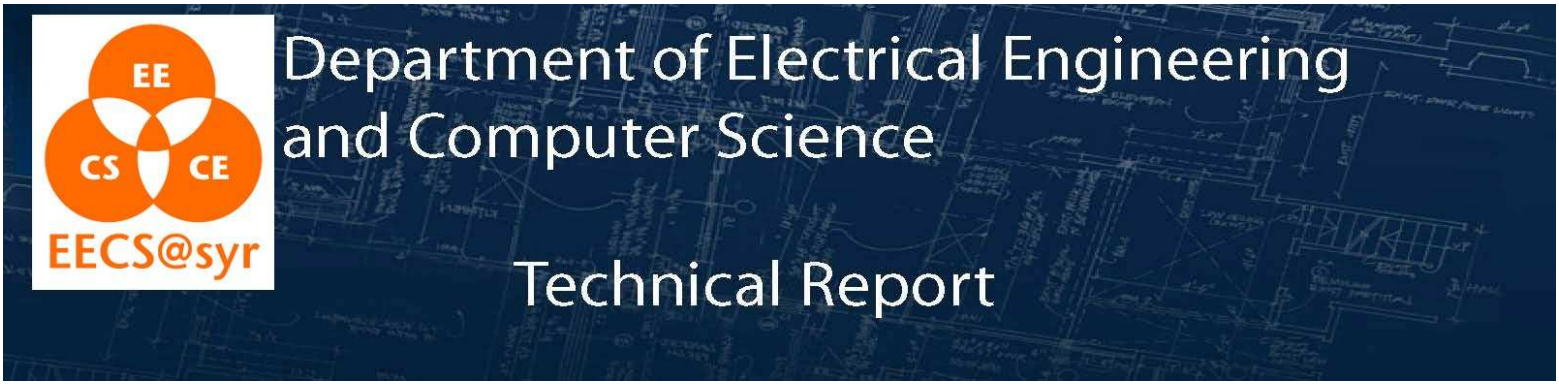
9-10-2009

UniverCL 1.0 --- Phase I of a complete OpenCL implementation

Phil Pratt-Szeliga

Jim Fawcett
Syracuse University, jfawcett@twcny.rr.com



SYR-EECS-2009-02 Sept. 10, 2009

UniverCL 1.0 – Phase I of a complete OpenCL implementation

Phil Pratt-Szeliga pcpratts@syr.edu
Dr. Jim Fawcett jfawcett@twcny.rr.com

ABSTRACT: Recently the Khronos group has released an open specification for OpenCL, the Open Computation Language. OpenCL strives to create a standard language for programming novel computer architectures such as the multi-core CPU, the GPU and accelerators. The programming model is similar to Nvidia's CUDA. UniverCL is a partial implementation of the OpenCL 1.0 Specification that has enough functionality to demonstrate the major components of the specification. It currently includes support for two hardware devices: the multi-core CPU and the Cell Broadband Engine. This technical report captures the work done in Phase I of UniverCL's development and outlines Phase II of development.

KEYWORDS: OpenCL, Parallel Programming, Multi-Core CPU, Cell Broadband Engine

Syracuse University - Department of EECS,
4-206 CST, Syracuse, NY 13244
(P) 315.443.2652 (F) 315.443.2583
http://eecs.syr.edu

Technical Report SYR-EECS-2009-02

UniverCL 1.0 – Phase I of a complete OpenCL implementation

Phil Pratt-Szeliga – pcpratts@syr.edu

Adviser:
Dr. Jim Fawcett - jfawcett@twcny.rr.com

Department of Electrical Engineering and Computer Science

Syracuse University

Version 1.4
09/10/2009

Table of Contents

1. Introduction

Recently the Khronos group has released a specification for OpenCL, the Open Computation Language. OpenCL strives to create a standard language for programming novel computer architectures such as the multi-core CPU, the GPU and accelerators. The programming model is similar to Nvidia's CUDA. UniverCL is a partial implementation of the OpenCL 1.0 Specification that has enough functionality to demonstrate the major components of the specification. It currently includes support for two hardware devices: the multi-core CPU and the Cell Broadband Engine. This technical report gives an overview of OpenCL and captures the work done in Phase I of UniverCL's development. Phase I was supported through the Google Summer of Code Program.

The OpenCL Specification describes a set of APIs and an OpenCL C compiler. Phase I of UniverCL supports a minimal set of functions in the APIs. An example was created that performs the Deluanay Triangulation computation that serves as a stress test and a developers example. Phase I does not support the OpenCL C compiler, programs that run on the target computing devices need to be written in the native target device language.

1.1 Summary of Contributions

Phil Pratt-Szeliga created the implementation of the minimal set of functions needed to run a real-world OpenCL program on the CPU and Cell Broadband Engine. The OpenCL functions that are supported fall into the following categories: context functions, command queue functions, memory object functions, program functions and kernel functions. The complete listing of every function supported is provided in Appendix A. To summarize, everything needed for launching a kernel is supported. In order to implement these functions the design of the internals of OpenCL and a device driver architecture where created. The OpenCL internals and device driver architecture are posed to support the GPU, but a driver has not been created yet. Phil Pratt-Szeliga also wrote this report. Dr. Jim Fawcett advised Phil Pratt-Szeliga and edited this report.

1.2 Report Structure

The remainder of this report is structured into eight parts. These parts are OpenCL Overview, UniverCL Internal Architecture, Differences from OpenCL Specification 1.0, Examples, Summary of Work Done, Acknowledgments, Future Work, References and Appendices.

2. OpenCL Overview

This section is devoted to familiarizing the reader with OpenCL, without requiring him or her to read and understand the entire 304 page specification. It is broken into two sections, motivation and tutorial

2.1 Motivation

Currently, a researcher trying to improve the performance of an application using hardware acceleration has many options. For instance, there are multi-core CPUs, GPUs and other accelerators such as the Cell Broadband Engine. Each of these examples has a different programming model that the researcher must

painstakingly learn. Programming multi-core CPUs requires creating threads or processes. Programming GPUs requires learning proprietary languages, with each vendor supporting their own language. ATI has the Stream language and Nvidia has the CUDA language. Lastly, programming the Cell Broadband Engine requires knowledge of its own language. In addition to learning the programming methodologies of the specific device, a researcher must be well acquainted with the underlying hardware architecture to realize improved performance over the baseline. If the researcher does not learn the specifics of the underlying hardware architecture, he or she can actually get worse performance than a standard serial version of their algorithm. See [5] for an example of a developer porting a serial application to a GPU and achieving 60x worse performance, then optimizing the implementation to only arrive at 9x worse performance.

The purpose of OpenCL is to create one language for all of these hardware acceleration devices and a common API to control the execution of the programs written in OpenCL C. This will better enable researchers to answer questions like: "What is the fastest hardware acceleration device for my problem?" and further "Can we create predictive schedulers that can choose the right device for my problem?"

2.2 Tutorial

The OpenCL API defines a set of functions to manage contexts, command queues, memory objects, programs, kernels and events. It also defines a set of functions to query device information. Below are a set of definitions which elaborate the OpenCL architecture.

A Host is the processor running the operating system that can control the devices. The CPU is the only type of host.

A Device is a component controlled by the host. Example of devices are the CPU, GPU or accelerator devices such as the Cell Broadband Engine.

Contexts represent an environment that kernels can execute in. They also represent the domain in which synchronization and memory management is defined[1].

Command Queues allow commands related to memory management, synchronization and kernel execution to be enqueued on the host side using the OpenCL API and dequeued internally in the OpenCL runtime. Once the commands are dequeued they are processed by a combination of host and device resources.

Memory Objects are an abstraction that allows one interface for data storage on all supported devices. Reads and writes to and from memory objects are enqueued on a command queue (on the host side) or triggered by the function async_work_group_copy (on the device side)

Programs in OpenCL are a group of compiled regular functions and kernel functions. A program groups kernel functions with regular functions, each of which can be executed on the device.

Kernels are special functions in a program that can be enqueued in the command queue on the host side and executed on the device. They are entry points into an OpenCL program. This usage of kernel is distinct from an operating system kernel. The usage of kernel that we are using derives from the usage of the word kernel in image processing. In image processing a kernel is a small matrix of pixels, that is used as an operator during image convolution [2]. In this sense, OpenCL Kernels can be described as operators that transform data, in parallel.

The parallel programming paradigm for OpenCL is very similar to CUDA by Nvidia. Kernels are enqueued to command queues and then executed in parallel. Kernels have multiple dimensions each with its own range. One task (a copy of the kernel function) is executed for each range element in each dimension. A unique indexer is assigned to each task allowing the task to index into a memory object and obtain input data unique to itself. A pictorial representation of this process is show in figure 1.

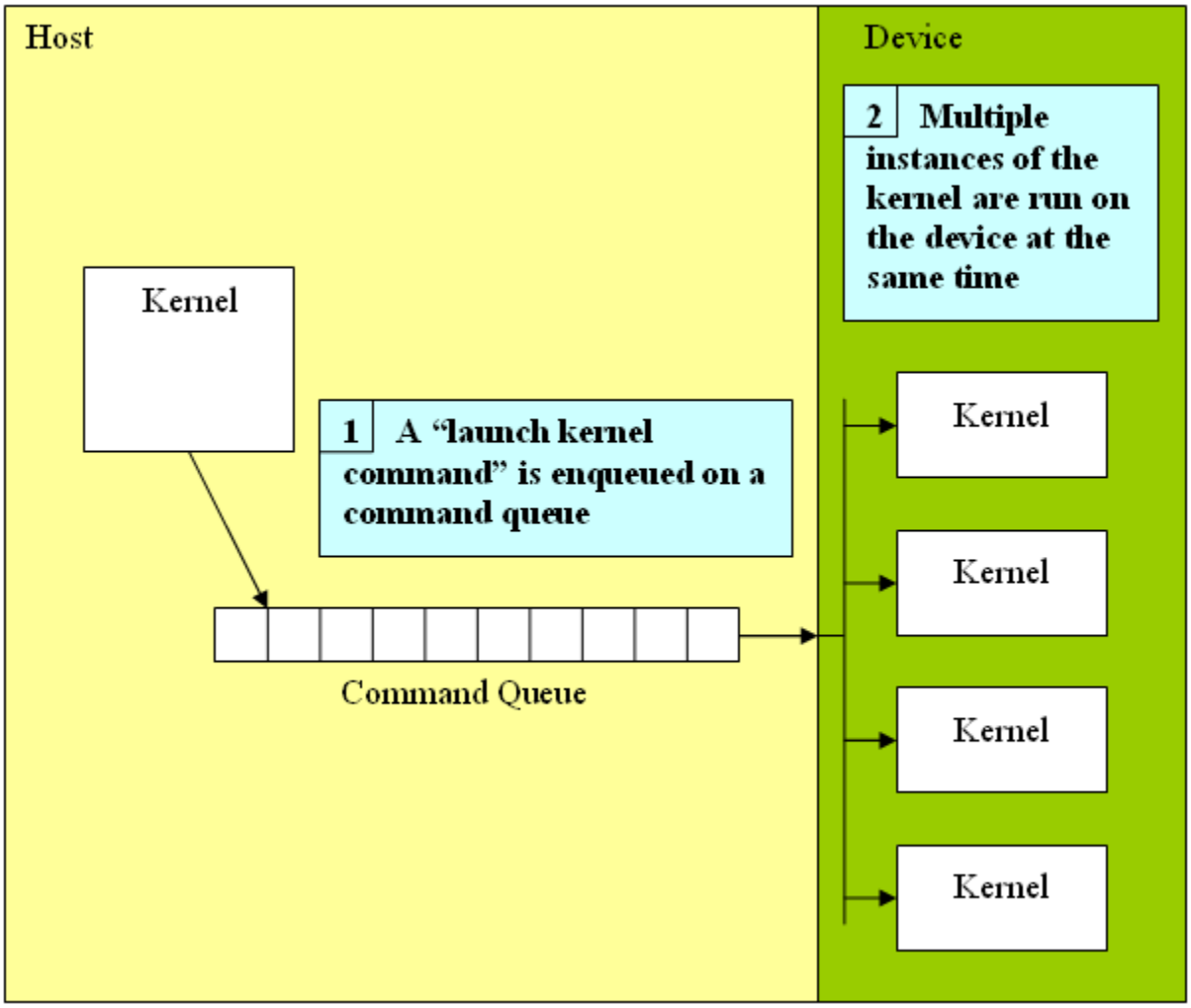


Figure 1 – Kernel Launch Diagram

An more thorough explanation of OpenCL can be found in the OpenCL Specification [1]. A copy of the specification is found in the docs folder or at: http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf

3. UniverCL Internal Architecture

The architecture is separated into API functions, clInternal functions and device driver functions. The API functions are any function listed in sections 2.1 and 2.2. They usually extensively check input arguments and then will perform some action by calling a clInternal* function or device driver function.

To give an idea of how the major parts of the system work, I will explain what happens from compile time to querying devices to creating runtime objects (context, command queue, memory object, program and kernel) to running an NDRange Kernel to making a call back from the kernel to the runtime to finishing execution.

3.1 Compiling and Initializing

To get UniverCL you can use git, an open source distributed version control system. From a command prompt type:

#git clone git://github.com/pcpratts/gcc_opencl.git

If the command git is not found you need to download git.

To compile and run UniverCL you need to have the following dependencies:

1. gcc 4.3 or greater
2. libxml2
3. libglib2.0
4. libffi5 (CPU target only)
5. libjit0 (CPU target only)
6. lshw
7. libspe2 (Cell target only)
8. spu-gcc (Cell target only)

UniverCL currently supports the CPU and Cell devices. When you compile UniverCL (by typing make in the root folder of the project) it automatically detects if your system supports the Cell device. Currently, when you are querying the devices available the OpenCL runtime will return the type CL_DEVICE_TYPE_CPU if you are running on a regular CPU and only the type CL_DEVICE_TYPE_ACCELERATOR if you are running on the Cell. In the future, modifications will be made so that, when running on the Cell, the OpenCL runtime will return two types, namely CL_DEVICE_TYPE_CPU and CL_DEVICE_TYPE_ACCELERATOR.

Once you have compiled UniverCL there are two more steps before successfully running an example.

The first step is creating the lshw_output.xml file and the second step is modifying the opencl_config.xml file. In the future this will be automated.

To create the lshw_output.xml file execute the following command as the root user

#lshw -xml > lshw_output.xml

To modify the opencl_config.xml file, open it in a text editor. It looks like this:

<opencl_config>

<device><name>cpu_generic</name><is_present>true</is_present><is_default>true</is_default></device>

<device><name>accel_cell</name><is_present>false</is_present><is_default>false</is_default></device>

<device><name>gpu_nvidia</name><is_present>false</is_present><is_default>false</is_default></device>

</opencl_config>

If you want to use the cpu, make is_present for the cpu_generic entry true. Similarly, if you want to use the Cell, make is_present for the accel_cell entry true. The gpu_nvidia entry must always be false right now, but it does not necessarily have to be present. Also, right now you can only have one type running at a time, so don't make both cpu_generic and accell_cell true.

3.2 Querying Devices

When you first query to see what the devices are available on the system (using clGetDeviceIDs found in the cl/cl_device_api.c file), the device drivers are initialized for the Cell and the CPU using an object factory (clInternalDeviceFactory found in cl/devices/factory.c). The object factory dynamically loads the module as a shared library and returns an instance of type clInternalDevice_s *. clGetDeviceIDs then initializes the device info portion of the driver. Each driver has a deviceinfo.c file, a driver.c file and a runtime.c file. The CPU deviceinfo.c file parses the lshw xml output while the Cell deviceinfo,c file bases all of its information on the Playstation 3 right now. The information extracted from lshw is listed in Appendix B. The last thing that is done when a device is being initialized is the creation of a thread for that device. Each device has its own thread that pulls jobs from command queues. The threading architecture will be described in section 3.4.

3.3 Creating Runtime Objects

Runtime objects are contexts, command queues, memory objects, programs and kernels. I will cover how each of these object is handled internally.

When creating a context with clCreateContext (found in cl/cl_context_api.c) the function verifies input arguments and then creates a context object. Each context object internally has a hashtable for all of the

command queues, memory objects, programs, kernels and events it will keep track of. When checking to see if an object from the client code is valid the corresponding hash table is checked for an identifier (this happens in cl/cl_internal.c). If the identifier is found, it is valid. Each context also has a mutex associated with it. So for design simplicity, each operation that must be thread safe locks it's context's mutex. This was chosen over the complexity associated with having a mutex for each element that must be thread safe. Lastly, the function clInternalAddContext is called to add the id of the context to the static hashtable that keeps track of the contexts in cl/cl_internal.c

When creating a command queue a thread is created for it (the function CommandQueueThreadProc is run in cl/cl_internal.c). There is one thread per command queue. This thread finds the best event to run based on an in-order or out-of-order scheduler. The best event to run with the out-of-order scheduler is the first event in the queue where all of the events that it is waiting for are complete. Once an event is selected it is removed from the command queue and the command queue thread waits until the device thread says it is ready for another job.

When creating memory objects the API function clCreateBuffer (found in cl/cl_mm_obj_api.c) calls a function in the device driver (CreateBufferOnDevice) to create a buffer on a device. The OpenCL abstraction is that one memory object can hold pointers to buffers on multiple devices. The CPU CreateBufferOnDevice just calls malloc to create a buffer which the Cell CreateBufferOnDevice calls memalign to get a buffer that is aligned on a 16 byte boundry (this is required in the Cell when doing DMA transfers).

When creating programs the API function clCreateProgramWithBinary (found in cl/cl_program_obj_api.c) calls a function in the device driver (OpenBinary). The CPU OpenBinary function memory maps the input binary to a file (/tmp/opencl.bin.#1.#2, where #1 and #2 are two unique identifiers) and then opens this file with g_module_open (part of libglib2.0). If it is successful then the binary is deemed good. The Cell OpenBinary function also memory maps the input binary to a file (/tmp/opencl.bin.#1.#2). The memory mapped file is then opened with spe_image_open (part of libspe2). If that succeeds then the binary is deemed good.

When creating kernels the API function clCreateKernel (found in cl/cl_kernel_obj_api.c) calls a function in the device driver (OpenKernel). The CPU OpenKernel function tries to find the function requested in the binary stored in the program using g_module_symbol (part of libglib2.0). If it can find it the function is a success, otherwise it returns failure. The Cell OpenKernel function returns failure unless the function requested is “cell_function”. This is because there is no dynamic function loading on the Cell SPU. I looked into doing the dynamically loading myself but I ran into troubles linking to dependencies. This problem will be easily resolved when the OpenCL C compiler is created in Phase II.

3.4 Running an NDRange Kernel

UniverCL supports enqueing the following commands on a command queue: write to a buffer, read from a buffer, task (an NDRange Kernel with just one work item), native kernel (CPU only) and NDRange Kernel. I will describe the NDRange Kernel because it is the most involved and the others are very similar, just less complicated.

When an NDRange Kernel event is enqueued on a command queue, it goes into an actual queue. Each event has a list of events it must wait for before it can run (this list can be empty). Once the command queue thread schedules the event (determines that it is ready to run) it waits until the device thread signals that it is ready for another event. The first command queue thread that gets awakened from the semaphore wait (remember there are multiple command queues, each which their own threads) gets to send its event to the device thread. The device thread then sends the event to the device driver which also has threads. For an independent core device such as the CPU and Cell there is one software thread per core. For a device without independent cores such as the GPU, there is only one software thread. Right now the GPU implementation has not even been started, but the architecture is there to support it.

In the Cell and CPU the events are run one at a time on the core threads. The NDRange Kernel event is a special event where the work items are split up on the cores. For example, if there are 32 work items and 4 CPU cores, 4 work items will run at a time, until all work items complete, with an exception. If a barrier [3] is encountered, 28 threads will temporarily be created so that each work item can get to the barrier. The 28 threads will die after some elapsed time (60 seconds currently in the code) if there is not a need for them. To actually call the function with variable arguments on the CPU, libffi [4] is used. Libffi allows a programmer to call a function that, at compile time, does not have a definition of the number and types of it's parameters. Libffi does not currently have a Cell SPU target so the function called on the Cell must have one argument and be named cell_function.

Once the last thread in the group of 32 is done (using a clInternalIsLast function found with the rest of the other synchronization functions in cl/sync/sync.c) a condition variable is signaled to let the outside world know that the event is done.

3.5 Making a callback from the kernel

To support the simulated OpenCL C intrinsics (get_work_dim, for instance) without a compiler the following problem had to be overcome: We have to get data into a kernel. There are two similar solutions, one for the CPU and another for the Cell.

On the CPU I place the cl_event data of the event in a hashtable before I call the kernel function. The key to the hashtable is the current stack address. When I want to retrieve the data from the hashtable I use libjit to walk up the stack. At each frame in the stack I check to see if the stack address is in the hashtable. If it is, I return the value in the hashtable. See cl/devices/cpu/generic/runtime.c for this, specifically the function LaunchKernel, FindEvent, FindKernelInfo and ShutdownKernel.

On the Cell I also place the kernel_info data in a data structure called the control block (in cell programming speak it is called the control block) that is sent to the SPU when it is starting up, before the cell_function is called. I also place a unique identifier in the control block. The unique identifier is just a number incremented by one each time that is allowed to rollover. This method for unique identifier generation is used throughout UniverCL. The unique identifier in the control block is used to get the ndrange kernel's barrier structure when a runtime barrier is called.

4. Differences from OpenCL Specification 1.0

In order to make the system work without the OpenCL C compiler, two functions were renamed and one was added. As discussed in Appendix A.3: async_work_group_copy (local to global version) is named async_work_group_copy_local_to_global and async_work_group_copy (global to local memoryversion) is named async_work_group_copy_global_to_local.

The other difference is that the function clGetMemPtr was added. This function returns a void * pointing to the internal memory buffer of a cl_mem object for a certain device. This is useful when running an NDRange Kernel and passing a buffer pointer to it. Eventually when the compiler is done this function will not be needed. The prototype of this function is: void * clGetMemPtr(cl_mem memobj, cl_device_id device, cl_int * errorcode_ret);

5. Examples

I have included a very small set of examples of how to compile and run a kernel for the CPU and the Cell. These example kernels and host functions are located in examples/helloworld/. A large example is located in examples/delaunay/. Due to technical constraints that will be solved when the OpenCL C compiler is complete, the Cell Kernel function must always be named 'cell_function' and accept only one argument of type unsigned long long. The Cpu Kernel can have any name with any number of arguments of any size (within reasonable limits).

6. Summary of Work Done

While sponsored by Google, Inc. I created the functions listed in Appendix A and the design and implementation of the internal architecture to support all of their operations. I created a device driver framework and implemented the devices drivers for the CPU and Cell processor. This allows for an OpenCL kernel to be executed in parallel on either a CPU or a Cell processor with basic memory transfer and synchrnoization support. The device driver framework is structured so that it will be easy to add support for GPUs. I also created a stress test and example program (Delaunay Triangulation Computation) for the CPU and Cell. While debugging the example program I fixed many race conditions in UniverCL and now the valgrind concurrency checker (helgrind) reports no possible concurrency problems. Valgrind is a tool that is used primarily to check the correctness of programs by running them in an instrumented environment.

7. Acknowledgements

I would like to thank Google, Inc. and the Google Summer of Code Program for sponsoring this work, Paolo Bonzini for mentoring me and Dr. Jim Fawcett for mentoring me. Also I would like to thank everyone in the sponsoring organization, GCC. Lastly, I would like to thank Allison Figus for coming up with the name: UniverCL and implementing a function to create multibase numbers. Multibase numbers are numbers where each digit has a different base. This was used when assigning identifiers to kernel tasks.

8. Future Work

To fulfill my master's thesis I plan to complete Phase II of UniverCL. Phase II is an OpenCL C compiler that supports the CPU and Cell devices. I plan to create a minimal implementation that does not include every aspect of the OpenCL C standard. I plan to implement the core language

needed to compile the Delaunay triangulation example. This includes support for kernel functions, kernel function argument qualifiers, asyncronous memory transfers, event waiting and barriers, I also plan to support the following features not required to run my Delaunay Triangulation Example: vector data types, the restrictions found in section 6.8 (parts a, c, d, e, f, g, h, I, j, n and o) of the OpenCL specification [1].

9. References

1. OpenCL Specification 1.0 – Khronos OpenCL Working Group. Found in docs/opencl-1.0.33.pdf
2. Kernel (Image Processing) Definition: http://www.imaging-toolkit.com/kernel%20(image%20processing).htm
3. Barrier: http://en.wikipedia.org/wiki/Barrier_(computer_science)
4. libffi: http://sourceware.org/libffi/
5. Infernal-GPU: CUDA-Accelerated RNA Alignment, Adam Bazinet, December 2008, http://serine.umiacs.umd.edu/files/infernal-gpu_writeup.pdf

Appendix A.

A.1 Platform API

1. clGetPlatformInfo - Get info about OpenCL
2. clGetDeviceIDs - Get what devices are supported on system
3. clGetDeviceInfo - Get info about a specific device

A.2 Runtime API

1. clCreateContext - Create an OpenCL context
2. clReleaseContext - Release an OpenCL context
3. clRetainContext - Retain an OpenCL context
4. clCreateCommandQueue - Create a command-queue on a specific device
5. clReleaseCommandQueue - Release a command-queue
6. clRetainCommandQueue - Retain a command-queue
7. clCreateBuffer - Create a buffer object
8. clReleaseMemObject – Release a memory object (a buffer object is a memory object)
9. clRetainMemObject – Retain a memory object
10. clEnqueueReadBuffer - Enqueue a read
11. clEnqueueWriteBuffer - Enqueue a write
12. clCreateProgramWithBinary - Create a program object from a pre-compiled binary.
13. clReleaseProgram - Release a program object
14. clRetainProgram – Retain a program object
15. clCreateKernel - Create a kernel object

16. clReleaseKernel - Release a kernel object
17. clRetainKernel – Retain a kernel object
18. clSetKernelArg - Set the kernel arguments
19. clEnqueueNDRangeKernel - Enqueue a command to execute a kernel on a device
20. clEnqueueTask - Enqueue a single work item
21. clWaitForEvents - Wait for events to complete
22. clReleaseEvent - Release an event
23. clRetainEvent – Retain an event

A.3 Simulated OpenCL C intrinsics

1. get_work_dim – Get the number of dimensions for this job
2. get_global_size – Get the global size of a certain dimension
3. get_global_id – Get the global id of a certain dimension
4. get_local_size – Get the local size of a certain dimension
5. get_local_id – Get the local id of a certain dimension
6. get_num_groups – Get the number of groups of a certain dimension
7. get_group_id – Get the group id of a certain dimension
8. barrier – Wait until all work items have gotten to this barrier, and then continue execution
9. mem_fence – Wait until reads and writes proceding mem_fence are complete
10. read_mem_fence – Wait until reads proceding read_mem_fence are complete
11. write_mem_fence – Wait until writes proceding write_mem_fence are complete
12. async_work_group_copy (local to global memory) – Copy bytes from local to global memory (technical constraints forced me to name this async_work_group_copy_local_to_global)
13. async_work_group_copy (global to local memory) - Copy bytes from local to global memory (technical constraints forced me to name this async_work_group_copy_global_to_local)
14. wait_group_events – Wait for a group of copy events to finish

Appendix B

In this appendix the data extracted from the lshw xml output is summarized for the CPU and the Cell.

B.1. CPU Device Info

The CPU Device Info module extracts the following information from the lshw xml output.

1. Vendor ID
2. Max Compute Units
3. Max Clock Frequency
4. Address Bits
5. Global Memory Cache Line Size
6. Global Memory Size
7. Local Memory Size
8. Max Memory Allocation Size