program is responsible for managing the event dependencies associated with such usage. Improperly specified event dependencies passed to **`clEnqueueMigrateMemObjectEXT`** could produce undefined results.

5.2.1 Procedures and Functions

`clEnqueueMigrateMemObjectEXT`

```
cl_int clEnqueueMigrateMemObjectEXT(cl_command_queue command_queue,
                                    cl_uint num_mem_objects,
                                    const cl_mem *mem_objects,
                                    cl_mem_migration_flags_ext flags,
                                    cl_uint num_events_in_wait_list,
                                    const cl_event *event_wait_list,
                                    cl_event *event);
```

`clEnqueueMigrateMemObjectEXT` provides the application with a way to indicate which device a set of memory objects should be associated. Typically, memory objects are implicitly migrated to a device for which enqueued commands, using the memory object, are targeted. **`clEnqueueMigrateMemObjectEXT`** allows this migration to be explicitly performed ahead of dependent commands. This permits an application to overlap the placement of memory object with other unrelated operations. Once the OpenCL event, returned from **`clEnqueueMigrateMemObject`**, has been marked CL_COMPLETE, the memory objects specified in *`mem_objects`* have been successfully migrated to the device associate with *command_queue*. The migrated memory objects shall remain resident on the device until another command is enqueued that either implicitly or explicitly migrates it away. As well, **clEnqueueMigrateMemObjectEXT** can 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 **clEnqueueMigrateMemObjectEXT** could produce undefined results.

command_queue – is a valid 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 application host if the CL_MIGRATE_MEM_OBJECT_HOST_EXT flag is specified.

num_mem_object – is the number of memory objects specified in the *mem_objects* argument.

mem_objects – is a pointer to a list of valid memory objects.

flags – is a bit-field that is used to specify the migration options. Possible values for *flags* other than 0 include:

cl_mem_migration_flags_ext	**Description**
CL_MIGRATE_MEM_OBJECT_HOST_EXT	This flag specifies that the specified set of memory objects are to be migrated to the host, regardless of the target 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, then this particular command does not wait on any event to complete. If *event_wait_list* is **NULL**, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same.

event – returns an event object that identifies this particular command and can be used to query or queue a wait for this particular command to complete. *event* can be NULL in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

clEnqueueMigrateMemObjectEXT returns CL_SUCCESS if the operation was successfully queued. Otherwise, it returns one of the following errors:

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

- **`CL_INVALID_VALUE`** if *`mem_objects`* is `NULL`.
- **`CL_INVALID_VALUE`** if *`flags`* is not 0 or `CL_MIGRATE_MEM_OBJECT_HOST_EXT`.
- **`CL_INVALID_MEM_OBJECT`** if *`mem_objects`* contains an invalid memory object.
- **`CL_MEM_OBJECT_ALLOCATION_FAILURE`** if there is a failure to allocate memory of the specified set of memory objects in *`mem_objects`*.
- **`CL_OUT_OF_RESOURCES`** if there is a failure to allocate resource required by the OpenCL implememtation on the host.
- **`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.

5.3 IBM Debug (`cl_ibm_debug`)

The IBM debug extension provides a kernel built-in to obtain its compute unit's identifier. The compute unit's identifier, along with the work item functions described in section 6.11.1 of the OpenCL 1.1 Specification, is useful in discovering scheduling characteristics of the system and may be of assistance when performance debugging. It can also be used to limit the debug to a single or subset of the compute units.

Note: This facility should be used only for debugging applications in a controlled environment in which only a single application is running. This extension is only available when using the OpenCL debug library. See section 4.3.1 for details on using the debug library.

5.3.1 New Kernel Built-in Functions

Function	**Description**
`uint get_unit_id()`	Returns the current compute unit identifier. The unit id value returned is in the range of 0 to `CL_DEVICE_MAX_COMPUTE_UNITS`-1.

END OF DOCUMENT