A Generic OpenCL Implementation to Support Novel Architectures

Hau Tat Luk

Institute for System Level Integration

Student ID: 091613958

13th May, 2013

Contents

Abstract

OpenCL is an standard for computation on general purpose parallel computation device. It facilitates standardisation of the way in which parallel computation can be performed on different processing architectures. The standard is open, and is widely adopted by the industry, supported by major commercially available GPU and CPU devices, from vendors such as AMD and Intel.

This project aims to produce a generic, open implementation of OpenCL, with a host runtime and a device application. Such implementation would be useful in the research and development of novel parallel processor architectures, such as MORA.

This project starts with an rudimentary OpenCL runtime and device simulator. Over the course of the project the starting materials were transformed into a generic OpenCL implementation for use with any novel architectures. Advances were made to the runtime by refining existing functionalities and adding support for additional OpenCL features, Host-device communication was redesigned, and a scheduler was added to the device simulator to facilitate parallel processing of work items received from the host. Together the system can be used to simulate any novel parallel processing architectures, either on the same machine or across two machines where the device application runs on architectures more comparable to the architecture researched. It is envisaged that with minimal work the device application could be used to distribute work items onto an actual implementation of the researched architecture.

Introduction

As traditional methods of obtaining performance gains are yielding ever dimishing speed ups in uniprocessors, the move to parallel computation have become increasingly important. Many computational problems such as media processing, graphical acceleration and cryptography are inherently parallel. This applications could benefit from programming paradigm that allow parallelism to be explicitly defined. Parallel processing architectures have emerged to address the need for parallel processing computation platforms.

This project aims to provide a standard framework that facilitate execution of parallel computation on novel architectures. In particular, this project is aimed at Asymmetric coarse grain array of processors such as Media-Oriented Reconfigurable Array (MORA)[8] and Asynchronous Scalable Array of Processors [2].

In March 2012, Marcin Bujar published his work[3] on an OpenCL framework with the ultimate goal to allow OpenCL applications to be computed on a architectures designed for massively parallel computation. This project has been working towards extending his work to be used for general parallel computation on arbitrary architectures.

In the paper, *A C++-embedded Domain-Specific Language for Programming the MORA Soft Processor Array*[7], a high-level domain specific language for the MORA soft processor array called MORA-C++ is described. MORA-C++ is valid C++ which can be compiled to run on architectures other than MORA. When minimal modification to the kernel build procedure, it is possible that this framework will be able to directly compile MORA-C++ to MORA binary, while providing an application with an OpenCL API.

Report content

This report first lays down the context of the work done in this project by discussing the notion of parallel computation and different types of parallel processor architectures. It will then introduces the starting point of the work and the objects that must be achieved. A brief chapter on non-implementation related decisions is included. In the Design and Implementation session, design decisions and their rationle, and details of the actual implementation is explained. Finally, an evaluation chapter documents a comparison between this implementation and Intel's own OpenCL implementation, using an Intel example program. This section also discusses possible future improvements.

Background

This chapter provides and discuss the context in which this project was conceived, and allow the reader to better understand the aim of this project. The concept of Parallel computation is discussed, and the two main classifications of multiprocessor configuration, Homogenous and Heterogenous multiprocessors, are described.

Parallel computation

Parallel computing is a mode of computation in which more that one part of a task is computed at the same time. [1] This mode of computation exploits parallelism, implicitly or explicitly, that exists in a task at various level in order to speed up computation.

Traditionally many programs have be designed for sequential computation, running on a single uniprocessor. In his 1965 publication, *Cramming more components onto integrated circuits[4]*, Gordon E. Moore described his observation of the trend in the hardware capability of computers, which predicts that the processing power and memory of typical computer will double every 18 months. Advances in fabrication technology and design techniques in the past four decades have allowed the capability of processors in terms of speed, power, and memory resources, to closely follow the prediction of Moore's law. The constant shinking of silicon feature size of yielded steady performance gain. As feature size decreases, transistor power consumption falls, propagation time decreases and operating frequency increases. However, silicon fabrication technology is fast reaching its limit. At the time of writing, most high-end GPUs such as AMD's 79xx series, are built on 28nm technology, and high-end CPUs such as Intel's i7 are built on 22nm technology. As feature size decreases, quantum effects in the manufacturing process play an increasing role. Transistor characteristics become less reliable and the overall yield of the manufacuring process decreases. For these reasons, we can no longer be reliant on shinking feature sizes for future performance gain.

Initial application of design techniques such as superscalar processing and pipelining have yielded considable performance gain, but repeated application of these techniques suffer from Amdahl's law of diminishing return. In order to maintain future performance gain, a move to parallel computation seems inevitable.

Superscalar processing and pipelining both exploit instruction level parallelism that exists in a sequential program. These techniques cannot exploit thread level parallelism, which is better utilised by multiprocessors. The programmer needs to explicitly express this type of parallelism.

Homogeneous and Heterogenous computing

Homogeneous computing platforms consists of computing resources of the same architecture. A Homogenous system could have one of more processors, and all processors will be of the same design and have the same resources and capabilities.

Heterogenous computing platforms, in contrast, are made up of computation resourses of different architectures and capabilities.

Homogeneous and Heterogenous computing platforms each serve a different type of parallel processing.

Heterogeneous platforms typically have different types of processor architiure for different aspect of an application. An example of a heterogeneous architecture is the Open Media Applications Platform(OMAP) from Texas Instruments. The OMAP typically features a general purpose ARM processor and a DSP. In this case, the ARM processor is usually used for carriing out general tasks and processing the operating system, while the DSP compliments the ARM processor by accelerating digital signal processing tasks.

Homogeneous platforms target embarrassingly parallel computation. That is computation that have be divided into smaller sub-tasks, requiring little communication with each other. MORA and AsAP are both examples of Homogeneous platform.

Requirements

This session will outline aim of this project and give a description of the material that formed the starting point in this project.

Aim

This project is part of the development effort of the MORA architecture. The aim of this project is provide a generic OpenCL implementation, which can be used for simulation of computation on a parallel computation platform, such as a MORA device. The key deliverables of this project are an OpenCL runtime and device application, with workitems scheduling. The target architecture on which the simulation is to be run on is Tilera. However, building and optimising for the target architecture is out of the scope of this project.

Starting point

As a demostrable framework with an OpenCL host runtime and a simple device simulator, the deliverables from An OpenCL API for high-level FPGA programming by Marcin Bujar[3] make up the starting material of this project.

Device Simulator

The device simulator is a single threaded, simple application. It maintains two IP connections with the host runtime, one for control, the other for memory accesses. The host application can access memory on the device simulator via commands on the memory access connection. When instructed on the control channel to do so, the device simulator loads an OpenCL kernel as a shared library from a predetermined location on the local file directory and executes it.

Host runtime

The host runtime implements many of the functions specified in the OpenCL API[5] essential for basic OpenCL operations such as memory read, memory write and starting kernel execution.

The kernel is compiled into a shared library suitable for the device simulator through the following steps, quoted from Marcin Bujar's paper:

1. *Arguments specified in the application using the clSetKernelArg function are written to a file.*
2. *The createwrapper.py python script is called which generates the wrapper code using the kernel arguments file.*
3. *The compilekernel.sh bash script is called which compiles the kernel function and the wrapper code into a shared library.*
4. *The shared library is moved to the directory with the device simulator binary.*
5. *At runtime, the simulator performs a late linking with the shared library and calls the wrapper function, which in turn executes the kernel.*

During execution, the host application makes two TCP connections to the device application. One connection for starting kernel execution. The other to allow device memory accesses.

Host to device protocol

The protocol used between the original host and device simulator was very simple. It was almost text-based, similar to a command line interface. There were no guards against missing data.

Problem definition

The starting material provides a significant subset of the OpenCL API support to allow simulate an OpenCL platform, on which simple OpenCL applications can perform computation. However, it lacks work item scheduling capability, computation is not performed parallel. Many of the behaviours described in the OpenCL standard were not supported.

Acceptance criteria

The extended *device simulator* will be renamed as *device application*, to more closely reflect its true parallel computation cabilibility.

As specified by the project supervisior, the deliverable of this project should complete the following objectives:

- Kernel transfer from host to device simulator.
- Device application can perform parallel computation
- Device application work scheduler.

Kernel transfer

In Marcin Bujar's original work, the host application and the device simulator communicates via IP sockets. Theoretically this should allow the host and the device simulator to reside on different machines with potentially different architectures. In practice, however, the host and the device simulator must reside on the same machine, in the same filesystem directory. This is because the kernel binary is 'transferred' from host to device as a file on the file system.

The deliverable from this project should transfer the kernel binary via a communication channel between the host and the device.

Scheduler and parallel computation

The original device simulator had no scheduler. It could be thought of as a model of a device with exactly one Compute Unit. All work items were executed sequentially. A scheduler is required in order to achieve parallel computation. It would be desirable for the device application to be able to simulate parallel computation on arrays of Compute Units (CU). This requires the device simulator to be able to schedule work onto free CUs.

Planning

This section discusses the decisions taken in the planning process.

Software configuration management

Git was chosen as the software configuration management software throughout this project. The other candidate considered was Subversion. Both Git and Subversion are free and open-source. The two main reasons that Git was chosen over Subversion were: 1) Git is distributed, and 2) the ease of branch merging it offers.

Git allows changes to the project to be audited without connection to a central repository. Changes can be committed or revert at any time. Users can audit and annotate changes locally without affecting other users of the repository.

In Subversion, this can be achieved by creating a private branch, and managing changes within the branch before merging those changes back into the ‘trunk’. This would have worked in the simplest of cases where there are no conflicts between changes committed to the branch and changes made to the trunk after the branch was created. My previous attempts at nontrival merges from branch to trunk in a subversion repository have been disasterious. My experience in merging changes between Git branches have been relatively free of hassel.

Programming Languages

The original host runtime was written in C. Its functionality was also extended in C. There were no reasons to change the language.

The original device simulator was written in C. It was refactored and extended in C++. The object-orientedness of C++ suits the device simulator well with the numerious instances of Compute Units.

Design and Implementation

This session discusses the design decisions and implementation details of Host-Device communication, the OpenCL Host run-time and the Device application.

Host to Device Communications

Connections

The original design maintains two connections between the host and the device simulator, one for control, the other for memory access. Two seperate but very similar sections of code manage each of the connections. There are several major drawbacks to this approach:

- Hard to maintain
  Changes to the control link code may or may not apply to memory link code
  Having to maintain seperate lists of commands and their processing code, making it hard to add new functionalities.

- Prone to bugs and corner cases
  Error handling required when either connection is lost.

- Functionalitis are rigidly catalogorise
  Requests must either be a memory operation or a control operation.

The new implementation uses a single connection between the host and the device simulator. The TCP port on which the device listens on is specified as a command line argument. Although there is no generic way for the host application to specify which port the host runtime should try to connect with a device on. The host runtime uses the hardcoded port 5000.

Perhaps as a possible future improvement, the host runtime port could be specified in a config file, or as an environmental variable.

Although the dual connection design in the original device simulator was replaced with a single host-device communication link, a decision has been made to keep this link an IP connection.

There was a brief debate between the project supervisior and myself on whether an IP socket connection would be efficient if the host and the device were running the same machine. The project supervisior was inclined to keep the link as an IP socket connection because it would allow the device simulator on a different machine to the host application. I agreed.

Protocol

A new communications protocol between host runtime and device was designed. The new design is concise, reliable and extensible.

All commands are sent as packets with a version byte, a packet length and a command id with which to identify the necessary processing required for each command. The common start of packet allows easy packet parsing and handing off to processing functions. The length word provides demarkation between packets, allowing packets to be queued and sent as a byte stream.

Field	Length(octets)	Description
Version	1	Currently always 0x01.
Length	2	The Length of the entire packet.
Command ID	1	1 byte command identifier.
Payload	Length - 4	Command payload

Table 1. Host-Device command packet format.

Command specific payloads are given in the payload field, which always comes after Command ID. The follow table details the commands supported and their effects.

Commands

Reset (0x00)

The reset command resets the device simulator. No payload.

Memory write (0x01)

The write command writes a given segment of data into a specific address on the device.

Payload:

Field	Length(octets)	Description
Offset	4	Start offset the write should start from.
Access length	4	The number of bytes to write
Data	Access length	The data to write.

Memory read (0x02)

Request a read from device memory.

Payload:

Field	Length(octets)	Description
Offset	4	Start offset the read should start from.
Access length	4	The number of bytes to read

Memory Read Response (0x03)

Memory read response. This is the response message for a memory read request. It travels from device to host.

Payload:

Field	Length(octets)	Description
Offset	4	Start offset of the read
Access length	4	The number of bytes read
Data	Access length	Data read

Load kernel image (0x04)

This packet is used for transferring the kernel binary from the host to the device. Each packet carries a fragment of the kernel binary. A series of packets transfer the entire kernel to the device.

Payload:

Field	Length(octets)	Description
Total length	4	Kernel binary size in octets
Offset	4	The offset of the current fragment of data
Data length	4	The length of the current fragment of data
Data	Data length	The fragment of kernel data

Start kernel (0x05)

This command starts kernel execution. No payload.

Global work size (0x06)

This command sets the global work size for the pending execution.

Payload:

Field	Length(octets)	Description
Global X	4	Work size on dimension N = 0
Global Y	4	Work size on dimension N = 1
Global Z	4	Work size on dimension N = 2

Acknowledgement(0xFE)

Acknowledgement to the previous command. No payload

No Acknowledgement(0xFF)

No Ack to the previous command. No payload.

OpenCL Device Application

The OpenCL Device Application is responsible for fulfilling operation requests from the host application, such as memory access and kernel execution. The device application is also responsible for scheduling work items across available Compute Units.

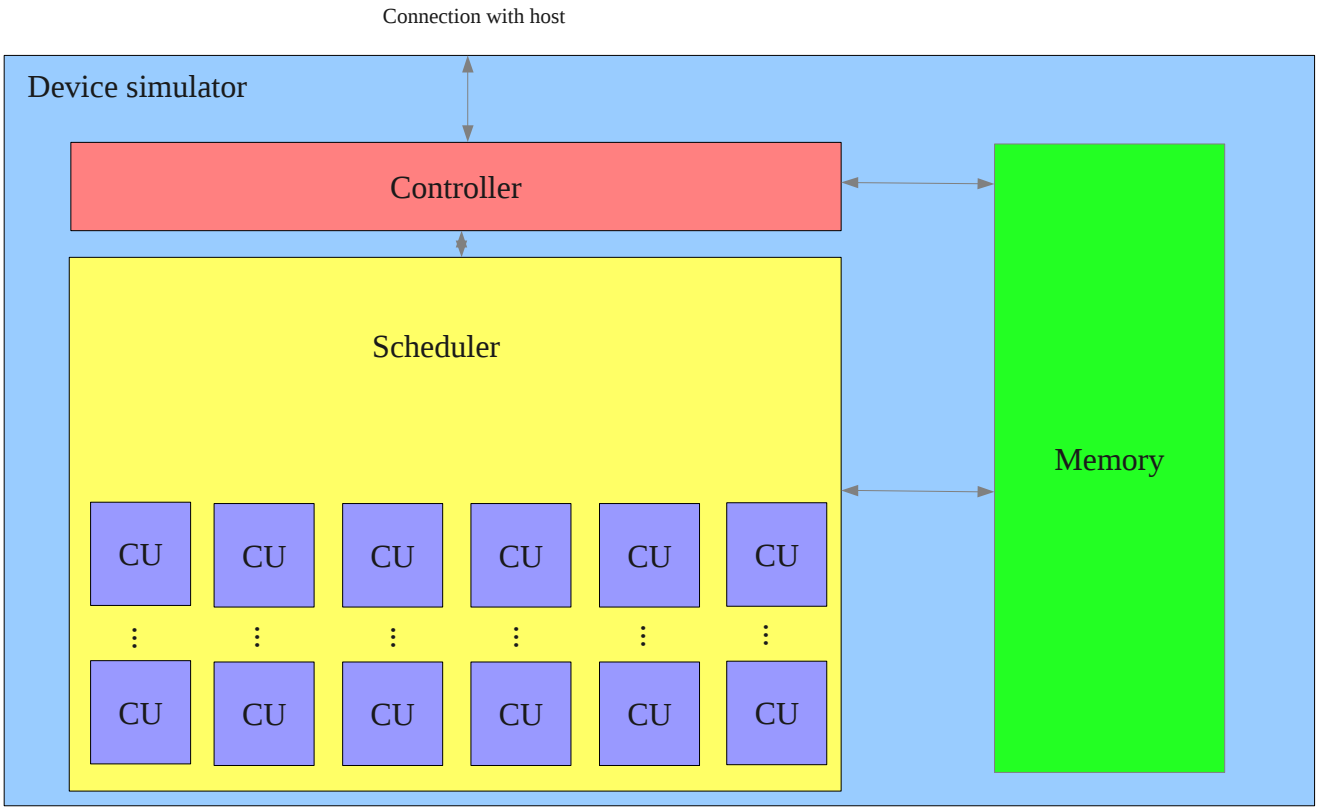

Figure 1 – Internal structure of the device application.

Memory

At startup, the device application allocates a region of memory from heap to serve as the device's memory space.

Controller

The controller object is instantiated at start-up. Initially it listens on the TCP/IP port specified in the device application's command line arguments. Once connected to by a OpenCL host, it maintains connection to the host. It also decodes and processes commands from the host. Memory access operations are processed directly in the controller object, while details of kernel computation requests are passed on to scheduler object.

Scheduler

The scheduler in an integral part of the device application. It's job is to dispatch tasks to any CU that are free. The scheduling algorithm in this case is simple, because the work and the amount of work is known. All work items are identical, and have the same priority. Whenever a CU becomes free, the scheduler only has to dispatch it with the same work again with a different global ID.

Here is how the scheduler is implemented. The scheduler maintains a free CU FIFO queue. The start kernel command from the host causes the scheduler to start executing the work. As long as the work is unfinished, the scheduler checks if the free CU FIFO queue is empty. If it is not, the scheduler pops the CU off the the front of the queue, sets it's global ID and kernel, starts the CU's execution, and throws its reference away. When the CU has completed its work item, it joins itself to the back of the scheduler's free CU FIFO queue and terminates its own thread.

Compute Unit

Each instance of compute unit class represents a compute unit on a MORA device. The CU is instanciated with the reference to a scheduler, and a unique ID for identification purpose. When being dispatched with a work item, each CU starts its own thread to process the work item. After execution of the work item is complete, the thread joins the CU instance to back of the scheduler's free CU FIFO queue before terminating itself.

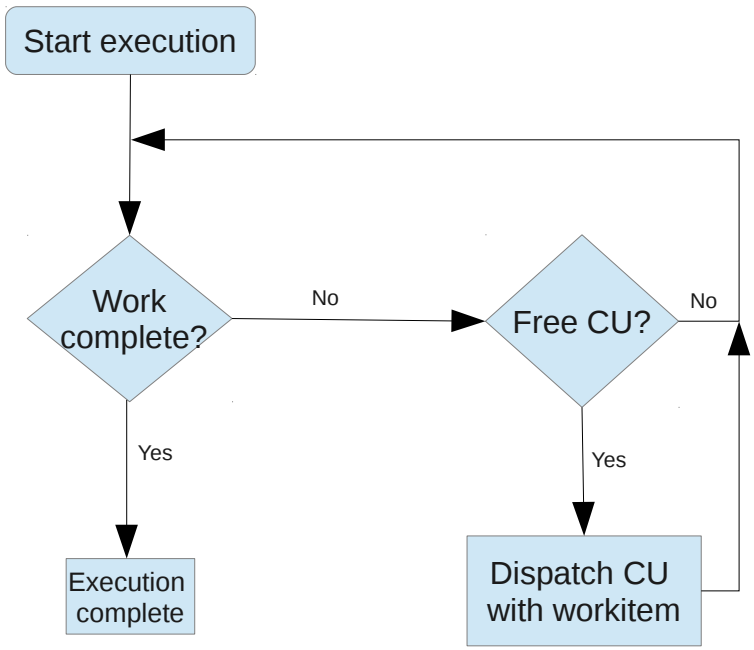

Figure 2 – Scheduler logic

Host OpenCL runtime

The host OpenCL runtime is a static library, which implements the API and behaviours specified in the OpenCL standard [5]. OpenCL application links against the runtime library at compile time.

Typical order of operations by the application:

1. Create platform
2. Create context
3. Create command queue with context
4. Create program
5. Build program
6. Create kernel with program
7. Create memory objects
8. Initialise memory on device by enqueue memory write commands
9. Enqueue NDRange kernel
10. Enqueue memory read from device to read back computation results.
11. Synchronise by calling clFinish()
12. Inspect returned memory data. Repeat steps 8 -12 if necessary.

Creating context

Before any OpenCL operations can be processed, the application must interrogate the OpenCL runtime for platform and device statuses. An OpenCL runtime may support multiple platforms. Each platform can have zero or more devices. Once the OpenCL application has determined that the OpenCL platform and devices are available and meets its requirements, it creates a context for the platform it wishes to operate with.

Creating command queue

In this implementation, the connection between the host runtime and the device is made during context object instantiation.

Creating OpenCL program

Support for creating opencl program with source has been added to the extended runtime. Source code text given with clCreateProgramWithSource call is kept within the newly created program object.

Creating memory(buffer) objects

The application requests allocation of device memory for the pending OpenCL computation work by creating memory objects on the context. These memory objects, or the reference to its associated region of memory are used for enqueuing memory access operations to the device, are passed to the kernel as arguments.

In this implementation, creation of a new memory object allocates the required memory by merely advancing an offset variable. For example:

- When context is created, free device memory starts from offset 0.
- The application creates buffer *A* of size a.
- The runtime reserves the memory for buffer *A* by advancing the free device memory offset variable to a. And set the start offset of *A* to 0.
- When the application creates another buffer *B* of size *b*, the runtime perform the same allocation steps to set B to start from device memory offset *a*. And the free device memory offset is advanced by *b* to become *b*+*a*.

So implicitly, buffer *A* starts from offset 0 and ends at *a*-1 and buffer *B* starts from offset a and ends at *a*+*b*-1.

Kernel compilation

Kernel compilation in our implementation is two stage process. In the original framework, program build and kernel build were done as a single step at NDRangeKernel

The reason for spliting the kernel compilation process is that support for clBuildProgram was added. According to the OpenCL specification[5], clBuildProgram is supposed to invoke the manufacturer specific OpenCL compiler if the program was created with source. After clBuildProgram returned successfully, the OpenCL application is supposed to be able to obtain the compiled program binary, which will allow it to save time in the future by creating the program with binary.

Neither getting program binary nor creating program with binary is supported in the deliverable, but building just the program at clBuildProgram call paves the way to supporting these two functions in the future.

We cannot compile the program with the kernel wrapper at clBuildProgram because the main program symbol name is known only after an OpenCL kernel object has been created using the program object.

Building the program

First the program is compiled at clBuildProgram call into an object file. At this point the build parameters given in clBuildProgram is filtered so that unsupported OpenCL build flags are not passed on to the C compiler. Macro definitions given in the build options are passed on to the build process. This is done by writing the filtered build options into a specific file, before executing the build script buildProgram.sh. Below are the steps involved in building the program object file:

1. Write program source to tmp.cl
2. Filter and write build options to tmp.options
3. Call buildprogram.sh

If the build was successful, program.o would be created.

Building the kernel

The second part of the kernel compilation involves linking the program with the kernel wrapper. This occurs at the time NDRangeKernel event object is being processed.

The following are the steps involved:

1. Call createwrapper.py to create the kernel wrapper with the kernel name
2. Call compilekernel.sh to compile the kernel wrapper and link it with program.o to create the kernel shared library.
3. If step 1 and step 2 were successful, kernel.so would be created.

Since work item scheduling has been moved to the device simulator, the simple scheduling loop has been taken out of the kernel wrapper. Also the kernel wrapper function has been extended to support 3 dimensional global Ids.

Start of computation

The NGRangeKernel call from the application is a request for execution of a given kernel. In the previous section we described the process of kernel compilation. Once the program has been linked with the kernel wrapper and a binary kernel is produced, it is sent to the device in small increments throught the host-device communication link. When the kernel transfer is complete, the host signals the device to start executing the kernel. Whe execution is complete, the device sends an ACK message back to the host. At this point the host should unblock any calls on clFinish or clFlush.

Result read back

A typical application would read back the result from the device. It will then decide whether to enqueue more operations or terminate.

Command queues

In OpenCL, a command queue is an object for queueing operations to OpenCL device. In general, operations that the OpenCL host application wishes to perform on the device do not get executed immediately. A set of OpenCL API functions allows the application to 'enqueue' these operations. The OpenCL host runtime then dispatches these operations to the device to be performed, asynchronous to the host application. The host application is responsible for synchronisation, for example, by calling clFinish(), which blocks until all operations enqueued for an OpenCL device on the given command queue are completed.

OpenCL allows multiple command queues to be created to the same device, which makes it possible for the application to enqueue multiple streams of operations the device to be executed in parallel. This implementation does not support such feature. It has been marked as a potential future improvement.

The original framework did not support command queues. The host application was allowed to create command queues, but all OpenCL enqueue routines blocked. Each 'enqueued' operation was dispatched to the device, executed and result returned, before the enqeue function returned.

In order to be more compliant with the OpenCL specification, the command queue object was extended to allow actual queuing of objects. The queue of command objects enqueued by the application is processed in the order they are enqueued, asynchronous to the application.

The command queue allows the meaning implementation of synchronisation mechanisms such as the clFinish and clFlush calls, as well as emulated support for barrier command objects. The barrier command object, although queued and progress through the command queue as it should, does not provide any meaningful synchronisation effects because out of order execution is not supported in this implementation. Out of order execution has been noted in [potential future improvement]

Implementation details

Each command queue object creates its own worker thread upon instantiation. The worker thread executes independent from the application thread.

The command queue was implemented using single linked-list of command objects(also known as events). When the application calls an enqueue API function, a command object is created to hold the necessary information about the enqeued operation. The command object is added to the end of the command queue's linked list. Each command object also holds the processing status of the command. As per OpenCL specification, the reference to the command object/event can be returned to the application by the enqueue API call to allow the application to track of the command's progression through the command queue, through the use of clGetEventInfo() API call.

OpenCL specifies the following possible processing statuses for an event;

- CL_QUEUED – The event is enqueued
- CL_SUBMITTED – The event has been submitted to device for processing
- CL_RUNNING – The event is being processed by the device
- CL_COMPLETE – Event has been processed.

A simple approach to the command queue would be to dispatch an event, then delete the event object when it is complete. The advantage of this simple approach would be that the head of the linked list will be the next event to process. However, this approach would not be compliant with the OpenCL standard, as OpenCL allows the application to track the progress of the events it enqueued, and to use user events waiting on previous events to achieve synchronisation. The command queue would therefore need to keep a reference of an event even after it has been completed.

In the approach taken in the actual implementation, the command queue keeps a 'current task' pointer, which always points to the next event being dispatched. When an event joins the command queue, its status is given as CL_QUEUED. The command queue's worker thread periodically checks if there are events in the queue waiting to be processed. When the 'current task' pointer advances the an event, the event's status is changed to CL_RUNNING. The command queue object then proceed with dispatching the operation detailed in the event object to the device for processing. Once the operation is complete, the event's status is changed to CL_COMPLETE, and the command queue advances the 'current task' pointer to the next event (or NULL if the end of the linked-list is reached). Note: CL_SUBMITTED is never given as the status value of an event because in this implementation, operations are running as soon as it is submitted to the device.

Completed objects cannot be maintained indefinitely. Due to project time constrain, this implementation did not implement the event removal method specified by the OpenCL standard, which stipulates that event objects maintain a reference count and should only be deallocated once the event has been fully released, and the reference count has reached zero. Instead, this implementation timestamps completed events, and deallocates them after a specific amount of time has past. This expiration time is arbitrarily chosen as 5 seconds.

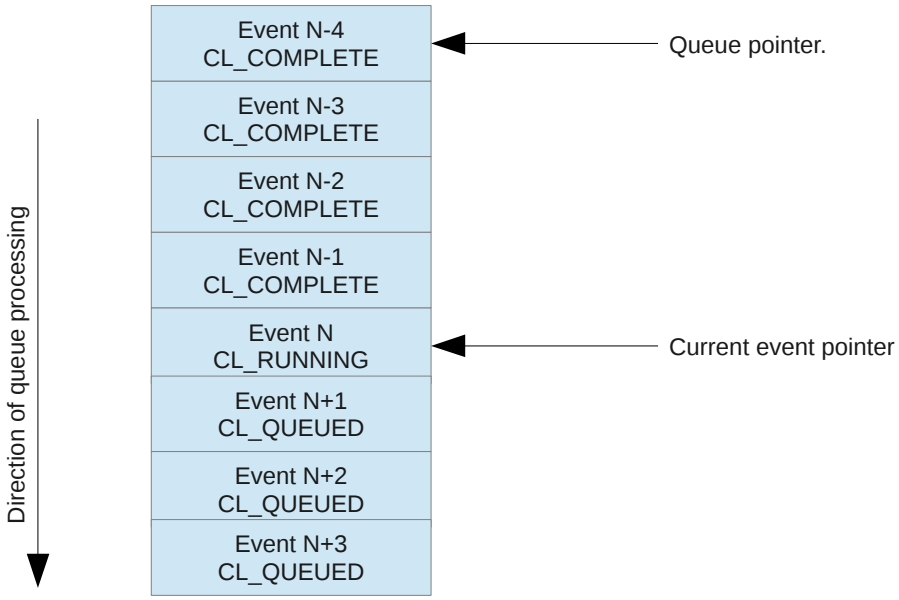

Figure 3 *– Command queue*

Command queue logic

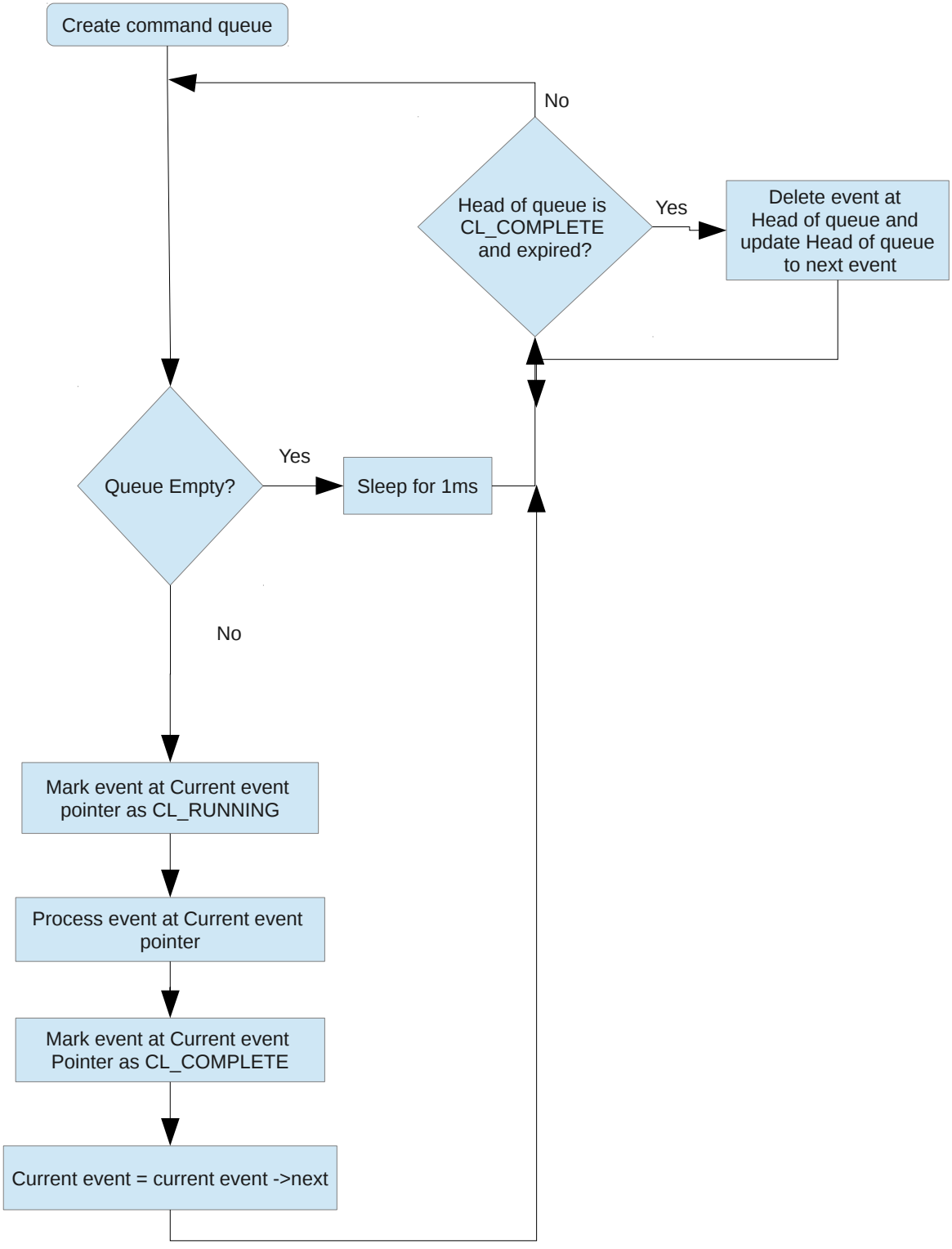

Figure 4 *– Command queue logic*

Debugging process

During the implementation process, problems were diagnosied largely through the use of debug messages and the use of GNU Debugger.

The design was exercised using the 7-elements array addition, and the 16 by 16 matrix multiplication examples, given by Marcin Bujar's work. These examples provide the work to exercise the design and verify the results after computation is complete.

Debug messages were used to trace the path of execution and to identify points of failure. In cases where execution was interrupted by segmentation fault for non-trival reasons, GNU Debugger was used to produce a stack trace at the point of failure and to allow inspection of pointer values.

Once the design were refined to achieve computation of the array addition and matrix multiplication samples correctly, the *Monte Carlo method for stock options pricing* example by Intel for evaluating its OpenCL implementation, was adapted to flush out missing functionalities in the implementation.

Evaluation

This section documents the evaluation of the final design.

Evaluation using Intel's OpenCL sample

The design was evaluated using the *Monte Carlo method for stock options pricing* example by Intel for its own OpenCL implementation.

Output from this implementation:

```
Platforms (1):
   [0] Prototype Intel [Selected]
Devices (1):
   [0] Mora array [Selected]
Build program options: "-D__DO_FLOAT__ -cl-denorms-are-zero -cl-fast-relaxed-math -cl-single-precision-constant -DNSAMP=262144"
[ WARNING ] Unable to load OpenCL source code file "montecarlo.cl" at the default location.
Trying to open the file from the directory with executable... OK
Running Monte Carlo options pricing for 200 options, with 262144 samples
Size of memory region for each array: 800 bytes
Using risk free rate = 0.05 and volatility = 0.2
Host time: 1.63676 sec.
Host perf: 122.192 Options per second
Host time: 1.67135 sec.
Host perf: 119.664 Options per second
Host time: 1.74267 sec.
Host perf: 114.766 Options per second
Host time: 1.7359 sec.
Host perf: 115.214 Options per second
Host time: 1.7997 sec.
Host perf: 111.13 Options per second
Host time: 1.82412 sec.
Host perf: 109.642 Options per second
Host time: 1.84638 sec.
Host perf: 108.32 Options per second
Host time: 1.93385 sec.
Host perf: 103.421 Options per second
Host time: 1.9933 sec.
Host perf: 100.336 Options per second
Host time: 1.98305 sec.
Host perf: 100.855 Options per second
```

It is expected that performance from Intel's own OpenCL implementation would be better. I was unable to run the same sample on Linux because at the time of writing Intel does not support OpenCL for i7 devices under linux. And I was unable to compile the sample under Windows 7 because I was unfamiliar with Windows environment.

Future improvements

The following list shall be considered as possible improvements.

- Handle CL build options

  During evaluation with Intel’s sample program it was found that certain kernel constants are defined through build options. In order for the kernel to be successfully compiled, the build options have to be passed to the compiler. The build option string contains OpenCL specific options, which are not supported by the GCC compiler. These options are

filtered out before being passed to the build process.

- The OpenCL runtime implementation is still not complete

  This implementation of OpenCL runtime is still not complete. For instance, getInfo calls such as getDeviceInfo does not yet support returning of all possible variables the application can request. It was found at a very late stage of this write up that setKernelArgs does not always work. These issues can cause some applications to fail.

  Better error handling and reporting code is required in various places.

- Better object tracking

  In the OpenCL paradigm, platforms, devices, programs, memory, commands(events), command queues are all objects. This implementation does not keep track of what objects are created by the host application. Instead, it relies on the host application to keep track of what it has created and to only pass valid objects in and out of the runtime library. By not keeping track of objects, this implementation is susceptible to memory leak problems.

  By keeping a pool of valid objects, the runtime will be able tell if an object passed in from the application is still valid. It would also allow the runtime to be more autonomous in its object management. That is to say, the runtime would not have to complete tasks like object deletion during application API calls, making it a blocking call.

  Proper removal method should be implemented for event object.

- Multiple command dispatch

  The current implementation of the Host-Device protocol only allow one command to be despatched at any given time. It is possible that commands in the command queue could be issued to device at the same time.

- Out of order execution.

  It is conceivable that the application might create multiple command queues, each filled with commands that can be handled out of order.

- Support multiple platforms/devices

  Support for more than one device could be an extremely desirable feature. For example, it would allow a single application to distribute work to multiple devices, or even multiple architectures. It will also add substantial usefulness to the implementation in real world applications.

- Extend Host device protocol

  Many of the advanced features in OpenCL cannot be supported with the currently limited set of commands and responses in the Host-Device protocol, e.g. Multiple command dispatch.

  The protocol could also do with some methods to allow the host to get more information on the device's status and capabilities.

- Device Memory management

Device memory management is an area that might have room for improvements, in order to better simulate actual OpenCL device memory model. The current implementation of the device 'statically' allocates a piece of dynamic memory at startup and treat it like the global memory for the device.

- Support for customising compute unit execution for real device

  It would not be hard to modify the current device simulator to act as a controller and scheduler of work to an actual implementation of a novel implementation. Perhaps support could be added to the CU class code or to the code structure of the device simulator in general to make this even easier and more portable.

Other Potential applications

The generic nature of this implementation lends itself to some interesting potential applications. For example, it can serve as the generic OpenCL support for architectures that do not yet have native OpenCL framework.

Another intriging idea is to use the host runtime as a wrapper to other proprietory opencl implementations. As an operator of a small GPU farm, I have multiple machines that each run a copy of the OpenCL application using the same OpenCL runtime supplied by the same manufacturer. It is time comsuming to manage all these instances of OpenCL computations individually. By running a modified instance of the device application on each of these machines, and making the runtime connect to the device application on all the machines. I can create a pool of different GPU devices under the same platform. Then only a single instance of the OpenCL host application is required. This might pave the way to the commoditisation of GPU resource.

References

[1] Wikipedia on Parallel Computing
http://en.wikipedia.org/wiki/Parallel_computing

[2] MORA – An Architecture and Programming Model for a Resource Efficient Coarse Grained Reconfigurable Processor

[3] An OpenCL API for high-level FPGA programming

[4] Cramming more components onto integrated circuits, Gordon Moore

[5] OpenCL 1.1 specification, Khronos Group
http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[6] Heterogeneous and Homogeneous computing
http://www.ece.ucdavis.edu/vcl/pubs/2008.03.JSSC/

[7] A C++-embedded Domain-Specific Language for Programming the MORA Soft Processor Array - ISBN 9781424469666

[8] An Asynchronous Array of Simple Processors - *VLSI Computation Laboratory, University of California, Davis*

[9] S. Craven, C. Patterson, and P. Athanas, “A Methodology for Generating Application-Specific Heterogeneous Processor Arrays,” in HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, vol. 39, p. 251, Citeseer, 2006.