A Generic OpenCL Implementation to Support Novel Architectures

Hau Tat Luk

Institute for System Level Integration

Student ID: 091613958

26th August, 2013

Contents

Abstract

For a long time, computer designs in general have been based around uniprocessor. The increase in performance has been gained through rapid advances in silicon fabrication technologies and processor design techniques. In recent years, the realisation that sustainable performance increase cannot be gained through feature size shrink alone and the shift of users' focus from speed to power efficiency have prompted the industry to explore multiprocessor machines as a way forward in providing ever increasing computing power. As a result, multiprocessor architectures have gained significant interest from the research community in recent years.

Multiprocessor architectures provide performance gain by taking advantage of explicit thread level and process level parallelism in applications. Most applications are not embarrassingly parallel, meaning that while parts of the application are executed in parallel, a certain amount of communication exists between them. The level of parallelism differ between applications, so do the patterns of inter-thread communications. Research in multiprocessing architectures typically focus on processor core design, multiprocessor topology, and communications network architecture. The Media-Oriented Reconfigurable Array (MORA), for instance, investigates the benefit of having reconfigurability in processor cores to suit a given application.

Irrespective of research focus, researchers will inevitably meet to need to simulate their designs with real computation. This paper presents a framework that enables the reuse of existing applications for this purpose. This framework originally aimed to facilitate the simulation of MORA device, but can be adapted to dispatch computation to an alternative design.

1 Introduction

This paper describes a computation dispatch framework designed to facilitate research on novel multiprocessor architectures and on-chip network designs. The framework aims to reduce the effort required to provide computation for developing parallel processing architectures for the purpose of collecting statistical and performance data. The first section attempts to present a summary of the history of the progression of microprocessors in terms of performance and the limiting factors that prompted the move towards multiprocessor designs. The next section provides a background to the development of the framework. Section 3 will describe the framework in more detail and explain how it will achieve its purpose. Section 4 presents a demonstration of the implementation with a few simple applications, as well as an off-the-self application. Finally we document the limitations of the current implementation and propose a list of possible future improvements.

1.1 A brief history of multiprocessing

Mainstream computing resources have traditionally consisted of uniprocessor machines, that is, each machine utilises no more than one central processing unit. This was true for consumer electronics, personal computers and commercial mainframes.

1.1.1 Dependence on shrinking transistors

Until the early 2000s, the general purpose computer industry have largely relied on ever more powerful microprocessors to satisfy the demand for higher performance. The performance gained was largely due to advances in silicon fabrication technologies, which allowed the use of ever smaller transistor sizes in integrated circuitry. Smaller feature sizes translate to more gates on a given area of silicon, lower logic delays and lower operating voltages. These in turn allowed higher operating frequencies, lower power consumption, and smaller die sizes. [1.1]

The Intel 4004 processor, generally considered to be the first commercially available single IC processor, was first advertised on 15th November, 1971. [2] It was manufactured with minimum feature size of 10 microns, (10µm) and had a maximum clock rate of 740kHz (1.35µm minimum clock cycle, 10.8 µm instruction cycle). [3]
Fast-forward to 2006, when Intel introduced the last of its Pentium 4 processors based on its Cedar Mill architecture, which was fabricated using 65nm technology, 153 times finer than that used by the Intel 4004. The Pentium 4 661 has a factory clock rate of 3.6GHz, has a maximum register width of 64 bits, supports hyper-threading (superscalar processing), 2MB L2 cache, and an instruction set far richer than that offered by the 4004 [4]. The vast amount of extra memory and features included in the Pentium 4 was made possible by the smaller feature size and therefore the larger number of devices that can fit in the design.

However, the trend of shrinking feature sizes cannot be sustained indefinitely. Feature size of a single atom marks the lower bound of what can be achieved with silicon technology in a sense compatible with classical computing. [5] As of July 2013, the smallest feature size used in the fabrication of commercially available general purpose processors is 22nm [6][7]. Although we are not quite hitting the single atom limit yet, manufacturing becomes increasingly costly as feature size progresses down the nano scale. At this scale, quantum effects become significant. The design becomes sensitive to manufacturing defects, and the overall manufacturing yield is reduced.

Increasing operating frequencies coupled with densely packed logic also present problems with heat dissipation. The operating frequency of the Pentium 4 Cedar Mill processor is not limited by logic delay, but by heat dissipation. With extreme cooling provision, it has been reported that a Pentium 4 Cedar Mill 631 3GHz processor can be over-clocked to up to 8GHz. [8]

1.1.2 Performance from better designs

Design techniques also played a role in further increasing processor performance. Techniques such as pipelining and superscalar architectures exploit instruction level parallelism.

Pipelining [1.2] takes advantage of the fact that the processing of instructions usually involves multiple stages such as Instruction Fetch, Instruction Decode, Execution, and Memory Write Back, and may take multiple clock cycles to complete. By splitting the instruction processing into stages, a number of instructions can be processed at different stages at the same time, thereby increasing instruction throughput. The time required for the logic in any single stage to settle would also be less than the time required in processing a whole instruction. This means the overall clock rate of the processor could be increased.

A superscalar design [1.3] contains more that one set of logical resources required to support its instruction set architecture, i.e. a pool of resources. In contrast with the pipeline approach, a superscalar processor exploits tendency in the instructions to depend on different processing resources. Clearly, all instructions required fetching from memory, but some instructions require memory access, some require arithmetic processing and some are conditional and will cause branching. By carrying a pool of different processing resources, a superscalar processor can dispatch multiple instructions at the same time. Some instructions don't even depend on each other's results, which means the superscalar processor can effectively act like a multiprocessor and simultaneously execute those instructions outright.

The inconvenient truth about performance enhancing processor design techniques is that they all suffer from the law of diminishing return. Repeated application of these techniques to future designs will often yield ever diminishing performance gains.

While longer pipe lines and shorter pipe stages enable high operating frequencies and higher theoretical instruction throughput, branch miss penalty and other pipeline hazards become more significant. First, instructions are not indefinitely divisible, so it is not possible to increase pipeline stages indefinitely. Secondly, an execution path of a typical program will inevitably hit a conditional branch instruction, and when that happens, and the instruction pipeline was loaded with the wrong branch of execution, the entire pipeline after the branch has to be purged. In a long pipeline, this means significant losses in instruction cycles.

Hardware and software interlocks are employed to ensure the simultaneous execution of instructions in a superscalar processor does not affect program behaviour. These mechanism determine interdependencies between upcoming instructions and decide whether they can be executed out of sequence. As the amount of simultaneous execution increases, these interlocks become more sophisticated, adding complexity to the overall processor design.

1.1.3 Shift to multiprocessing

The idea of using of multiple computer to provide additional performance is not new. It is typical for businesses and government organisations to employ clusters of machines to satisfy their computing needs(also known as Loosely Coupled Multiprocessing), a few examples of such needs are serving online contents to the public, information management, and simulation-aided research. The use of multiple computers allow these tasks to be performed at an acceptable rate. This level of performance may not by achievable on a single uniprocessor machine at the time. The engineering of these applications involved the developers factoring the multiprocessing environment into their designs. These developers thought in terms of threads and processes, and accepted the fact that each process will not run at spectacular speeds, but that the required overall performance will be achieved through the collective of processes executed simultaneously.

The wider industry has, for a long time, clung to the paradigm of architectural transparency. Architectural changes that led to performance gain must remain obscured from the users, and the users must not be required to change their programming practices. The exhaustion of design techniques to gain performance, pending physical limitations of silicon manufacturing processes have called into question the sustainability of such paradigm.

In recent years, there had been a rapid increase in the interest of multiprocessing architectures. In addition to including multiple processor cores in traditional single processor designs with enhanced inter-processor communication, researchers have come up with a variety of multiprocessor architectures that offer speed-up and better power efficiency, in certain types of computation, by employing novel processor and communication network design ideas. An example is the Asynchronous Array of Simple Processors (AsAP), which explores the performance of scalable arrays of simple processors with loosely coupled, mash-like processor interconnect. The developers of AsAP propose that such architecture would be suited for data stream processing applications, where the processing of a stream of data can be separated into small simple stages.[9] Another example is the Media-Oriented Reconfigurable Array (MORA), which explores the idea of processors reconfiguring to suit the application at runtime, bringing together the high performance of hard processor cores and the flexibility FPGAs offer. [10]

A research team would typically produce a working model of the architecture researched. The model would be benchmarked with various computational tasks to produce performance data. To ease production of benchmarking applications, and to promote wider adoption, the research team must produce a development toolchain for their architecture. The unconventional nature that many of these architectures means that even after the toolchains have been made available, software has to be written in ways specific to the architecture, e.g. by using a modified version of a programming language or by utilising an API specific to the platform. This results in the lack of portability in code developed for these architectures, and the same class of problems had to be rewritten over and over again by different research teams. This project aims to mitigate the problem by creating a framework that enables the use of common, existing applications on parallel computation architectures. The project hopes to enable researchers adopting this framework to focus on the parts necessary for computation to benefit from their architecture.

1.2 Origin of this project

This project is part of the development effort on the Media-Oriented Reconfigurable Array. [10] The MORA architecture is a stream processor array that is configurable on the fly to speed up computation. It bridges the difference in performance and reconfigurability between hard processors and Field Programmable Gate Arrays(FPGAs). MORA offers some of the flexibility available in FPGAs while retaining the performance of being fabricated directly on silicon. This project aims to provide the means to simulate the MORA architecture on general machines, with existing parallel computation.

2 Analysis

2.1 Starting point

The deliverables from *An OpenCL API for high-level FPGA programming* by Marcin Bujar[11] make up the starting material of this project. In those deliverables are a partial OpenCL host runtime and a simple device simulator. The deliverables are demonstrable with three sample applications, “helloworld”, “addition” and “matrix”.

2.1.1 Device Simulator

The device simulator is a simple single threaded application that carries out computation. It maintains two IP connections with the host runtime, one for control, the other for memory access. 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.

2.1.2 Host runtime

The host runtime implements essential functions specified in the OpenCL API required for memory read, memory write and starting kernel execution.

2.1.3 Kernel Compilation

The OpenCL source provided by an application is compiled into an OpenCL kernel with the use of scripts. The *createwrapper.py* script creates a wrapper that includes a kernel header, kernel arguments and a for loop in which the source is placed inside. The *compilekernel.sh* compiles the wrapped source into a shared library, which would be the kernel.

2.1.4 Host to device protocol

The host and device simulator communicate via a simple, almost text-based the protocol. The command format is similar to commands used a command line interface. Commands between the host and device are transferred on the control connection, while data from memory reads and writes are carried on the data connection.

2.1.5 Operation

During operation, the device simulator acts as the server and the host acts as the client.
The host application makes two TCP connections the device simulator. The host application would typically instruct the host runtime to instantiates and initialise the memory buffers required by the computation. Once this is done, the host application will instruct the host runtime to compile and execute the kernel.

The steps taken by the host-runtime to compile the kernel as quoted from Marcin Bujar's paper, are:

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.*

Once the kernel is compiled the host-runtime instructs the device simulator to begin executing the kernel, which would be at a specific location and named *kernel.so*.

At the end of the computation, the host reads the result back from the device simulator and disconnects from the simulator. The simulator will then exit.

2.1.6 Files

```
|-- device                    (Device Simulator directory)
|   |-- core.c                (Device Simulator)
|   |-- device.h              (Device Simulator definitions)
|   `-- Makefile                   (Device Simulator Makefile)
|-- host                           (Host runtime directory)
|   |-- cl_context.c          (Host runtime OpenCL context implementation)
|   |-- cl_cqueue.c           (Host runtime OpenCL queue implementation)
|   |-- cl_defs.h                  (Host runtime OpenCL implementation definitions)
|   |-- cl_device.c           (Host runtime OpenCL device implementation)
|   |-- cl_event.c            (Host runtime OpenCL event implementation)
|   |-- cl_kernel.c           (Host runtime OpenCL kernel implementation)
|   |-- cl_mem.c              (Host runtime OpenCL memory implementation)
|   |-- cl_platform.c         (Host runtime OpenCL platform implementation)
|   |-- cl_program.c          (Host runtime OpenCL program implementation)
|   |-- debug.h               (Host runtime debug macro)
|   |-- dev_interface.c       (Host runtime to device communication)
|   |-- dev_interface.h       (Host runtime to device communication definition)
|   `-- Makefile                   (Host runtime Makefile)
|-- addition.cc               (Demo application, addition)
|-- compilekernel.sh          (OpenCL Kernel compile script)
|-- createwrapper.py          (OpenCL Kernel wrapper generation script)
|-- helloworld.cc             (Demo application, hello world memory readback)
|-- hello_world.cl            (Demo application, hello world OpenCL file)
|-- int_sum.cl                (Demo application, addition OpenCL file)
|-- kernel.h                  (OpenCL Kernel definitions)
|-- Makefile                  (Top level Makefile)
|-- matrix.cc                 (Demo application, matrix multiplication)
|-- matrix_mul.cl             (Demo application, matrix multiplication OpenCL file)
|-- ocl.h                     (OpenCL definitions for demo apps)
`-- run_host.sh                    (Script to run demos)
```

2.2 Project objectives

This project aims to produce a demonstrable framework for simulation and exercising of novel multiprocessing architectures. The material described in the *Starting point* section already provides some form of simulation of execution, it makes a good foundation for this project to build on. The project will reuse the model of the starting material. That is, to provide the host application with an OpenCL host runtime that communicates with a device simulator to carry out computation.

In order to fulfil the purpose of simulating a MORA device on conventional mainframe environment, the device simulator will need to possess active task scheduling capability.

Key deliverables of this project are:

- OpenCL Host runtime
- Robust Host-Device communications protocol
- A device simulator with task scheduling.

2.3 Technologies

2.3.1 Software configuration management

Software configuration management (SCM) is an important part to any respectable software projects. An effective SCM allows changes to the product to be tracked and be rolled back should it be necessary. An SCM will also ease version management. In the straight-forward case, an SCM allows a snapshot to be taken at the time of release, so that all released products are reproducible from the source repository. Where different feature sets on the same base product are required, a good SCM will allow branches from the base code. Release specific features can then be committed to just the relevant branches. Where branches are used, effective branch merging facilities are also important.

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 supports branching, and 2) the ease of branch merging it offers.

While one of the main differences of Git from Subversion is that it is distributed, the distributed nature of Git repositories is largely irrelevant to this project since no more than one user actively contribute to the code. One useful feature of Git, however, is that it allows changes to the project to be audited without connection to a central repository. Changes can be committed or reverted at any time, even when the user is offline.

In SCM terminology, a branch is commonly defined as a duplicate of a revision controlled object at a point in time, which modification can be made to, without affecting the original revision controlled object duplicated. Wikipedia defines it as "*the duplication of an object under revision control so that modifications can happen in parallel along both branches*". [12] Subversion does not explicitly support branching, although it can be done by the use of subdirectories. In contrast, Git tracks all changes in branches explicitly.

When it comes to merging, Subversion does support a merge command, which merges changes between files or trees of files. In my experience with Subversion, this does work 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. On non-trivial merges however, the art of reconciling conflicting subversion commits without disastrous consequences continues to elude me. My experience in merging changes between Git branches have been far more pleasant.

For the above reasons, Git was chosen as the software configuration management software.

2.3.2 Programming Languages

In the starting material, the host runtime and the device simulator were both written in the C language while the demo applications were written in C++.

As the OpenCL API is defined for C, OpenCL implementations are typically also written C. The new host runtime implementation remained written in C. The host applications can be written in C or C++ (or other languages) through binding.

The new device simulator has to emulate parallel computation at least at the thread level. It will have to simulate processing units executing threads of computation in parallel, transitioning between being free and begin busy. This makes an object-oriented model mode suitable. The new simulator has been written in C++, which natively supports the object-oriented model.

2.3.3 The OpenCL standard

OpenCL stands for Open Computing Language. Khronos Group is the standardising body. It is an open, royalty free standard aimed increasing portability of parallel computation. [13] As mentioned in Section 1.3, parallel computing hardware comes in different designs with different methods to achieve speed-ups. For instance, Intel's general purpose multiprocessor offering is vastly different from the GPGPUs produced by ATI Technologies, the former has a few very powerful processor cores, the latter has a vast number of simpler processors. Hardware vendors have to provide Software Development Kits (SDKs) for programmers to create applications for their platforms. OpenCL specifies a standard API that lets vendors hide the platform specific details in their OpenCL Implementations and allow applications to be written in a standardised, platform independent way.

The OpenCL paradigm is suited for applications requiring the same computations repeated many times for variable input and output. Given the nature of the MORA architecture this project ultimately aims to simulate (array of processors), OpenCL is perfectly suitable. Many processor vendors including Intel[14], AMD, ATI Technologies[15] and Nvidia[16] have provided their own OpenCL runtime implementations for their products, and many applications utilising OpenCL already exist.

3 Design and Implementation Details

In this section, designs of key deliverables will described in detail.

3.1 Host to Device Communications

3.1.1 Connections

As described in section 2.1.1, the two separate connections were used for communications between the host and the device simulator, one for control, the other for memory access.

The new communications mechanism uses a single connection between the host and the device simulator. By employing just one connection, the communications management code can be simpler and more maintainable. Using a single connection also eliminates potential synchronisation hazards, e.g. commands/data arriving at the same time contenting for the same memory locations.

The port the device simulator listens on can be specified as a command line argument. It is not possible to specify from the host application which port the host runtime should try to connect to the device simulator on. The host runtime uses the hardcoded port 5000 and will always try to connect to the device on localhost:5000.

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.

A proxy program is included in the deliverable, which can be used to proxy the connection between the host and the device simulator to allow the device simulator to run on a remote machine. It was written by Duane Wessels [17].

3.1.2 Protocol

A communications protocol between host runtime and device was redesigned to be 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.

3.1.2.1 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.

Table 2 - Memory write command payload.

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

Table 3 - Memory read command payload.

Memory read response (0x03)

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

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

Table 4 - Memory read response payload.

Load kernel image (0x04)

This command is used for transferring the kernel binary from the host to the device. Each packet carries a fragment of the kernel binary and the offset of that fragment. 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

Table 5 – Kernel transfer payload.

Start kernel (0x05)

This command starts kernel execution. Before sending this command, a kernel must have been transferred using *Load Kernel Image* commands, and a global work size must have been set using *Global work size* command. This command has no payload.

Global work size (0x06)

This command sets the global work size for the impending execution. It is typically sent after kernel transfer, and before kernel start.

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

Table 5 – Global work size payload.

Acknowledgement(0xFE)

Acknowledgement to the previous command. No payload

No Acknowledgement(0xFF)

No Ack to the previous command. No payload.

3.2 Device Simulator

The device simulator is the part of the framework that fulfils the purpose of simulating the device. It is responsible for performing the operations requested by the host application, such as memory access and kernel execution. The device simulator also emulates scheduling of work items across available Compute Units. This section will explain the different parts of the device simulator.

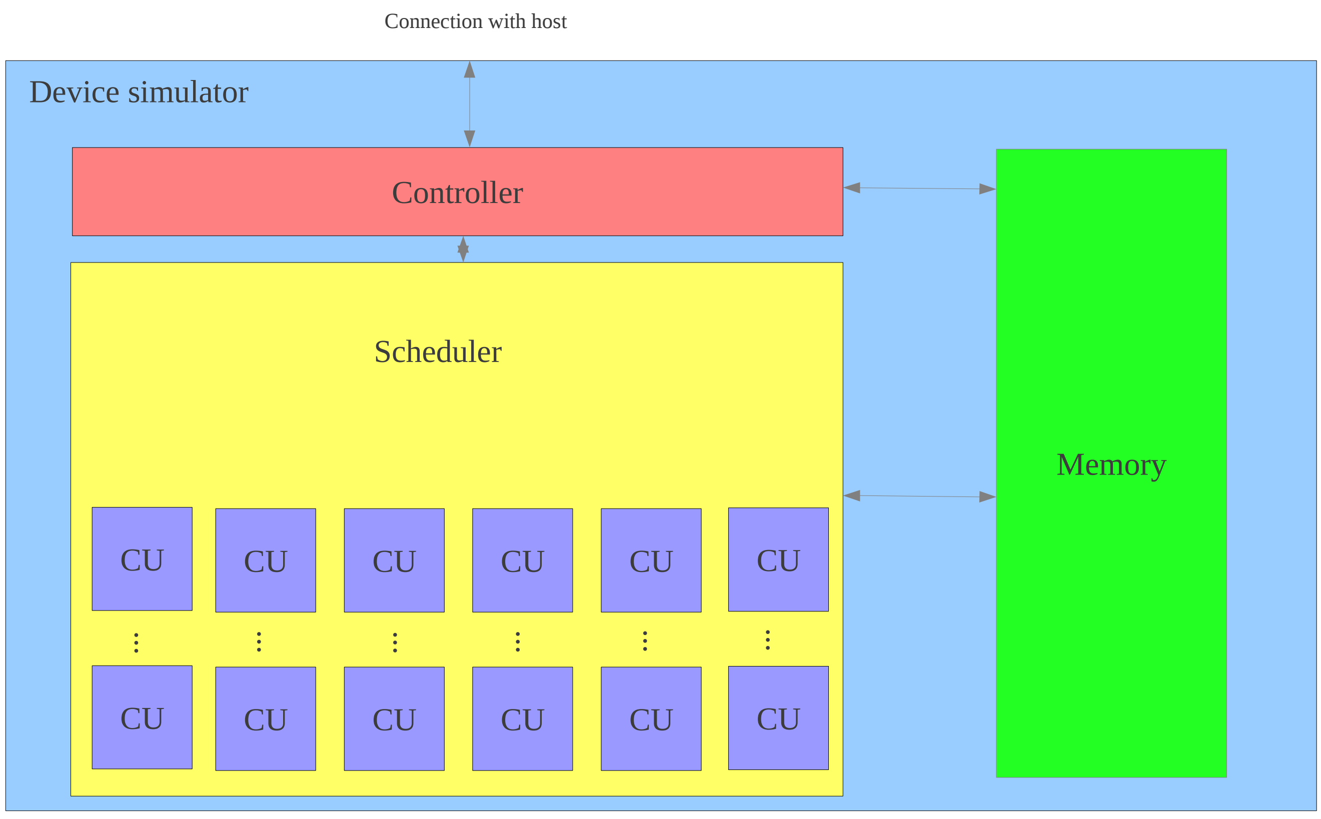


Figure 1 – Internal structure of the device application.

3.2.1 Memory

The device maintains a global region of memory that is shared by all Compute Units. It serves as the memory of the simulated device. This piece of memory is allocated at start-up.

3.2.2 Controller

The controller allows the host to control the device. The controller object maintains the TCP connection with the host runtime and processes and responds to commands from the host. The port can be specified in the device simulator's first command line arguments. The controller performs memory access operations and is responsible for initiating work on the scheduler. The controller object is instantiated at start-up.

3.2.3 Scheduler

The scheduler is the part of the device simulator that dispatches work to Compute Units (CU) according to the scheduling algorithm. The device simulator is structured in a way that allows different schedulers to be swapped in easily, as long as schedules implement the scheduler interface:

```
class IScheduler{
public:
    IScheduler();
    virtual void addWork(int globalWS[3]) = 0;
    virtual void CUDone(ComputeUnit *free_cu) = 0;
    virtual ~IScheduler() = 0;
};
```

ISchedule() is the constructor. The construction is called at instantiation of the scheduler, initialisation of the scheduler occurs here.

addWork(int globalWS[3]) tells the scheduler to start executing the kernel with the dimensions given in *globalWS*.

*CUDone(ComputeUnit *free_cu)* is the method that a compute unit to notify the scheduler that it has finished processing its work item.

~*IScheduler()* is the destructor. This is called when the scheduler is deleted. It contains the scheduler's clean up operations.

The scheduling algorithm delivered maintains a number of CUs on two queues. At initialisation, all CUs are kept in the free queue. When the device simulator is instructed to start work, the scheduler dispatches tasks (each with a different Global ID) to any Compute Unit that are in the free CU queue. CUs that are dispatched are removed from the free queue. The scheduler continues to dispatch work until the free CU queue is empty. CUs that have completed their work item will call the scheduler's CUDone method with the reference to itself. CUDone places the finished CUs onto the done queue. If the free queue is empty, and there are more work items, the schedule checks the done CU queue until the done queue is not empty. CUs that are in the done queue are sanitised and place onto the free queue, ready for the next work item. As the kernel is dynamically loaded, the sanitisation of CUs involve closing the dynamic library handle. **Figure 2** illustrates the scheduler's logic.

3.2.4 Compute Unit

Each instance of the compute unit class represents a compute unit on a MORA device. The CU is instantiated with the reference to a parent 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, the thread calls the CUDone method of the CU's parent scheduler to let it know the CU is done with the work item.

As described in 3.2.3, the scheduler dispatches as many CUs with work as possible at all times. Synchronisation between CUs and the scheduler thread is done using a single mutex lock. Access to either the free or done queue must be done after the mutex lock has been acquired.

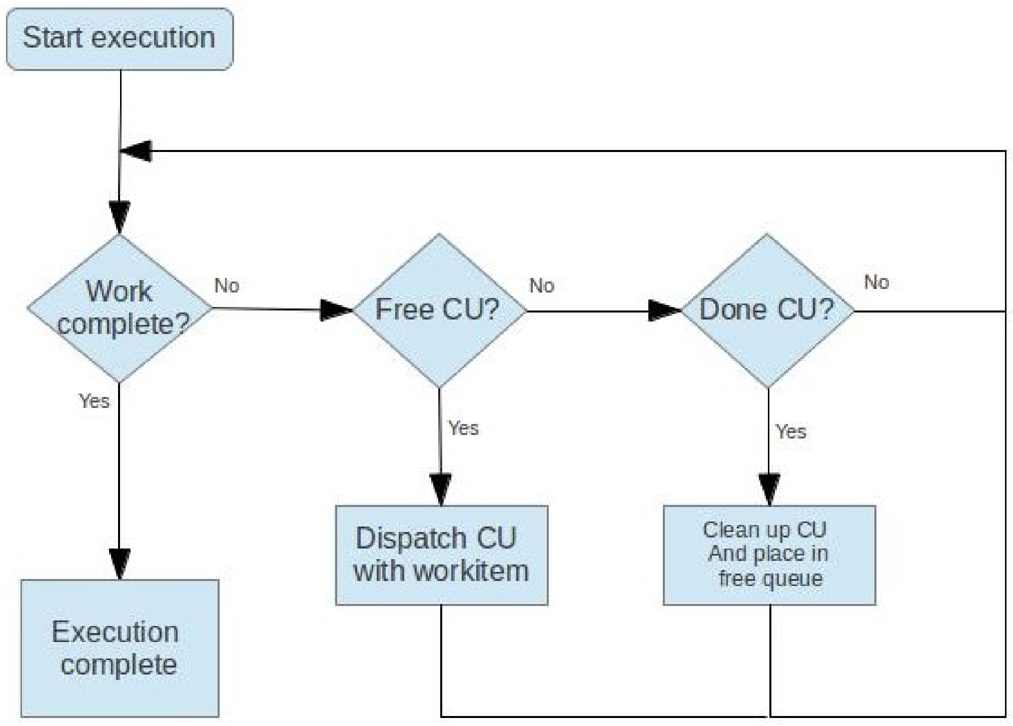


Figure 2 **–** Scheduler logic

3.3 Host OpenCL runtime

The host OpenCL runtime is a shared library that implements the OpenCL API. An application wishing to dispatch computation to a client device would link against this library at runtime. Before computation can begin, the host application must create a number of OpenCL objects.

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.

3.3.1 Creating platform and context

An OpenCL runtime may support multiple platforms. Each platform can have zero or more devices. The application must interrogate the OpenCL runtime to learn about the platform and device statuses. 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. In this implementation, the connection between the host runtime and the device is made during context object instantiation.

3.3.2 Creating command queue

The command queue will hold any operations enqueued by the host application to be executed on the device. Section 3.3.8 provides a detailed description of this implementation's command queue mechanics.

3.3.3 Creating OpenCL program

At this stage, the host application needs to create a program, either from OpenCL source, or an existing binary. A program is a set of kernels, identified within the program with the __kernel qualifier.

Source code text is supplied to the implementation through the clCreateProgramWithSource call. An existing binary is supplied though clCreateProgramWithBinary call.

This implementation only supports only one kernel per program, but the distinction is still made, as a program represents the entire set of computation and a kernel is the program with a specific function name that is the kernel function.

3.3.4 Creating memory(buffer) objects

Memory objects created at the host by the host application correspond to regions memory allocated at the device. 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.

3.3.5 Kernel compilation

Kernel compilation in our implementation is two stage process. First the program is built by the call clBuildProgram. Then the program binary is wrapped with the kernel header and wrapper function.

Although this implementation supports only one kernel per program, spliting the kernel compilation process into building program and compiling kernel allows the host application to retrieve the program binary without having to create a kernel object, which is allowed by the OpenCL standard, and is used by applications to shorten future build time by supplying binary built previously instead of program source.

3.3.5.1 Building the program

The program is compiled at clBuildProgram call into an object file. The build options are passed on to the build process writing them into a specific file. The build script buildProgram.sh wraps the OpenCL source code with the kernel header, then compiles it into the object file program.o using GCC, with build options from tmp.options. 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.

3.3.5.2 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.

3.3.6 Start of computation

The *NDRangeKernel* command from the application is a request to start execution of a given kernel at the device. 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 through the host-device communication link. When the kernel transfer is complete, the host signals the device to start executing the kernel. When 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.

To reduce bandwidth use and latency in starting execution, this implementation checks if the program has changed since the last *NDRangeKernel* command. If the program has not changed, the kernel will not be transferred again.

3.3.7 Result read back

Once kernel execution is complete, the application would typically want to retrieve the result of the computation. The application typically does this by enqueueing a memory read command after the *NDRangeKernel* command. The read command should be enqueued after a barrier command or after calling clfinish to ensure kernel execution has completed before the read is carried out.

3.3.8 Command queues

A command queue is an object for queueing operations to an OpenCL device. In general, operations that the OpenCL host application dispatches to the device are queued to be carried out at the device's pace, rather than to be executed and returned 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, and is a potential future improvement.

The command queue allows the meaningful implementation of synchronisation mechanisms such as the clFinish and clFlush calls, as well as emulated support for barrier command objects. According to the OpenCL standard, a barrier object is a synchronisation point. Events after a barrier command must not be dispatched until the barrier command has been dispatched. In an implementation where out-of-order execution is supported, the barrier command ensures that the previous set of commands will not interfere with the next set of commands. In our implementation however, the barrier command is not meaningful, since out-of-order execution of commands is not supported. The barrier command is still queued and will progress through the command queue as it should.

3.3.8.1 Implementation details of the command queue

Each command queue object creates its own worker thread upon instantiation.

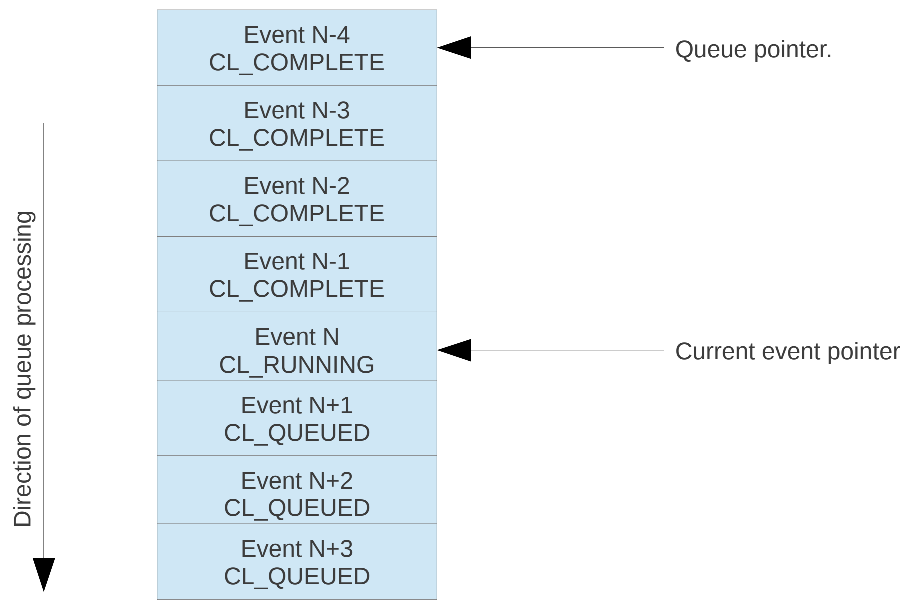


***Figure 3** – Command queue*

The command queue was implemented using single linked-list of command objects (also known as events). When the application calls an API function that enqueues an operation, a command object is created to hold the necessary information about the enqueued 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 approach would be that the head of the linked list will always 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 that waits 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 this 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 4 provides a simple diagrammatic illustration of the command queue mechanism.

3.3.8.2 Command queue logic

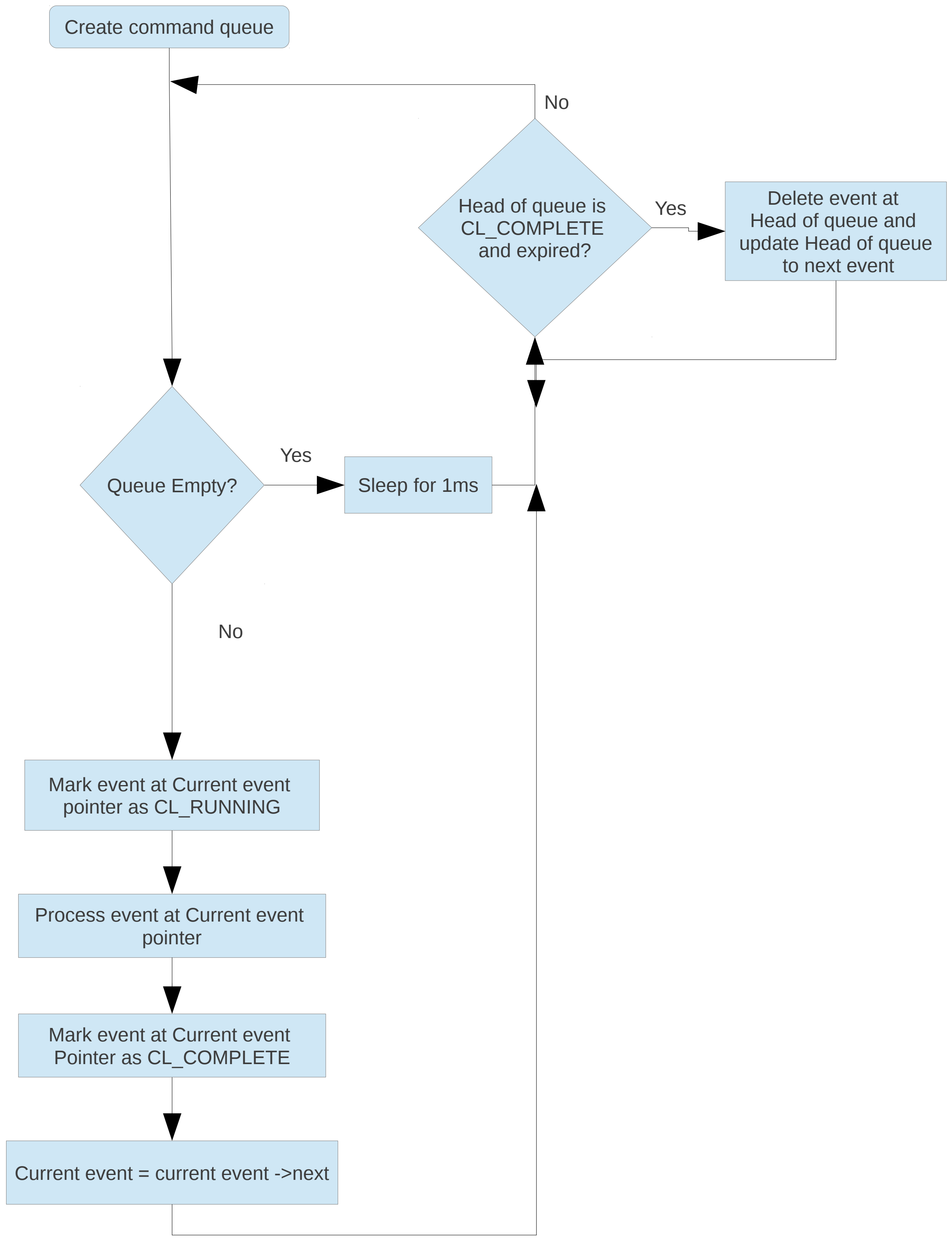

Create command queue
No
Head of queue is CL_COMPLETE and expired?
Yes
Delete event at Head of queue and update Head of queue to next event
Queue Empty?
Yes
Sleep for 1ms
No
Mark event at Current event pointer as CL_RUNNING
Process event at Current event pointer
Mark event at Current event Pointer as CL_COMPLETE
Current event = current event ->next

***Figure 4** – Command queue logic*

3.3.9 Debugging process

During the implementation process, problems were diagnosed largely through the use of debug messages and the use of GNU Debugger. When problems are discovered, the order in which the debug messages are logged usually points out where the problem might lie. For non-trivial crashes, GNU Debugger is helpful in allowing the state of application at the time of the crash to be inspected.

In trying to get Cgminer operating with the framework, a lot of the missing features were flagged up and subsequently implemented. Running Cgminer over an extended period of time also helped to validate the stability of the framework.

3.4 Deliverable File Tree

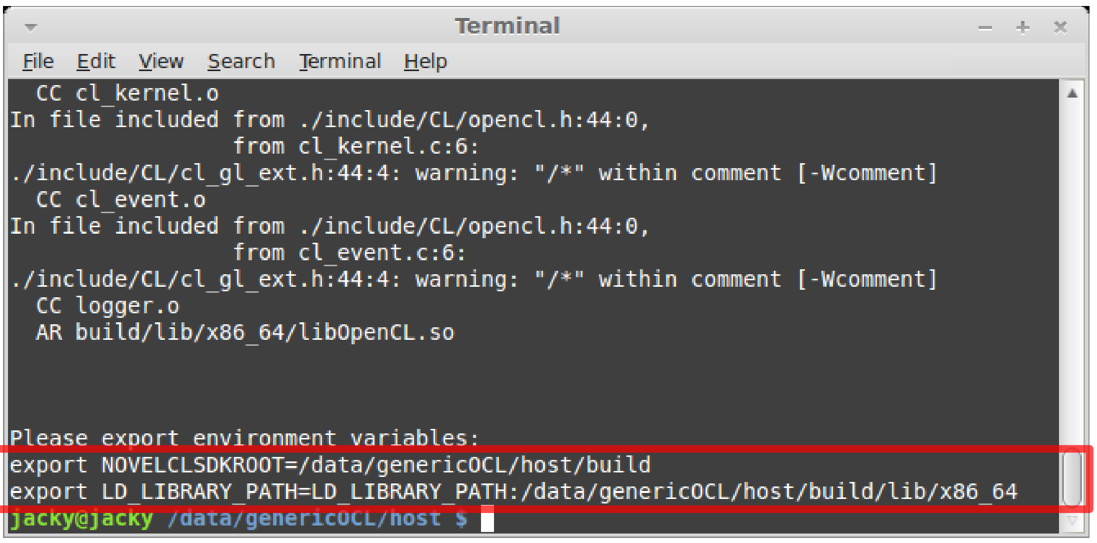
```
|-- demos                          Example programs
|   |-- addition                   Addition example
|   |   |-- addition.cc            Addition example host application
|   |   |-- int_sum.cl             Addition example OpenCL source
|   |   |-- Makefile               Addition example Makefile
|   |   `-- ocl.h                  Definitions
|   |-- cgminer                    Cgminer
|   |   | …...                     Cgminer source as cloned from official repository, not shown here.
|   |-- helloworld                 Hello world example
|   |   |-- helloworld.cc          Hello world example host application
|   |   |-- hello_world.cl         Hello world example OpenCL source
|   |   |-- Makefile               Hello world example Makefile
|   |   `-- ocl.h                  Definitions
|   `-- matrix                     Matrix multiplication example
|   |   |-- Makefile               Matrix multiplication example Makefile
|   |   |-- matrix.cc              Matrix multiplication example  host application
|   |   |-- matrix_mul.cl          Matrix multiplication OpenCL source
|       `-- ocl.h                  Definition
|   |-- Makefile                   Makefile for example programs.
|-- device                         Device simulator directory
|   |-- ComputeUnit.cpp            Compute Unit source
|   |-- ComputeUnit.hpp            Compute Unit header
|   |-- ControlLink.cpp            Controller source
|   |-- ControlLink.hpp            Controller header
|   |-- core.cpp                   Device simulator top level
|   |-- debug.h                    Debug macro definition
|   |-- Device.cpp                 Device simulator class
|   |-- Device.hpp                 Device simulator header
|   |-- GlobalDef.hpp              Global definitions
|   |-- Ischeduler.hpp             Schedule Interface definition
|   |-- Makefile                   Device simulator Makefile
|   |-- SocketConnector.cpp        Connection manager
|   |-- SocketConnector.hpp        Connection manager header
|   |-- TPScheduler.cpp            Thread pool Scheduler
|   `-- TPScheduler.hpp            Thread pool Scheduler header
|-- deviceProxy                    Device simulator proxy program
|   |-- Makefile                   Device simulator proxy Makefile
|   `-- simpleProxy.c              Device simulator proxy program source
|-- host                           Host runtime implementation directory
|   |-- cl_context.c               OpenCL Context implementation
|   |-- cl_cqueue.c             OpenCL Command Queue implementation
|   |-- cl_defs.h               Internal definition of OpenCL data structures
|   |-- cl_device.c             OpenCL device object implementation
|   |-- cl_event.c              OpenCL event object implementation
|   |-- cl_kernel.c             OpenCL kernel object implementation
|   |-- cl_mem.c                OpenCL memory object implementation
|   |-- cl_platform.c           OpenCL platform implementation
|   |-- cl_program.c            OpenCL program object implementation
|   |-- debug.h                 Debug macro definition
|   |-- dev_interface.c         Device communications code
|   |-- dev_interface.h         Device communications definitions
|   |-- include                 Header files needed by the host runtime and applications
|   |   |-- CL                  OpenCL definitions from Khronos
|   |   |   |-- cl_ext.h
|   |   |   |-- cl_gl_ext.h
|   |   |   |-- cl_gl.h
|   |   |   |-- cl.h
|   |   |   |-- cl.hpp
|   |   |   |-- cl_platform.h
|   |   |   `-- opencl.h
|   |   `-- kernel.h            Kernel header, needed for kernel compilation
|   |-- logger.c                Debug logger. Logs debug messages to /tmp/logfile.txt
|   |-- Makefile                Host runtime Makefile
|   `-- scripts                 Scripts for wrapping and building kernel.
|       |-- buildprogram.sh     Program compiler script
|       |-- compilekernel.sh    Kernel linking script
|       `-- createwrapper.py    Kernel wrapper script
|-- Makefile                    Top-level Makefile
|-- README                      README
`-- README.md                   Same as README, read by Github.
```

4 Evaluation

This section attempts to demonstrate that the deliverables function correctly. First, the three simple demonstration applications supplied in the starting material are used exercise the host runtime and the device simulator, and the results are included. Then an existing OpenCL application is used to demonstrate that the deliverables work beyond the examples, and to provide and indication of the device simulator's performance.

4.1 Running examples

The device, host runtime and the demos (except for cgminer) can all be compiled using the top-level Makefile. In line with conventions adopted by other OpenCL implementations to co-exist on the same system, an environment variable is used to specify the path to the libOpenCL.so shared library, OpenCL header files and any other supporting files. The variable used by this implementation is “*NOVELCLSDKROOT*”. The end of the host runtime compilation suggests the value $*NOVELCLSDKROOT* should have on the given build. See *Screenshot 1*.



```
  CC cl_kernel.o
In file included from ./include/CL/opencl.h:44:0,
                 from cl_kernel.c:6:
./include/CL/cl_gl_ext.h:44:4: warning: "/*" within comment [-Wcomment]
  CC cl_event.o
In file included from ./include/CL/opencl.h:44:0,
                 from cl_event.c:6:
./include/CL/cl_gl_ext.h:44:4: warning: "/*" within comment [-Wcomment]
  CC logger.o
  AR build/lib/x86_64/libOpenCL.so



Please export environment variables:
export NOVELCLSDKROOT=/data/genericOCL/host/build
export LD_LIBRARY_PATH=LD_LIBRARY_PATH:/data/genericOCL/host/build/lib/x86_64
jacky@jacky /data/genericOCL/host $
```

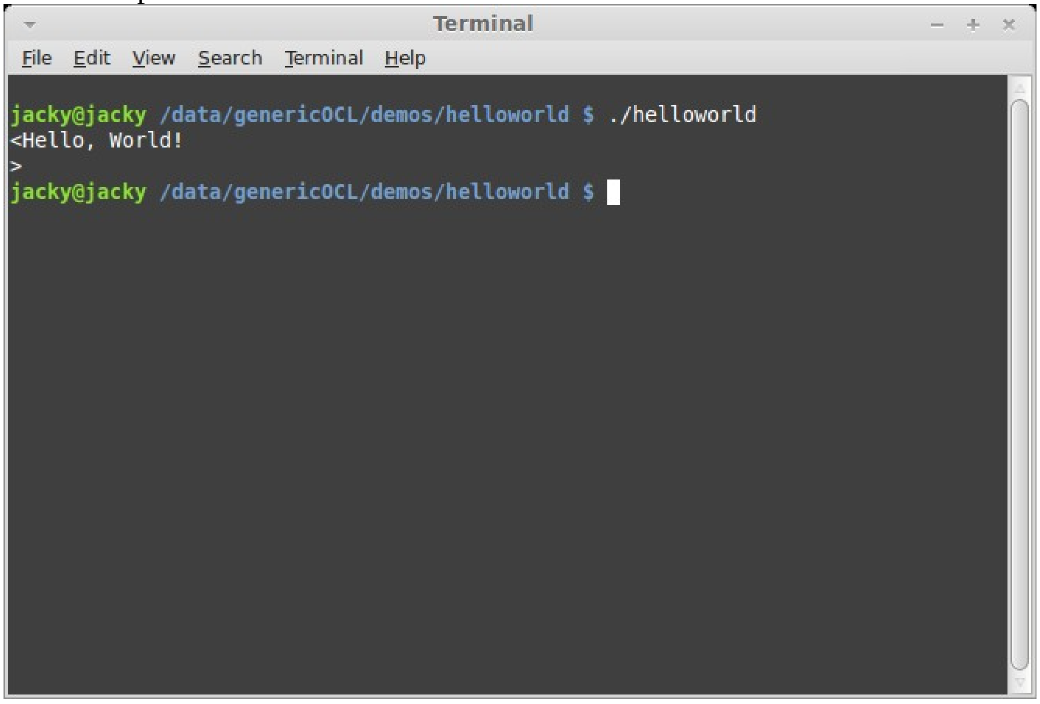
***Screenshot 1**: End of host runtime compilation*

The demos must be built after the host runtime. The three demo applications provided in the starting material, helloworld, addition and matrix, are included to demostrate this implementation. As they are located within the same build system as the host runtime, for ease of compilation the location to the host runtime library is hard-coded in the applications' Makefiles.

4.1.1 helloworld

The helloworld application executes a kernel that writes the string “Hello World!\n” to the device's memory, the host application then reads the device memory and print it out between the characters '<' and '>'.

Host application output:



```
jacky@jacky /data/genericOCL/demos/helloworld $ ./helloworld
<Hello, World!
>
jacky@jacky /data/genericOCL/demos/helloworld $
```

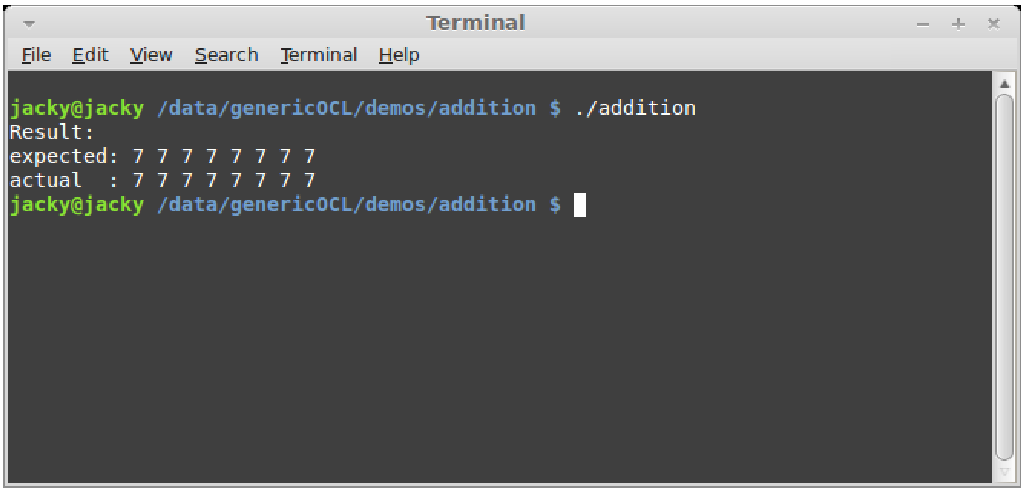
***Screenshot 2**: Hello world example application output*

4.1.2 addition

The addition application adds 2 vectors of 8 elements. The first vector is initialised to [0,1,2,3,4,5,6,7], the second vector is initialised to [7,6,5,4,3,2,1,0]. The addition is computed at the host, and at the device by the OpenCL kernel. The result is read back from the device and printed out. The format of the output is:

```
Result:
expected: <result computed at the host>
actual: <result read back from the device after kernel execution>
```

Host application output:



```
jacky@jacky /data/genericOCL/demos/addition $ ./addition
Result:
expected: 7 7 7 7 7 7 7 7
actual  : 7 7 7 7 7 7 7 7
jacky@jacky /data/genericOCL/demos/addition $ 
```

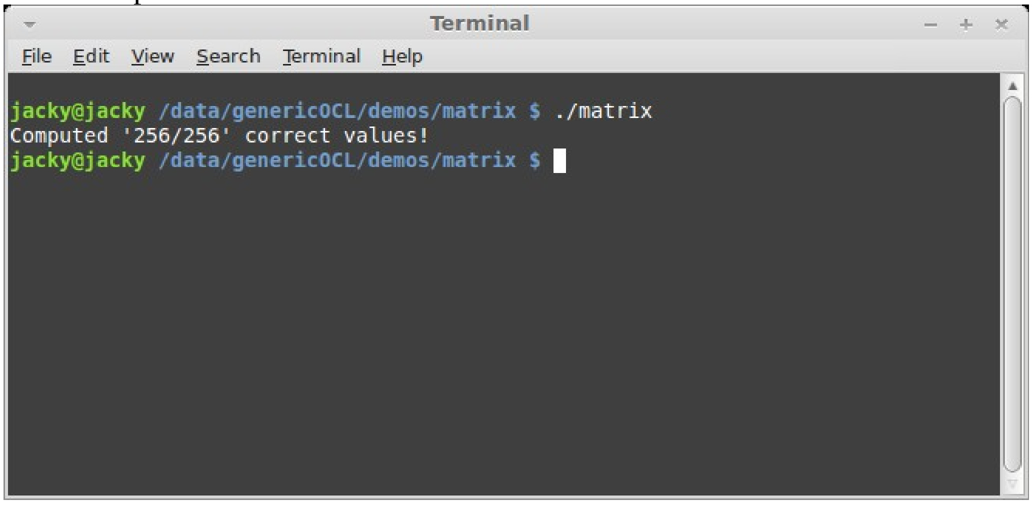
***Screenshot 3**: Array addition example application output*

4.1.3 matrix

The matrix application performs a multiplication of 2 1-by-16 matrices, which results in a 1-by-256 result. Similar to the addition example, the matrix application computes the problem both on the host and on the device, then compare the host's result with the result read back from the device.
The format of the output is:

```
[diff x if element x is unexpected]
                    :
                    :
Computed '<correct>/256' correct values!
```

Host application output:



```
jacky@jacky /data/genericOCL/demos/matrix $ ./matrix
Computed '256/256' correct values!
jacky@jacky /data/genericOCL/demos/matrix $
```

***Screenshot 4**: Matrix multiplication example application output*

4.2 Real world application demo

To demostate the usability of this implementation by existing real world application, a GPU bitcoin mining is used. The application called Cgminer by Con Kolivas. [18] It is one of the most popular bitcoin mining software that utilises GPGPUs.

4.2.1 Bitcoin

Bitcoin is a peer-to-peer decentralised digital currency, maintained only by its participants. This section attempts to give a brief explaination on bitcoins and the relevance of massively parallel processing in bitcoin creation. For more detailed explaination, please see [19].

The bitcoin network contains a public ledger of all transactions ever carried out. This public ledger is called the block chain.
The block chain is made up of blocks of transactions. Each successive block is crytographically related to the previous block in the block chain, and each transaction in a block is cryptographically related to the block.
For a transaction to count, a bitcoin payer must broadcast the transaction and wait for it to be included in the block chain. Transactions that have been broadcasted but have not yet been included in the block chain are called *floating transactions*. Maintaining participants of the bitcoin network (known as *miners*) gather the floating transactions and try to cryptographically incorporate them into the next block. Miners are incentivised with newly created bitcoins by the network to process transactions. The creator of a block is allowed to include one extra transaction into the block, which pays bitcoins to the creator out of nowhere.

The supply of bitcoins is governed by an algorithm, which is mutually agreed upon by the participants of the bitcoin network. Blocks will only be incorporated into the block chain if the majority of the participants agree that those blocks are valid according to the algorithm. The algorithm dictates the amount of new bitcoins created through each block and maintains the average amount of time each block to be 10 minutes, by varying the *network difficulty*. The cryptographic solution that makes a block valid is called the *proof of work*. The proof of work is designed to be computationally difficult. The higher the network difficulty, the more difficult it is to create.

To produce the proof of work, the miner first creates a merkle tree of the SHA-256 hash of all the transactions to be included in the block. The miner then appends a nounce to the merkle root with a nounce and hash it twice with the SHA-256 hash function(known as SHA-256D). The result of the SHA-256D (aka *block hash*) is compared to the *target*, which is a number derived from the *difficulty*. If the block hash has more leading binary zeroes than the target, then the miner has found the nounce required to make the block valid. The miner will continue to SHA-256D hash the merkle root + nounce combination, changing the nounce every time, until the hash result is less than the target.

Summary of mining process:

1. Add reward transaction to the block
2. Add floating transactions to the block
3. Create merkle tree from the transactions and the hash of the previous block.
4. Append nounce to merkle root and double SHA-256 hash the result.
5. Compare the hash with the target, if hash has more binary leading zeroes, broadcast the block, else increment nounce and repeat step 4.

If a new block is detected from another miner during the mining process, all miners must start again, which updated floating transactions. The desired nounce is different for each miner, as the reward transaction is different.

Described in another way, mining basically involves finding a number that when appended to a constant and hashed twice with SHA-256, gives a result that is numerically less that the target. This is a bruteforce problem, and can easily be parallelised. Cgminer uses GPGPUs to perform the double hashing required to produce the valid proof of work.

At the time of writing, the difficulty of producing a valid block stands at 65.8 million. At this level of diffculty, it is not feasible for an individual miner to mine a block on its own (known as solo mining). Instead miners often join a mining pool server, which divides the nounce search space to participating miners. (known as pool mining).

4.2.2 Build instructions

A version of cgminer is included in the deliverable at demoes/cgminer. It was taken from cgminer's official repository. No modifications were made to the source, except to add *NOVELCLSDKROOT* to the search path for OpenCL runtime, into the autoconf config file.

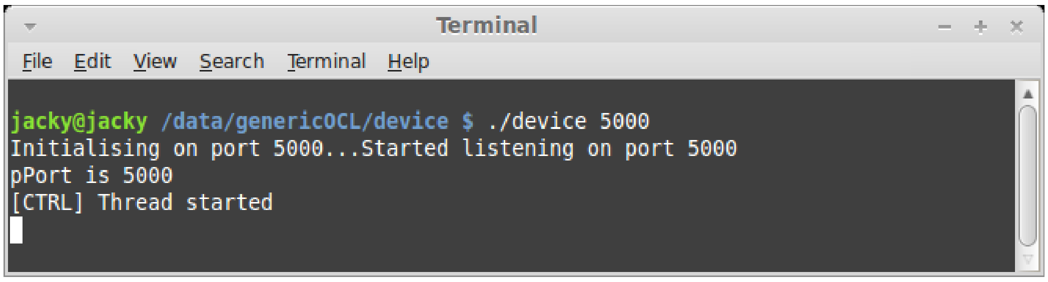
To build cgminer, first make sure that the *NOVELCLSDKROOT* environment variable is set, then go to cgminer's directory and run:

```
./autogen.sh
make
```

After this, cgminer should be compiled.

4.2.3 Running instructions

Before running cgminer, make sure the device simulator is running and listening to local port 5000.



```
jacky@jacky /data/genericOCL/device $ ./device 5000
Initialising on port 5000...Started listening on port 5000
pPort is 5000
[CTRL] Thread started
```

***Screenshot 5**: Device simulator running*

To start cgminer:

```
./cgminer -O<Your mining server credential> -o<Your mining server url> -v1 -I6
```

The -v1 option tells cgminer to use vector size of 1. Vectors of sizes 2, 4, 8 or 16 are not supported by the device simulator.
The -I option specifices the *intensity*. The intensity setting is related to the workgroup size of each

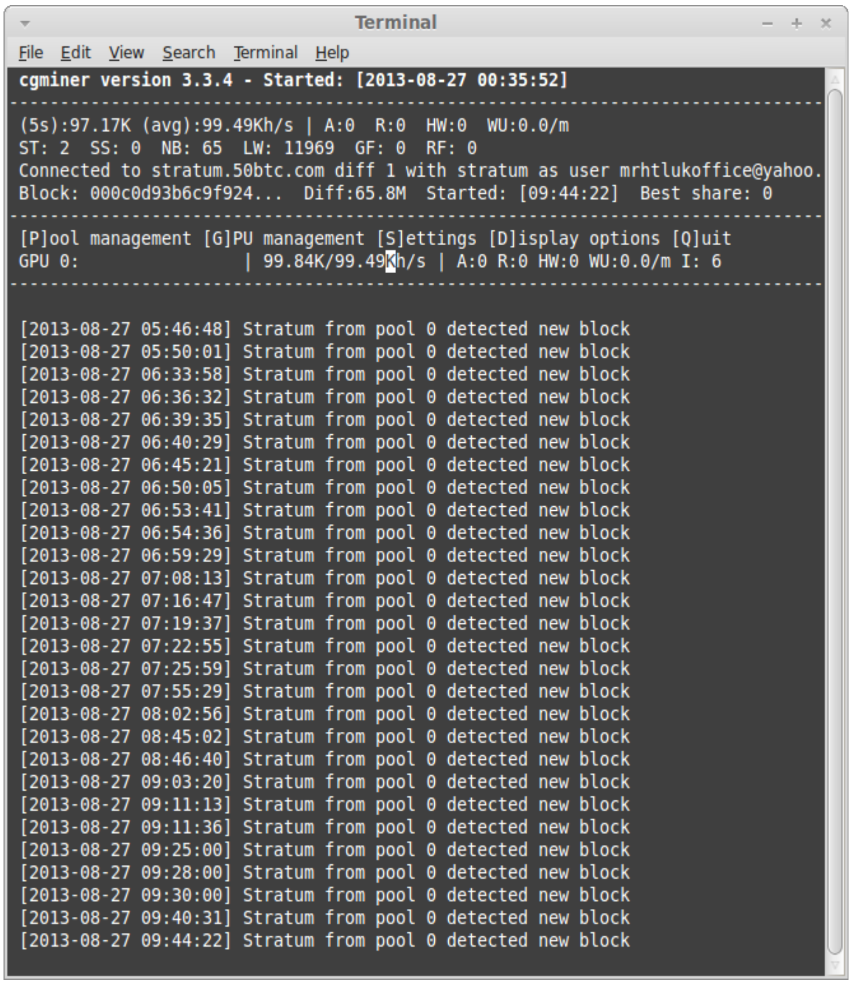
iteration. The lower the intensity the smaller the workgroup size. This means execution will return more quickly, but host-device communications overhead becomes more significant. Conversely, higher intensity means larger workgroup sizes, longer kernel execution time, and longer time between solution being found by a work item and the solution being read back by the host application after kernel execution. An intensity value of 6 is about right on the development machine, with device simulator spending roughly 95% of the time executing the kernel and 5% of the time communicating with the host.

4.2.4 Performance

Performance in bitcoin mining is measured in double-hashes per second (KH/s). Cgminer achieved between 90KH/s and 100KH/s on our framework running on an Intel i7 3770k 4 core (8 threads) multi processor at 3.50GHz clock frequency. The smallest work obtainable from a mining server has a difficulty of 1. This translates to an average of 1 solution every 2^{32} hashes. At this rate, a share can be found roughly every 11.93 hours. Unfortunately no solutions have been found during the trial runs.

In comparison, cgminer running on the same hardware under Windows 7 directly with CPU mining code (without using OpenCL on CPU) can achieve ~55MH/s.

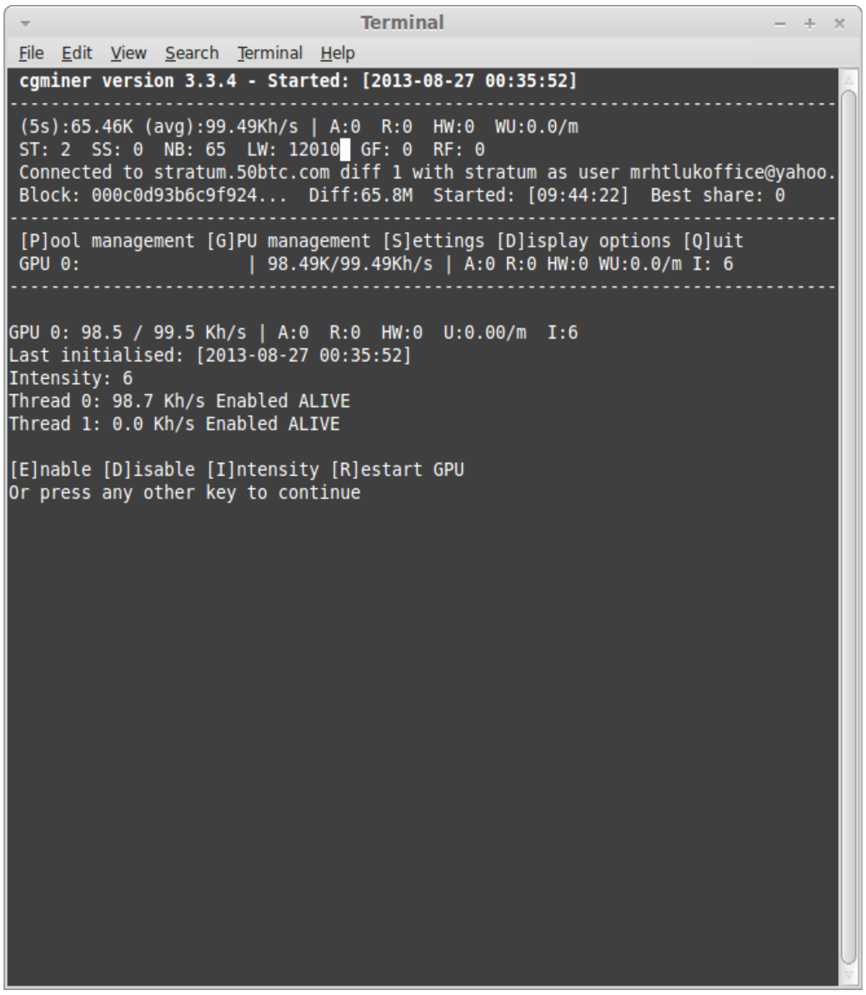
The lower performance is likely due to the overhead in compute unit threads starting and stopping, and the dynamic linking of kernel in the device simulator. The design simulator is designed to emulate scheduling of work among Compute Units, rather than for performance in kernel execution.



```
cgminer version 3.3.4 - Started: [2013-08-27 00:35:52]
--------------------------------------------------------------------------------
(5s):97.17K (avg):99.49Kh/s | A:0  R:0  HW:0  WU:0.0/m
ST: 2  SS: 0  NB: 65  LW: 11969  GF: 0  RF: 0
Connected to stratum.50btc.com diff 1 with stratum as user mrhtlukoffice@yahoo.
Block: 000c0d93b6c9f924...  Diff:65.8M  Started: [09:44:22]  Best share: 0
--------------------------------------------------------------------------------
[P]ool management [G]PU management [S]ettings [D]isplay options [Q]uit
GPU 0:                | 99.84K/99.49Kh/s | A:0 R:0 HW:0 WU:0.0/m I: 6
--------------------------------------------------------------------------------


[2013-08-27 05:46:48] Stratum from pool 0 detected new block
[2013-08-27 05:50:01] Stratum from pool 0 detected new block
[2013-08-27 06:33:58] Stratum from pool 0 detected new block
[2013-08-27 06:36:32] Stratum from pool 0 detected new block
[2013-08-27 06:39:35] Stratum from pool 0 detected new block
[2013-08-27 06:40:29] Stratum from pool 0 detected new block
[2013-08-27 06:45:21] Stratum from pool 0 detected new block
[2013-08-27 06:50:05] Stratum from pool 0 detected new block
[2013-08-27 06:53:41] Stratum from pool 0 detected new block
[2013-08-27 06:54:36] Stratum from pool 0 detected new block
[2013-08-27 06:59:29] Stratum from pool 0 detected new block
[2013-08-27 07:08:13] Stratum from pool 0 detected new block
[2013-08-27 07:16:47] Stratum from pool 0 detected new block
[2013-08-27 07:19:37] Stratum from pool 0 detected new block
[2013-08-27 07:22:55] Stratum from pool 0 detected new block
[2013-08-27 07:25:59] Stratum from pool 0 detected new block
[2013-08-27 07:55:29] Stratum from pool 0 detected new block
[2013-08-27 08:02:56] Stratum from pool 0 detected new block
[2013-08-27 08:45:02] Stratum from pool 0 detected new block
[2013-08-27 08:46:40] Stratum from pool 0 detected new block
[2013-08-27 09:03:20] Stratum from pool 0 detected new block
[2013-08-27 09:11:13] Stratum from pool 0 detected new block
[2013-08-27 09:11:36] Stratum from pool 0 detected new block
[2013-08-27 09:25:00] Stratum from pool 0 detected new block
[2013-08-27 09:28:00] Stratum from pool 0 detected new block
[2013-08-27 09:30:00] Stratum from pool 0 detected new block
[2013-08-27 09:40:31] Stratum from pool 0 detected new block
[2013-08-27 09:44:22] Stratum from pool 0 detected new block
```

Screenshot 6*: Cgminer performing bitcoin mining with the host runtime*



```
 cgminer version 3.3.4 - Started: [2013-08-27 00:35:52]
--------------------------------------------------------------------------------
 (5s):65.46K (avg):99.49Kh/s | A:0  R:0  HW:0  WU:0.0/m
 ST: 2  SS: 0  NB: 65  LW: 12010  GF: 0  RF: 0
 Connected to stratum.50btc.com diff 1 with stratum as user mrhtlukoffice@yahoo.
 Block: 000c0d93b6c9f924...  Diff:65.8M  Started: [09:44:22]  Best share: 0
--------------------------------------------------------------------------------
 [P]ool management [G]PU management [S]ettings [D]isplay options [Q]uit
 GPU 0:                | 98.49K/99.49Kh/s | A:0 R:0 HW:0 WU:0.0/m I: 6
--------------------------------------------------------------------------------


GPU 0: 98.5 / 99.5 Kh/s | A:0  R:0  HW:0  U:0.00/m  I:6
Last initialised: [2013-08-27 00:35:52]
Intensity: 6
Thread 0: 98.7 Kh/s Enabled ALIVE
Thread 1: 0.0 Kh/s Enabled ALIVE

[E]nable [D]isable [I]ntensity [R]estart GPU
Or press any other key to continue
```

***Screenshot** 7: Cgminer GPU management page*

4.3 Future improvements

The following list shall be considered as possible improvements.

- 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 start-up and treat it like the global memory for the device.

4.4 Other Potential applications

The generic nature of this implementation lends itself to some interesting potential applications. An intriguing idea is to use the host runtime as a wrapper to other proprietary OpenCL implementations. Such a wrapper would allow a pool of OpenCL enabled computing resources by different vendors to be managed under a single OpenCL runtime. This would perhaps be useful in commoditising massively parallel computing resources, e.g. GPU farms.

5 Conclusion

This project has delivered a generic framework for dispatching computation to a client device and scheduling a device simulator. The framework allows applications using OpenCL standard APIs to dispatch computational work through the framework to the device client. The device simulator simulates work items scheduling amongst Compute Units.

Three demonstration programs and a real world application have been included in the deliverable, and have been used to demonstrate the functional correctness and robustness of the host-runtime implementation, the device simulator, and the communications protocol between them.

It is hoped that the device simulator will be useful in the simulation of a MORA device on a general purpose mainframe.

6 References

[1]Computer Architecture, A Quantitative approach, 4th Edition
by John L. Hennessy and David A. Patterson
[1.1]Page 17 – Scaling of Transistor Performance and wires
[1.2]Appendix A, Pipelining: Basic and Intermedia Concepts
[1.3]Page 115 – Exploiting ILP Using Multiple Issue and Static Scheduling

[2] The Story of the Intel 4004, Intel
http://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html

[3] Datasheet, Intel 4004 Single Chip 4-bit P-Channel Microprocessor,
http://www.intel.com/Assets/PDF/DataSheet/4004_datasheet.pdf

[4] Intel® Pentium® 4 Processor 661 supporting HT Technology
http://ark.intel.com/products/27485/Intel-Pentium-4-Processor-661-supporting-HT-Technology-2M-Cache-3_60-GHz-800-MHz-FSB

[5] A Single Atom Transistor
http://www.nature.com/nnano/journal/v7/n4/full/nnano.2012.21.html

[6] Intel 3rd generation Core processor
http://semiaccurate.com/2012/04/23/intel-launches-ivy-bridge-amid-crushing-marketing-buzzwords/

[7] Intel 4th generation i7
http://newsroom.intel.com/community/intel_newsroom/blog/2013/04/26/chip-shot-4th-generation-intel-core-coming-soon/

[8] Pentium 4 CPU Overclocked to 8GHz!, Mark Whiting
http://www.1up.com/news/pentium-4-cpu-overclocked-8ghz

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

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

[11] An OpenCL API for high-level FPGA programming, Marcin Bujar

[12] Branching(revision control), Wikipedia
http://en.wikipedia.org/wiki/Branching_(revision_control)

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

[14] Intel SDK for OpenCL
http://software.intel.com/en-us/vcsource/tools/opencl-sdk

[15] AMD OpenCL Zone. AMD APP SDK. For AMD processors and ATI GPGPUs.
http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/

[16] Nvidia OpenCL Support
https://developer.nvidia.com/opencl

[17] Simple TCP Proxy, Duane Wessel
http://www.life-gone-hazy.com/src/simple-tcp-proxy/

[18] CGMiner GPU bitcoin miner
https://en.bitcoin.it/wiki/CGMiner

[19] Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto
http://www.bitcoin.org/bitcoin.pdf