OpenCL Development Kit for Linux on Power

Installation and User's Guide

Version 0.1.1

December 1, 2009

Printed in the United States of America December 1, 2009

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

December 1, 2009

Table of Contents

1 Overview

The OpenCL Development Kit for Linux on Power and 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 Vector/SIMD Multimedia Extensions (Power/VMX) 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.

As a technology preview, the Development Kit has not gone through rigorous product testing and OpenCL conformance validation; however, it has been developed with the objective of being fully conformant in the future. See the Restrictions and Limitations section for specific details on the restrictions and limitations of this release.

1.1 How to read this document

This document is provided as an Acrobat Portable Document (PDF) which can be printed and read. The PDF file contains hypertext links to internal sections and external URLs shown as blue text. These links are best used within Adobe Reader version 7 or greater (http://www.adobe.com/products/acrobat/readermain.html). Text that is part of a system command or sample output is shown in `this font`.

1.2 Tested platforms

OpenCL Development Kit for Linux on Power has been tested on the IBM BladeCenter QS22 running Fedora 9 and the IBM BladeCenter JS23 running Red Hat Enterprise Linux 5.3. The use of this software on any other systems or Linux operating systems is untested and is not guaranteed functional.

1.3 A Word about Licenses

The OpenCL Development Kit for Linux on Power v0.1 and the IBM XL C for OpenCL v0.1 compiler is licensed under two IBM International License Agreement for Early Release of Programs, or ILARs.

1.4 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 that the Development Kit is provided on an "as-is" basis. Wherever possible, workarounds to problems will be provided on the forum.

1.5 Related Documentation

[1] *The OpenCL Specification*, Version: 1.0, Document Revision: 48, Khronos OpenCL Working Group, Editor: Aaftab Munshi , http://www.khronos.org/registry/cl.

[2] *OpenCL API 1.0 Quick Reference Card*, Revision 0409, Copyright © 2009 Khronos Group, http://www.khronos.org/files/opencl-quick-reference-card.pdf.

[3] *OpenCL 1.0 Reference Pages*, Khronos Group, http://www.khronos.org/opencl/sdk/1.0/docs/man/xhtml

[4] *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.

[5] Cell Broadband Engine Resource Center, http://www.ibm.com/developerworks/power/cell.

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 Super Computing site.

libspe2-2.2.80-132.ppc.rpm
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 systems consists of the following package which can be installed with the distro yum install utility.

numactl.ppc.rpm

2.2 RPM Summary

The OpenCL Development Kit consists of 2 components – the OpenCL runtime and the OpenCL kernel compiler. The runtime component is provided in the **opencl-01.1.tar.gz** compressed tar file which contains the following installable packages:

- **OpenCL-0.1.1-0.ppc.rpm** – contains runtime software for CPU devices on QS22 and JS23 systems.
- **OpenCL-cell-0.1.1-0.ppc.rpm** – contains runtime software for SPU Accelerator devices on QS22 systems.
- **OpenCL-devel-0.1.1-0.ppc.rpm** – contains software needed to develop OpenCL applications for CPU devices on QS22 and JS23 systems.
- **OpenCL-cell-devel-0.1.1-0.ppc.rpm** – contains software needed to develop OpenCL applications for SPU Accelerator devices on QS22 systems.
- **OpenCL-samples-0.1.1-0.noarch.rpm** contains sample OpenCL application software. See the OpenCL Samples section for more details on the sample content.

The OpenCL kernel compiler component is provided in the **xlc.opencl.01.1.linux.ppc.iso** image. This ISO image contains the IBM XL C for OpenCL kernel compiler supporting both Power/VMX CPU compute units and CBEA SPU accelerator compute units.

2.3 Development Kit Install / Update Procedure

Installation of the Development Kit differs between the two tested systems.

2.3.1 Installing on a CBEA system (IBM BladeCenter QS22)

Step 1 – Install prerequisite software

This step may be omitted when updating a previous install of the OpenCL Development Kit version 0.1, in that the prerequisite software should already have been installed.

Download the following rpm's from the Barcelona Super Computing site:
http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.1/CellSDK-Open-Fedora/cbea/

libspe2-2.2.80-132.ppc.rpm
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 rpm's.

```
rpm –ivh *.rpm
```

As root, yum install the numactl package.

```
yum install numactl.ppc
```

Step 2 – Install / Update development kit software

The OpenCL Development Kit consists of two files opencl-01.1.tar.gz and xlc.opencl.01.1.linux.ppc.iso. To the install the software, become root and do the following:

Extract the files from opencl-01.1.tar.gz. This must be done in a writeable directory and should not contain other software packages.

```
tar –xzf opencl-01.1.tar.gz
```

The following rpm's should now be available for installation:

OpenCL-0.1.1-0.ppc.rpm
OpenCL-cell-0.1.1-0.ppc.rpm
OpenCL-cell-devel-0.1.1-0.ppc.rpm
OpenCL-devel-0.1.1-0.ppc.rpm
OpenCL-samples-0.1.1-0.noarch.rpm

Install/update the newly extracted rpm's.

```
rpm –Uvh *.rpm
```

Create a mount point and mount the IBM XL C for OpenCL ISO image. This image contains the OpenCL kernel compiler.

```
mkdir –p /tmp/xlc
mount –o loop xlc.opencl.01.1.linux.ppc.iso /tmp/xlc
```

Install/update the compiler software.

```
/tmp/xlc/xlc_install
```

Finally unmount the ISO image and optionally remove any files install files you no longer require.

```
umount /tmp/xlc
```

2.3.2 Installing on a Power/VMX system (IBM BladeCenter JS23)

Step 1 – Install prerequisite software

This step may be omitted when updating a previous install of the OpenCL Development Kit version 0.1, in that the prerequisite software should already have been installed.

As root, yum install the numactl package.

```
yum install numactl.ppc
```

Step 2 – Install / Update development kit software

The OpenCL Development Kit consists of two files opencl-01.1.tar.gz and xlc.opencl.01.1.linux.ppc.iso. To the install the software, become root and do the following:

Extract the files from opencl-01.1.tar.gz. This must be done in a writeable directory and should not contain other software packages.

```
tar –xzf opencl-01.1.tar.gz
```

The following rpm's should now be available for installation:

OpenCL-0.1.1-0.ppc.rpm
OpenCL-cell-0.1.1-0.ppc.rpm
OpenCL-cell-devel-0.1.1-0.ppc.rpm
OpenCL-devel-0.1.1-0.ppc.rpm
OpenCL-samples-0.1.1-0.noarch.rpm

Install/update a subset of the extracted rpm's.

```
rpm –Uvh OpenCL-0.1.1-0.ppc.rpm
rpm –Uvh OpenCL-devel-0.1.1-0.ppc.rpm
rpm -Uvh OpenCL-samples-0.1.1-0.noarch.rpm
```

Create a mount point and mount the IBM XL C for OpenCL ISO image. This image contains the OpenCL kernel compiler.

```
mkdir –p /tmp/xlc
mount –o loop xlc.opencl.01.1.linux.ppc.iso /tmp/xlc
```

Install/update the compiler software.

```
/tmp/xlc/xlc_install
```

Finally unmount the ISO image and optionally remove any files install files you no longer require.

```
umount /tmp/xlc
```

2.4 Development Kit Uninstall Procedure

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

2.4.1 Uninstalling on a CBEA system (IBM BladeCenter QS22)

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-cell-xlc-cmp opencl-cell-xlc-lib
```

As root uninstall the OpenCL runtime, developer, and sample packages:

```
rpm –e OpenCL OpenCL-cell OpenCL-cell-devel OpenCL-devel OpenCL-samples
```

2.4.2 Uninstalling on a Power/VMX system (IBM BladeCenter JS23)

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
```

As root uninstall the OpenCL runtime, developer, and sample packages:

```
rpm –e OpenCL OpenCL-devel OpenCL-samples
```

3 Implementation Details

3.1 OpenCL Devices

The OpenCL Development Kit for Linux on Power provides a full profile Power/VMX CPU device as well as an embedded profile SPU accelerator device. Both device types currently enable 32-bit OpenCL applications.

Both device types support execution of device kernels with up to 3 dimensions and an overall work-group size of 256 work-items. Native kernel execution takes place on the CPU device for both supported OpenCL device types.

The following OpenCL features are supported on both device types:

- Device and native kernel execution
- Compilers are available
- Out-of-order command execution

The following OpenCL features are unsupported on both device types:

- OpenCL image objects
- 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 and accelerator devices coexist and may be simultaneously used. Special care should be taken during simultaneous use as both memory and compute resources are shared. Global memory is a shared between the devices, which means memory consumption on one effects the availability on both. As well, the QS22 CPU is used by both devices during execution.

3.1.1 CPU Device

The JS23 Power6™ processor and the QS22 Power Processor Element (PPE) are both Power/VMX CPU devices. The Power/VMX CPU device is of the CL_DEVICE_TYPE_CPU OpenCL device type. The OpenCL full profile is supported on this device type.

The maximum number of compute units on a Power/VMX CPU device depends on the executing architecture. JS23 Power6 processors have a maximum of 8 compute units whereas the QS22 CPU device has a maximum of 4.

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 are:

- Floating-point denorms are supported
- Floating-point IEEE 754-2008 fused multiply-add is supported

3.1.2 SPU Accelerator Device

The QS22's Synergistic Processor Units (SPU) 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 are:

- Floating-point rounding to zero (rtz) mode is supported.
- Floating-point IEEE754-2008 fused multiply-add is supported.
- 64-bit long and unsigned long data types are supported.
- Infinities and NaNs are not supported except as specified in section 10, list item 6 of the OpenCL specification.
- The native built-in functions (see Table 6.8 of the OpenCL specification) have the same accuracy of their non-native counterparts. However, the native functions support the SPE's single-precision, extended number range. Infinities and NaNs are treated as large numbers and computation overflow result in ±0x1.FFFFEp+127 instead of ∞. Functions in which the specification mandates a NaN result are undefined.

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 extension:

- **Byte Addressable Store** (cl_khr_byte_addressable_store). See OpenCL specification section 9.9 for complete details.

Any requests for specific OpenCL extensions are welcome and should be expressed via this technology's alphaWorks forum.

3.3 Restrictions and Limitations

To release this technology preview in a timely fashion, some functional compromises were made. The following table documents the compromises and subsequent limitations and restrictions of the implementation.

Specification Section	Restriction / Limitation
5.4.3.2	The `-cl-single-precision-constant` and `-cl-denorms-are-zero` compilation options are ignored. It is recommended that all single-precision floating-point constants be declared with the 'f' suffix.

5.6	To use work-group sizes greater than 1, the `reqd_work_group_size` kernel attribute qualifier must be specified. However, if work-group sizes greater than 1 are used in conjunction of no optimization (i.e., `-cl-opt-disable` compile option), then the compilation will fail with severe error (S).
	This can programmatically be accommodated by omitting the required work-group size depending on the compile optimization level. For example:
	`#if __OPTIMIZE__ > 0`
	`  __kernel __attribute__((reqd_work_group_size(16,1,1)))`
	`  void emptyKernel_noarg() { }`
	`#else`
	`  __kernel void emptyKernel_noarg() { }`
	`#endif`
	Increasing the work-group size generally increases performance; however, it may also lead to an increase in the amount of stack space used for __private variables. This increase in stack usage may result in unexpected application termination if the device does not have adequate stack space. These failures can be avoided by reducing the size of the __private variables or by reducing the work-group size of the kernel. Because the CPU device has more stack space available than the SPU accelerator device (the SPU is limited by the 256 KB local storage), programmers may find it helpful to test large work-group sizes on the CPU device. The OpenCL Application programmer may then need to reduce the work-group size when targeting the SPU accelerator device.

5.7	Asynchronous kernel errors result in aborting the application, so no errors can be returned.
6.1.1.1	The half data type is not supported.
6.2.3.2	Explicit conversions with non-default rounding modes (e.g. _rte, _rtz, _rtp, _rtn) are not supported. Use of these converts will result in a compilation error. To map rounded converts to the default convert, specify the following compile option: `-D__MAP_ROUNDING_MODE_CONVERT__`
6.3,i	The ternary selection operator (?:) is not supported for vector expressions. As a workaround, use the `bitselect` built-in.
6.7.2	The optional attribute qualifier, `vec_type_hint`, is not supported. Use of this

	qualifier will result in compilation errors.
6.11.2.1	The OPENCL FP_CONTRACT pragma is ignored. Contractions will occur for optimized kernels.
6.11.7	The `vload_half*` and `vstore_half*` built-ins are not supported.

For additional documentation of known bugs at the time of the release, see the product Readme available on the alphaWorks website.

4 OpenCL Programming

4.1 OpenCL Samples

This release includes the following OpenCL sample application programs.

- **Perlin Noise** – This sample is a Perlin Noise generator, which is used to compute noise images using a time parameter. It is based on Ken Perlin's Improved Noise implementation (http://mrl.nyu.edu/~perlin/paper445.pdf).
- **Julia** – Inspired by Keenan Crane's work, this sample implements a Julia Set ray-tracer.
- **Black Scholes Option Pricing** – This sample demonstrates the computation of European put and call options on non-dividend-paying stocks.

These samples exemplify some of the best practices discussed in section 4.4 Best Practices, and demonstrate various techniques of using the OpenCL framework to leverage multiple compute units. The samples are packaged in the OpenCL-samples RPM, and they each include a readme text document with additional details.

The installed sample source code can be found in /usr/share/doc/OpenCL-0.1-ibm/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 Application

The host application can be built on either a Power System™ or CBEA system, using gcc, ppu-gcc, xlc or ppuxlc. The application will need to include the OpenCL header file (#include <CL/cl.h>), and be linked with the OpenCL library (-lCL) during the build.

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. This utility is installed into /usr/bin as part of the base OpenCL runtime package. It's 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 package.

4.3 Application Debugging

OpenCL applications can encounter the following types of errors:

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

IBM's implementation of OpenCL provides some user controls to help identify the source and resolve these types of problems.

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 two different error conditions can result in the same failing return code. To provide additional guidance, this release includes a debug library which prints further details in some instances. Using it is simply a matter of adding the '/usr/lib/CL/debug' directory to the LD_LIBRARY_PATH environment variable and running the application. For example:

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

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

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

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 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 a debugger such as 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.

Debugging with GDB

Running the OpenCL host application within gdb on a Power System or ppu-gdb on a CBEA system, is helpful for diagnosing system segfaults, setting breakpoints, single stepping code, displaying variables, etc. It is recommended that applications be linked to the debug runtime library prior to using gdb. OpenCL computation kernels built by applications linked to the debug library are automatically compiled with the '-g' flag so that debug information is included.

CPU GDB Debugging

Once gdb has started, it is possible to set breakpoints within the OpenCL computation kernel prior to it being loaded. These are referred to a 'pending' break points. For example, the following sample output is a debug session of the Perlin noise sample (see OpenCL Samples):

```
$ export LD_LIBRARY_PATH=/usr/lib/CL/debug:$LD_LIBRARY_PATH
$ ppu-gdb ./perlin
(gdb) break compute_perlin_noise
Function "compute_perlin_noise" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (compute_perlin_noise) pending.
```

```
(gdb) run –cpu

Starting program: perlin_noise/ppc/perlin –cpu
[Thread debugging using libthread_db enabled]
Device: "CPU Cell Broadband Engine, altivec supported"
[New Thread 0xf7d6f4a0 (LWP 8068)]
[New Thread 0xf736f4a0 (LWP 8069)]
[New Thread 0xf696f4a0 (LWP 8070)]
[New Thread 0xf5f6f4a0 (LWP 8073)]
Loading program source 'perlin_kernel.cl'....
Device max work group size 64
Building kernel...

Saving program binary
'perlin_kernel_CPU_Cell_Broadband_Engine,_altivec_supported_lwgsize1'
Device Kernel Work Group Size 1
2D Work Group Size 1
Global Work Group Size 256 x 1024
Compute Device Data
[Switching to Thread 0xf736f4a0 (LWP 8069)]

Breakpoint 1, compute_perlin_noise (output=0xf4560080, time=0, rowstride=1024)
at kernel.c:201
201     kernel.c: No such file or directory.
(gdb)
```

To perform source level debugging of the kernel, create a soft link named “kernel.c” to your kernel source in the execution directory, then restart gdb, and rerun the application. For example:

```
$ ln -s ../src/perlin_kernel.cl ./kernel.c
$ ppu-gdb ./perlin
(gdb) break compute_perlin_noise
Function "compute_perlin_noise" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (compute_perlin_noise) pending.
(gdb) run –cpu
Starting program: perlin_noise/ppc/perlin –cpu
```

<some output omitted for brevity>

```
Breakpoint 1, compute_perlin_noise (output=0xf4560080, time=0, rowstride=1024) at
kernel.c:201
201       float4 vt = (float4) time;
(gdb)
```

Printing vector types within gdb is also possible. For example, a float4 is displayed as a float array with 4 values. Here's an example where the float4 variable 'vt'.

```
(gdb) print vt
$2 = {0, 0, 0, 0}
```

Often it is easier to debug an OpenCL application when it executes on a single compute unit. 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
```

Accelerator GDB Debugging

Debugging on the SPU accelerator device can be more difficult as the kernel is dynamically loaded into the SPU local store by the runtime. This inhibits the user from setting a kernel breakpoint prior to execution. However, if kernel execution is interrupted by the gdb user (e.g. ctrl-C), or a signal is received, its symbols can be loaded, and debugging can be performed. The following procedure lists the steps.

1. To begin, there must be a saved binary file of the OpenCL kernel. This file can be created using the opencl_build_program utility, or by dumping the program binary data acquired within an application using the clGetProgramInfo() API with the CL_PROGRAM_BINARIES flag. The included samples demonstrate this method.
2. Within gdb, after the kernel has been interrupted (from either the user or a signal), issue the following commands:
   - `thread <spu_thread_id>` (switch to the specified spu thread)
   - `add-symbol-file <kernel_binary_filename> 0x0` (and answer 'y' to the prompt to load the symbols)

Here's an example where the Perlin sample was interrupted during device execution.

```
(gdb) add-symbol-file ./perlin_kernel_ACCELERATOR_PowerXCell8i_processor_lwgsize1
0x0
add symbol table from file
"./perlin_kernel_ACCELERATOR_PowerXCell8i_processor_lwgsize1" at .text_addr = 0x0(y
or n) y

Reading symbols from
perlin_noise/ppc/perlin_kernel_ACCELERATOR_PowerXCell8i_processor_lwgsize1...done.
(gdb) bt
#0  0x00000af4 in grad (hash={164, 164, 164, 164}, x=
    {-0.75, -0.6875, -0.625, -0.5625}, y=
    {-0.799999952, -0.799999952, -0.799999952, -0.799999952}, z=
    {-0.550000191, -0.550000191, -0.550000191, -0.550000191}) at kernel.c:91
#1  0x00000fc0 in noise3 (x={0.25, 0.3125, 0.375, 0.4375}, y=
    {0.200000003, 0.200000003, 0.200000003, 0.200000003}, z=
    {0.449999809, 0.449999809, 0.449999809, 0.449999809}) at kernel.c:138
#2  0x000011cc in compute_perlin_noise (output=0x20080, time=0.100000001,
    rowstride=1024) at kernel.c:228
#3  0x000015fc in __compute_perlin_noise_exec (currentWorkGroup=98304,
    workGroupCount=2853, args=0x3ad00) at kernel.c:334
#4  0x0003ce0c in executeKernelCommand ()
    from CLRuntimeAccelCellSPU@0xfae0c80 <19>
#5  0x0003d988 in main () from CLRuntimeAccelCellSPU@0xfae0c80 <19>
#6  0x0003b08c in _start () from CLRuntimeAccelCellSPU@0xfae0c80 <19>
#7  <cross-architecture call>
#8  0xf7f16ef8 in syscall () from /lib/libc.so.6
#9  0xf7de018c in _base_spe_context_run () from /usr/lib/libspe2.so.2
#10 0xf7dd5204 in spe_context_run () from /usr/lib/libspe2.so.2
#11 0x0facb8bc in ibm::openclDeviceCellSPU::CellSPUUnit::threadFunc ()
    from /usr/lib/CL/debug/CL/device/libCLDevAccelCellSPU.so
#12 0x0fc66e3c in start_thread () from /lib/libpthread.so.0
#13 0xf7f1b670 in clone () from /lib/libc.so.6
(gdb)
```

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.

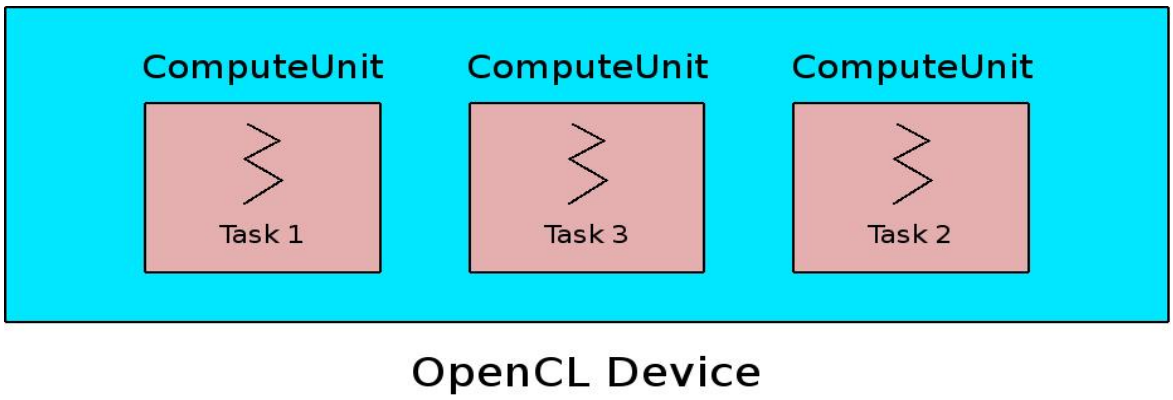
The section Implementation Details documents attributes of the supported hardware devices. It is important that the OpenCL programmer structures their application to map to the strengths of the system it will be deployed on.

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 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 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. Task are enqueued to a command queue through the OpenCL clEnqueueTask() API call.

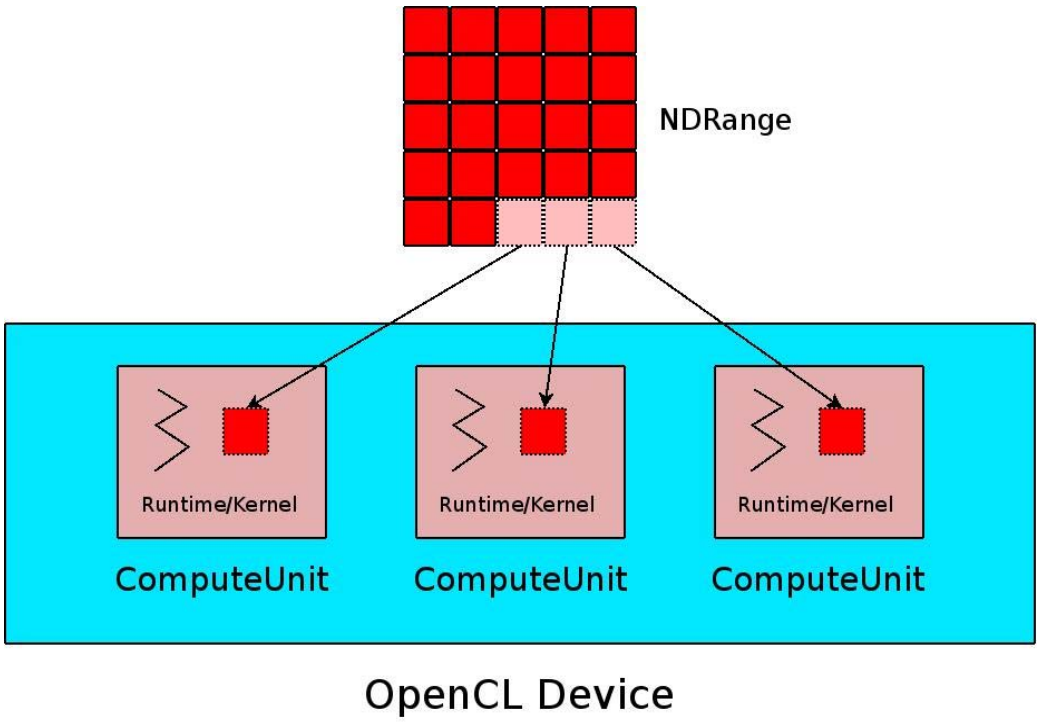


Tasks shift the burden of scheduling work off of the OpenCL runtime and onto the application's kernels. Tasks should be used when the OpenCL programmer is comfortable explicitly managing the flow of

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

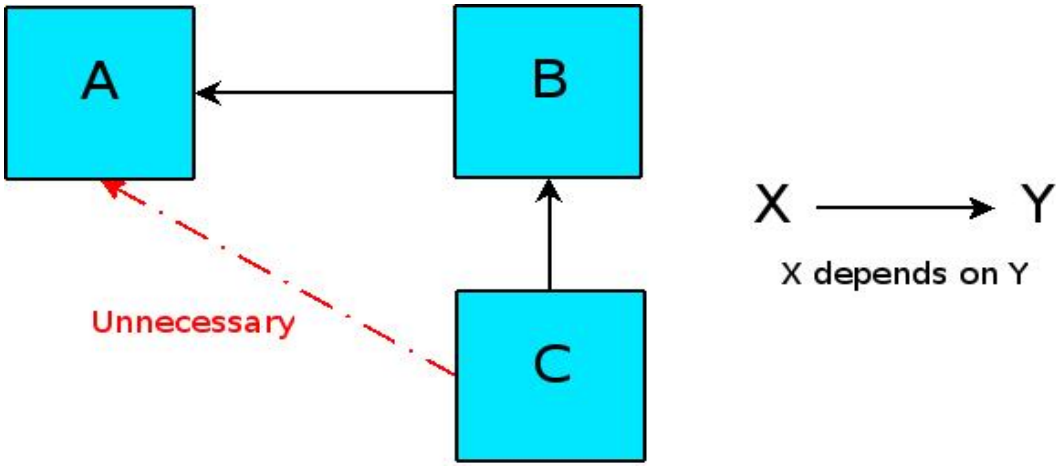
The OpenCL Development Kit for Linux on Power uses a memory model where the application, OpenCL accelerator devices, and OpenCL CPU device share a common memory bus. On systems, where host and device memory are on separate buses, there is a performance penalty incurred when sharing a memory object between 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(), clEnqueueWriteBuffer() or clEnqueueNativeKernel() to examine or modify memory buffers. Both clEnqueueReadBuffer() and clEnqueueWriteBuffer() 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 clEnqueueReadBuffer() and clEnqueueWriteBuffer().

IBM's OpenCL runtime implementation currently supports only 32-bit applications. This means that the total address space for the OpenCL application and runtime is limited to 4GB. The application should be careful to manage its own 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.



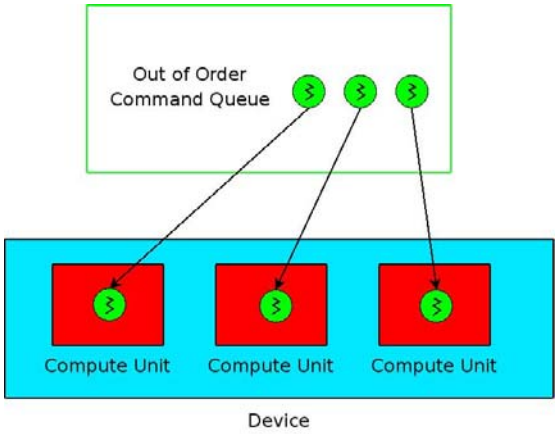
The application can reduce the amount of work required by the runtime to process event dependencies by eliminating redundant dependencies. For example, given three events A, B and C where B depends

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().

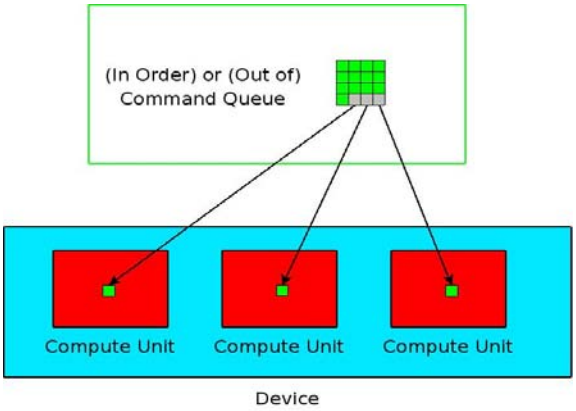
4.4.4 Command Queues

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.

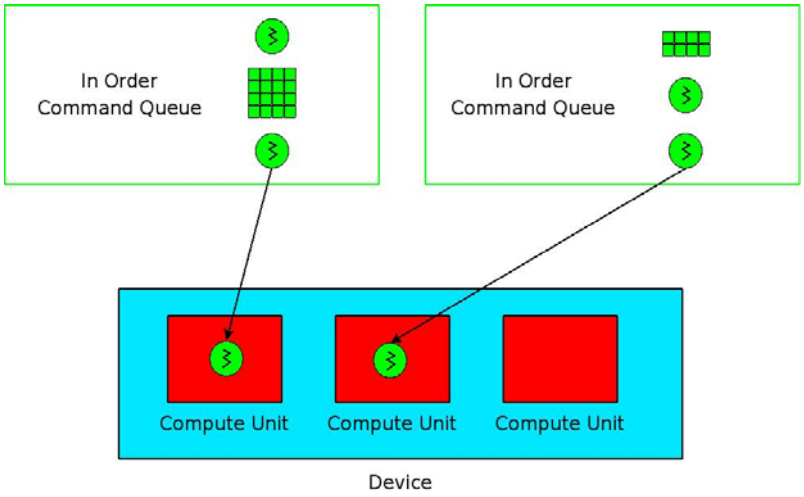
Out-of-Order Command Queue



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 is compiled with the reqd_work_group_size attribute. If the attribute is not set, only one work item per work-group is supported. 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 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 clCreatePogramWithSource() to reduce the run-time. The opencl_build_probram utility is also provided to aid in creating a program binary. See Compiling OpenCL Kernels for additional information.

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 JS23. 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 not implemented in the CPU device.

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 comprises of all SPUs available on the system.

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 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, OpenCL program's text, __local variables 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 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 ~80KB 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. async_work_group_copy() 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.

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 like:

- Floating-point conditionals may be transformed such that strict compare ordering in the presense of NaNs may not be preserved.
- Floating point divides may be transformed into a reciprocal-multiply.
- Software support of infinites and NaNs is omitted for half_divide and half_recip built-ins.

END OF DOCUMENT