Investigation into hardware acceleration of HPC kernels, within a cross platform OpenCL environment

Stuart Fraser

Exam No: 7363547

August, 2010

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2010
Abstract

Diversification in standards and interfaces utilised in hardware accelerators for HPC computations has force the creation of a single standard compute language, OpenCL. Utilising heterogeneous environments we tested the feasibility of using OpenCL and managed object oriented languages to provide a performant, portable platform for the implementation of computationally expensive codes. Specifically a LU Factorisation kernel was ported and then re-written in a manner suitable for a GPGPU device. The initial port demonstrated poor performance compared to the CPU but second, parallel based code showed a just over twice the performance. To achieve performance gains a balance between the cost of execution and complexity of an architecture specific solution must be considered.
Contents

1 Introduction .................................................................................................................. 1

2 Background .................................................................................................................. 3
  2.1 Accelerators ......................................................................................................... 3
  2.2 Drive to portability .............................................................................................. 4
  2.3 HPC kernels ......................................................................................................... 5

3 OpenCL ......................................................................................................................... 6
  3.1 Framework .......................................................................................................... 7
  3.2 Platform .............................................................................................................. 8
    3.2.1 Generic Device ............................................................................................. 9
    3.2.2 Generic Host ............................................................................................... 10
  3.3 Software Architecture ......................................................................................... 15
    3.3.1 Host Code ................................................................................................... 15
    3.3.2 Device Code .............................................................................................. 16
  3.4 Execution Model ................................................................................................... 18

4 Common Language Runtime ...................................................................................... 20
  4.1 CLR Programmer Support ..................................................................................... 21
  4.2 Mono CLR ........................................................................................................... 21

5 Methodology ............................................................................................................... 23
  5.1 CPU Execution .................................................................................................... 24
    5.1.1 cpuBenchmarks ......................................................................................... 24
  5.2 OpenCL Execution ............................................................................................... 25
    5.2.1 OpenCL Libraries ....................................................................................... 25
5.2.2 oclEnumerate .......................................................................................26
5.2.3 oclHarness ..........................................................................................26
5.2.4 Kernel Porting .......................................................................................27
6 Results .............................................................................................................37
  6.1 Methods .....................................................................................................37
    6.1.1 Environment ......................................................................................37
  6.2 CPU Based Codes ....................................................................................39
  6.3 OpenCL Based Codes .............................................................................40
7 Project Work Plan ........................................................................................44
8 Conclusions ...................................................................................................45
9 Bibliography ..................................................................................................47
10 Appendix A – Original Project Plan ............................................................49
11 Appendix B – Raw data ................................................................................50
Table of Figures

Figure 2-1 – Manufacturer GPGPU performance metrics ................................................................. 3
Figure 3-1 – Framework objects ........................................................................................................ 7
Figure 3-2 – OpenCL Platform architecture ...................................................................................... 8
Figure 3-3 – Generic device layout .................................................................................................... 9
Figure 3-4 – Platform information ..................................................................................................... 10
Figure 3-5 – GPU Device Information ............................................................................................... 11
Figure 3-6 – CPU Device Information .............................................................................................. 12
Figure 3-7 – OpenCL Memory model ................................................................................................. 18
Figure 3-8 – OpenCL sample application flow .................................................................................. 19
Figure 5-1 – Mono debug output of native OpenCL library initialisation. ................................. 26
Figure 5-2 – Kernel definitions for dgefaCL and dgeslCL ................................................................. 28
Figure 5-3 – Main host code execution loop. .................................................................................... 28
Figure 5-4 – Profile of LUFact.cl for LU factorial n=250 ................................................................. 30
Figure 5-5 – Prototypes of kernels idamaxCLND, scaleCLND, elimCLND and solveCLND ................ 31
Figure 5-6 – Host code implementing dgesl using separate idamax, scale and eliminate kernels ............................................................................................................................ 32
Figure 5-7 – idamaxCLND parallel reduction model ........................................................................ 33
Figure 5-8 – Profile of LUFactPar.cl for LU Factorisation n=250 .................................................. 34
Figure 5-9 – Profile of LUFactPar.cl using dgefaCLND kernel ....................................................... 36
Figure 6-1 – Tests and problem size ................................................................................................. 37
Figure 6-2 – Test hardware .............................................................................................................. 38
Figure 6-3 – CPU execution of Java Grande Section 2 kernel codes, LU Factorial, Sparse Matrix Multiple and Fast Fourier Transformation ......................................................... 39
Figure 6-4 – Performance in MFlops of CPU and OpenCL codes n=250 ...............40
Figure 6-5 – Profile LUFactPar.cl C# host code execution.................................41
Figure 6-6 – Expansion of EnqueueNDRangeKernel execution profile ...............41
Figure 6-7 – Kernel execution profile of LUFactPar.cl........................................41
Figure 6-8 – Performance of C# implementation of dgefaCLND based LU Factorisation code for variable n. .................................................................42
Figure 6-9 – Graph of performance improvements as n increases in dgefaCLND based code.................................................................42
Figure 6-10 – Console output of kernel run............................................................43
Figure 6-11 – Profile output of dgefaCLND, n = 750 ............................................43
1 Introduction

Computers have always had hardware accelerators, performing an action on a device or circuit that is specifically designed to perform said function faster than would be possible by the central processor. These hardware accelerators have had many functions such as mathematics capabilities, graphics rendering, cryptography, multi-threaded execution etc. but many of these devices can be put other additional uses from which they were first designed, such as the current trend for General Purpose Graphics Processor Units (GPGPUs) for parallel mathematical computation. As is exemplified by the rapid take up in use of GPGPUs, this has led to disparate and incongruent software interfaces to program and control these devices.

There are many factors in choosing an accelerator, even within a class of device like a GPGPU such as performance, environmental concerns, device roadmap and advanced features. As in any competitive market the balance of these features, from any vendor, may vary and validate a specific solution for a period of time. Consideration must be given to any vendor specific lock which may cause unnecessary project costs and delays in porting to future hardware. Further, that even with a single hardware class such as GPGPU the major manufactures have had different programming interfaces again inhibiting code reuse and portability.

The two major GPU vendors NVidia and ATI have promoted CUDA\textsuperscript{[1]} and STREAM\textsuperscript{[2]} as the API of choice for their hardware until mid-2009 when the Khronos Group\textsuperscript{[3]} launched the Open Compute Language (OpenCL) standard\textsuperscript{[4]}. OpenCL is a common interface and approach to programming hardware accelerators whether they are GPGPU, Cell\textsuperscript{[5]}, FPGA\textsuperscript{[6]} or multicore CPU based.

With many vendors providing full support to the OpenCL programming standard it will be possible to independently choose the appropriate class and level of hardware device to accelerate the solution, without requiring major re-writes of complex algorithmic libraries. Further this would allow for upgrades in hardware capability, such as better power efficiencies, more performance or increased capabilities etc. without re-training and re-tooling the application developers for the new APIs.

Given that many HPC codes have been around for many years, longevity is an important consideration. Also the ability to have applications that can be easily transferred between platforms and interoperate external functions such as web service based information sources can further extend the reach and impact of many of these codes. There are a number of solutions to this but the most common is the use of a standard managed virtual execution environment such as Java\textsuperscript{[7]} Runtime or Common Language Runtime\textsuperscript{[8]} (CLR). Work has previously been performed by the Java Grande Forum\textsuperscript{[9]} on the suitability of these execution environments by comparing performance metrics of native C, FORTRAN and managed code\textsuperscript{[10]}. This has validated the use of managed runtime solutions but often showed that overhead of the virtual environment has had some detrimental effect.
Offloading computation to a device in common with each execution environment we look to demonstrate that performance increases through the use of accelerator hardware minimises the effects of managed runtime overheads. The benefit of this scenario is that it enables rapid development with high-level, object-oriented methods and languages without sacrificing computational performance.

The dissertation looks at a possible scenario for implementing platform independent, high performance applications using the modern high-level languages. We look to compare the execution of a fundamental numerical routine in a number of platforms, native and managed, both non hardware accelerated and with hardware acceleration.

Specifically three kernel based benchmarks were tested in a standard CPU based runtime and in an OpenCL GPGPU assisted environment. This involved a number of codes being written for host and GPU execution, some sections of code were ported while some original work was required for the GPU.

Given that a number of key technologies are nascent in nature, there is still flux in specification, driver compatibility and toolsets, where ever possible notes on the real world experiences have been included.

The structure of this report is as follows;

Chapter 2 lays the foundations for the core advancements that make such a goal possible.

Chapters 3 and 4 examine the specifics of the software technology employed, with emphasis on OpenCL as a key enabler for hardware accelerated codes regardless of hosting language.

In Chapter 5 the application codes and investigatory processes are described and documented.

Chapter 6, details the findings and performance characteristics from the investigatory code.

While the project goals did not change, Chapter 7 details amendments and rational of changes to the original project plan.

Chapter 8, what we may conclude from this piece of work.
2 Background

2.1 Accelerators

The hardware market is constantly evolving and producing more capable and performant devices. The proliferation of accelerator devices such as Cells, FPGA and more recently the GPGPU has provided the programmer great scope in choosing a platform to speeding up a scientific code, it has however tied that code to a particular class of device or manufacturer. OpenCL is seen as a software solution to provide generic approach to programming these devices and potentially future multi and many core CPUs.

Computational accelerator devices all have different properties that allow them to speed data manipulation either through different architectures or physical implementations. These differing architectures may be inherently suited to task parallel or data parallel problems or may just provide a hard rather than soft implementation of an algorithm such as on an FPGA. What is clear however is that once a suitable class of device has been selected the code will likely outlive the hardware for which is was initially designed.

As stated it is currently the GPGPU market that is driving innovation in the accelerator hardware field, NVidia with its GF100 (Fermi) chip based cards looks to provide high raw performance with error correcting memory in its Tesla 2000 series cards. AMD is also pushing its FireStream [11] platform which may not have the error correction memory but has exceptional integer and power consumption performance. In Figure 2-1 the performance statistics demonstrate that there can be compromises in choosing hardware platforms.

<table>
<thead>
<tr>
<th></th>
<th>NVidia GTX 480</th>
<th>AMD 9370</th>
<th>NVidia M2050</th>
<th>NVidia M2070</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Elements</td>
<td>480</td>
<td>800</td>
<td>448</td>
<td>448</td>
</tr>
<tr>
<td>Peak GFLOPs Single</td>
<td>1344</td>
<td>2640</td>
<td>1030</td>
<td>1030</td>
</tr>
<tr>
<td>Peak GFLOPs Double</td>
<td>336</td>
<td>528</td>
<td>515</td>
<td>515</td>
</tr>
<tr>
<td>Memory</td>
<td>1.5GB</td>
<td>4GB</td>
<td>3GB ECC</td>
<td>6GB ECC</td>
</tr>
<tr>
<td>Power, max (W)</td>
<td>238</td>
<td>240</td>
<td>238</td>
<td>238</td>
</tr>
<tr>
<td>Video Out</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Price (£)</td>
<td>300</td>
<td>~1200</td>
<td>2500</td>
<td>&gt;3000</td>
</tr>
</tbody>
</table>

Figure 2-1 – Manufacturer GPGPU performance metrics

Moreover in implementing any system, environmental considerations such as power consumption and heat dissipation are key issues. Given that a large part of the lifetime costs of a high performance system is that of on-going running cost, thought should be given to how many accelerators in how many hosts and how should the application be decomposed over the various pieces of hardware. It is not necessarily the case that lots of the biggest card you can acquire will provide an optimal performance increase to the application.
2.2 Drive to portability

Due to the diversity in the architectures of computing devices each has had its own development environment and toolsets, this range of environments and the vast array of solutions computers are used solve has spawned many languages and libraries. With such a large number of development options, diversity was inevitable. Options allow the programmer to use the most appropriate language for the code they are required to write such as C or FORTRAN for numeric routines and higher level object oriented languages for the graphical user interface or communications. Many applications do comprising more than one language in its development, for example many standard libraries such as MPI [12], FFTW3 [13] are written in C but are frequently used from FORTRAN codes. Interoperability of diverse code and libraries is often not as simple or portable as would be desired, with issues such as memory layout. A potential solution to this has been the use of a virtual execution environment such as the Java runtime and the CLR.

In particular the CLR represents a platform designed to run any number of different languages, from managed implementations of classic C and FORTRAN through to modern object and dynamic typed languages like C# and Python or functional languages like ML and F#. Within the CLR, common code from the various languages co-exists and interoperates through common standards for data types and method invocations.

The use of Open Standards and latterly Open Source Software has helped drive the take up of exemplar technologies providing a mechanism for more development effort to be focussed on the solving of solutions rather than building of frameworks. Providing a portable method to run HPC code on heterogeneous hardware can facilitate the abstraction of the numerical problem from the binary implementation i.e. once the code has been produced once it may be reused often, as in a library. Libraries however tend to have additional non-portable attributes such as specific operating systems, processors or compilers; as computational problems are solved the knowledge encapsulated by the algorithm or code should made as widely available, as license permits, without further superfluous porting or re-writing effort. The lowest levels of encapsulation of numerical routines could be within OpenCL kernel codes and would allow cross platform and cross language usage. This project used an object oriented interface, OpenCL.Net [14] to host and manage such OpenCL kernel code.
2.3 HPC kernels

High performance computing is mainly about the ability to run mathematically based computer simulations as fast and large as possible. At the core of the maths are the so called computational dwarves, these are classes of algorithms that when implemented and executed solve fundamental mathematical problems.

While the functions performed by these pieces of code may be part of a complex scientific system many are implemented as simple loops and arithmetic instructions. A number of such kernels were implemented by the Java Grande Forum as part of a benchmark in order to gauge suitability of a managed runtime to execute HPC tasks. These benchmark kernels were implemented primarily as serial codes and not scoped to cover parallel or hardware accelerated execution environments. However libraries of these functions such as basic linear algebra (BLAS) have, in part, execution profiles that are well suited to parallel execution or accelerator assistance. To enable the scaling and higher computational density we look to take these kernels and implement them on a hardware platform that can provide as performant an execution environment as possible.
3 OpenCL

Managed by the Khronos Group, Open Compute Language (OpenCL) is a framework for the programming and execution of accelerator code for cross-platform and cross-device environments in order to improve speed and capability of application execution. OpenCL is designed to be efficient and provide a close-to-the-metal programming interface providing for embedded up to HPC scale applications.

Khronos Group has been managing the OpenGL \cite{15} graphics and OpenSL \cite{16} audio as well as other standards for many years, for which they have gained community respect. In 2009 Apple computers in conjunction with NVidia, AMD, Intel, IBM and many others approached the Khronos Group to develop and maintain an open systems compute platform API that was closely linked by virtue of nascent GPGPU capabilities to the OpenGL graphics platform.

The standard was first publicly released in 2009 at version 1.0.42, to which most current implementations are accredited; there was however an updated specification version 1.1.33, released in June 2010. The update included a number of fixes plus some additional functionality and mandatory data types.

The OpenCL is a driver based interoperability layer that provides a generic view of a hardware accelerator device in order to facilitate a unified approach to accelerator programming. Being a driver based low-level framework OpenCL provides only elements that can be implemented across all compute devices, such as loops and basic arithmetic functions. There is no support for programmatic recursion or general IO but the close tie with OpenGL allows for compatibility between image formatted data. There is limited support for standard parallel programming intrinsic methods such as limited atomic actions and no reduction methods.

Some of the other GPGPU SDKs provide libraries such as basic linear algebra BLAS routines but OpenCL provides no such libraries. Some vendors are now providing sample routines such as Fast Fourier Transforms as example code on how to write code for a given device architecture or to use hardware specific features of their devices.

Being such a recent standard there is little material available for reference; at time of writing only a single electronic book \cite{17} was available with some tutorials \cite{18} and websites content being basic in nature. Further many of the development tools that more established frameworks can provide such as profiling and debugging are lacking or are released at beta testing stage.
3.1 Framework

In order to operate across a wide range of devices, OpenCL is provided as a runtime and support tools in-order to compile and manage execution of programmer provided OpenCL C based kernels.

Implementations of OpenCL are based on an installable client device (ICD) that provides the entry point to operating system driver access. The OpenCL functionality for a GPGPU device is generally implemented as part of the normal video driver package. On a standard UNIX based install, entry points to the OpenCL interfaces are listed as text files in the `/etc/OpenCL/vendors/` directory, within Microsoft Windows installations the ICD entries are listed in the `HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\OpenCL\Vendors` registry key. It is therefore possible to install device drivers from more than one vendor so as to simultaneously support multiple devices, such as a CPU and GPU devices.

The framework uses the ICD and OpenCL SDK to provide functionality to construct a programmer ‘Context’. This is used to link devices, memory and driver code together, in a management role. One of the key components of the context is the ‘Command Queue’; the command queue is the method in which you interoperate with the devices such as transferring memory, executing code etc.

The framework objects are hierarchical in nature as demonstrated in Figure 3-1 below. In OpenCL as compared to CUDA there is no direct initialisation of the platform but once constructs such as a context has been created then subordinate items like command queues must be associated with a parent context.

![Figure 3-1 – Framework objects](image-url)
It should be noted that devices are contained within a platform but must be associated with a context to be used. Command queues can be built utilising mixed disparate devices such as CPU and GPU devices.

3.2 Platform

A generic platform such as OpenCL must look to provide a consistent, coherent approach to implementation regardless of hardware in order to be of value. This is done by defining a host and slave device architecture as in Figure 3-2 below, where the compute platform is seen as subordinate. The host is the processor controlling and submitting code to run on the device. There are also three basic types of device types:

- CL_DEVICE_TYPE_CPU
- CL_DEVICE_TYPE_GPU
- CL_DEVICE_TYPE_ACCELERATOR

plus two additional device types for ease of programming:

- CL_DEVICE_TYPE_DEFAULT
- CL_DEVICE_TYPE_ALL

Regardless of device capabilities, generic terminology is used to describe architecture and capabilities of the devices.

![Figure 3-2 – OpenCL Platform architecture](image-url)
3.2.1 Generic Device

The generic device architecture is based around one or more compute units that contain a number of processing elements, as shown in Figure 3-3 below. The compute units would relate to a processor core within a CPU or a streaming multiprocessor within a GPGPU. Within the compute units there are a number of processing elements which execute in a workgroup of single instruction single data (SISD) or single instruction multiple data (SIMD) fashion; these could be a GPU core or a simultaneous multithreading [19] x86 core.

There are a number of compute units per device and these may be used single or in parallel to function as a workgroup of processors. Within the compute units, work items and are arranged in up to 3 dimensions with a variable given extent in each direction, not all devices may implement all dimensions. The maximum supported size and dimensions of the device can be retrieved from the device information data.

![Generic Device Diagram](image)

**Figure 3-3 – Generic device layout**

It is possible for a processing element to request its position within the local compute unit or in the global workgroup so as to distinguish the id or data unique to its thread.
3.2.2 Generic Host

The host consists of the main CPU and memory; it is responsible for the management of slave devices from initialisation, job queuing, data transfer etc. The host specification of the OpenCL standard provides methods to carry out the following:

- Device and capability enumeration,
- Creation and management of an execution context,
- The creation of executable kernels based on either source or binary code,
- The setting of kernel parameters for runtime execution,
- Creation, management and transfer of memory objects between host and device,
- Creation and management of job execution queues and
- Runtime diagnostics and operational metrics

The host connection to its devices is not specified by the standard and may make use of any communications channel such as PCI express bus or networked communications. Host to device communications may be a major inhibitor to performance if the cost of data transfer outweighs benefit of transferring the load.

Device enumeration

Within an OpenCL environment there may be one or more platforms, this will relate to the number of vendor drivers installed. Calling the `clGetPlatformIDs` OpenCL method and subsequent calls to `clGetPlatformInfo` will enumerate information on the platforms available as in Figure 3-4, below.

<table>
<thead>
<tr>
<th>Name</th>
<th>NVIDIA CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profile</td>
<td>FULL_PROFILE</td>
</tr>
<tr>
<td>Vendor</td>
<td>NVIDIA Corporation</td>
</tr>
<tr>
<td>Version</td>
<td>OpenCL 1.0 CUDA 3.1.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>ATI Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profile</td>
<td>FULL_PROFILE</td>
</tr>
<tr>
<td>Vendor</td>
<td>Advanced Micro Devices, Inc.</td>
</tr>
<tr>
<td>Version</td>
<td>OpenCL 1.0 ATI-Stream-v2.1 (145)</td>
</tr>
</tbody>
</table>

Figure 3-4 – Platform information

Within a platform, one or more devices may exist; these may be of mixed device types such as CPU and GPU etc. The `clGetDeviceIDs` and `clGetDeviceInfo` methods can be used to enumerate and query devices for standard properties such as name, type, computing units etc. and any specific device extensions that the hardware may support such as double precision calculations, vector types etc. See Figure 3-5 and Figure 3-6 below for examples of default properties and values for a GPU and CPU device respectively.
<table>
<thead>
<tr>
<th>Name</th>
<th>GeForce GTX 480</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor</td>
<td>NVIDIA Corporation</td>
</tr>
<tr>
<td>VendorID</td>
<td>4318</td>
</tr>
<tr>
<td>DriverVersion</td>
<td>257.21</td>
</tr>
<tr>
<td>Profile</td>
<td>FULL_PROFILE</td>
</tr>
<tr>
<td>Version</td>
<td>OpenCL 1.0 CUDA</td>
</tr>
<tr>
<td>Extensions</td>
<td>cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing cl_nv_d3d9_sharing cl_nv_d3d10_sharing cl_khr_d3d10_sharing cl_nv_d3d11_sharing cl_nv_compiler_options cl_nv_device_attribute_query cl_nvPragma_unroll cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_fp64</td>
</tr>
<tr>
<td>DeviceType</td>
<td>GPU</td>
</tr>
<tr>
<td>MaxComputeUnits</td>
<td>15</td>
</tr>
<tr>
<td>MaxWorkItemDimensions</td>
<td>3</td>
</tr>
<tr>
<td>MaxWorkItemSizes</td>
<td>0=1024 1=1024 2=64</td>
</tr>
<tr>
<td>MaxWorkGroupSize</td>
<td>1024</td>
</tr>
<tr>
<td>PreferredVectorWidthChar</td>
<td>1</td>
</tr>
<tr>
<td>PreferredVectorWidthShort</td>
<td>1</td>
</tr>
<tr>
<td>PreferredVectorWidthInt</td>
<td>1</td>
</tr>
<tr>
<td>PreferredVectorWidthLong</td>
<td>1</td>
</tr>
<tr>
<td>PreferredVectorWidthFloat</td>
<td>1</td>
</tr>
<tr>
<td>MaxClockFrequency</td>
<td>810</td>
</tr>
<tr>
<td>AddressBits</td>
<td>32</td>
</tr>
<tr>
<td>MaxMemAllocSize</td>
<td>394117120</td>
</tr>
<tr>
<td>ImageSupport</td>
<td>True</td>
</tr>
<tr>
<td>MaxReadImageArgs</td>
<td>128</td>
</tr>
<tr>
<td>MaxWriteImageArgs</td>
<td>8</td>
</tr>
<tr>
<td>Image2DMaxWidth</td>
<td>8192</td>
</tr>
<tr>
<td>Image2DMaxHeight</td>
<td>8192</td>
</tr>
<tr>
<td>Image3DMaxWidth</td>
<td>2048</td>
</tr>
<tr>
<td>Image3DMaxHeight</td>
<td>2048</td>
</tr>
<tr>
<td>Image3DMaxDepth</td>
<td>2048</td>
</tr>
<tr>
<td>MaxSamplers</td>
<td>16</td>
</tr>
<tr>
<td>MaxParameterSize</td>
<td>4352</td>
</tr>
<tr>
<td>MemBaseAddrAlign</td>
<td>512</td>
</tr>
<tr>
<td>MinDataTypeAlignSize</td>
<td>16</td>
</tr>
<tr>
<td>SingleFPConfig</td>
<td>63</td>
</tr>
<tr>
<td>GlobalMemCacheType</td>
<td>READ_WRITE_CACHE</td>
</tr>
<tr>
<td>GlobalMemCacheLineSize</td>
<td>128</td>
</tr>
<tr>
<td>GlobalMemCacheSize</td>
<td>245760</td>
</tr>
<tr>
<td>GlobalMemSize</td>
<td>157646840</td>
</tr>
<tr>
<td>MaxConstantBufferSize</td>
<td>65536</td>
</tr>
<tr>
<td>MaxConstantArgs</td>
<td>9</td>
</tr>
<tr>
<td>LocalMemType</td>
<td>LOCAL</td>
</tr>
<tr>
<td>LocalMemSize</td>
<td>49152</td>
</tr>
<tr>
<td>ErrorCorrectionSupport</td>
<td>False</td>
</tr>
<tr>
<td>ProfilingTimerResolution</td>
<td>1000</td>
</tr>
<tr>
<td>EndianLittle</td>
<td>True</td>
</tr>
<tr>
<td>Available</td>
<td>True</td>
</tr>
<tr>
<td>CompilerAvailable</td>
<td>True</td>
</tr>
<tr>
<td>ExecutionCapabilities</td>
<td>1</td>
</tr>
<tr>
<td>QueueProperties</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figure 3-5 – GPU Device Information**
Name: Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz
Vendor: GenuineIntel
VendorID: 4098
DriverVersion: 1.1
Profile: FULL_PROFILE
Version: OpenCL 1.0 ATI-Stream-v2.1 (145)
Extensions: cl_khr_icd cl_amd_fp64 cl_khr_global_int32_base_atomics
cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics
cl_khr_local_int32_extended_atomics cl_khr_byte_addressable_store
cl_khr_gl_sharing cl_ext_device_fission cl_amd_device_attribute_query
cl_amd_printf
DeviceType: CPU
MaxComputeUnits: 8
MaxWorkItemDimensions: 3
MaxWorkItemSizes: 0=1024 1=1024 2=1024
MaxWorkGroupSize: 1024
PreferredVectorWidthChar: 16
PreferredVectorWidthShort: 8
PreferredVectorWidthInt: 4
PreferredVectorWidthLong: 2
PreferredVectorWidthFloat: 4
PreferredVectorWidthDouble: 0
MaxClockFrequency: 2808
AddressBits: 32
MaxMemAllocSize: 536870912
ImageSupport: False
MaxReadImageArgs: 0
MaxWriteImageArgs: 0
Image2DMaxWidth: 0
Image2DMaxHeight: 0
Image3DMaxWidth: 0
Image3DMaxHeight: 0
Image3DMaxDepth: 0
MaxSamplers: 0
MaxParameterSize: 4096
MemBaseAddrAlign: 1024
MinDataTypeAlignSize: 128
SingleFPConfig: 7
GlobalMemCacheType: READ_WRITE_CACHE
GlobalMemCacheLineSize: 0
GlobalMemCacheSize: 0
GlobalMemSize: 1073741824
MaxConstantBufferSize: 65536
MaxConstantArgs: 8
LocalMemType: GLOBAL
LocalMemSize: 32768
ErrorCorrectionSupport: False
ProfilingTimerResolution: 1
EndianLittle: True
Available: True
CompilerAvailable: True
ExecutionCapabilities: 1
QueueProperties: 2

Figure 3-6 – CPU Device Information
Contexts

Contexts are the OpenCL management container that is used to associate one or more devices with command queues, memory objects, programs and kernels. OpenCL provides a number of ways in which a context can be created, either by a list of cl_device_id with clCreateContext or using a clCreateContextFromType method which can associate devices of a similar type such as GPGPU.

Executable

As a principal OpenCL is set to address portability it therefore allows for the execution of on-line or off-line compiled code. The device can be queried to ascertain if a compiler is available locally to compile a given source file, the programmer may then choose to compile device code at runtime or provide a precompiled binary executable. If a binary is provided then there may be dependencies on specific capabilities within the device that will reduce the portability of the code.

From source, device executable code is created using the clCreateProgramFromSource method. The source is normal text based C source code that is compiled using an llvm\(^{[20]}\) based compiler provided via the ICD. Should compilation fail then the build logs can be retrieved by using the clGetProgramBuildInfo method for diagnostic inspection.

Programs either from source or binary may contain a number of kernel routines which are functions marked with a __kernel prefix, as well as support routines and #pragma based options to enable or require features such as double precision calculations. The kernel routines are the primary entry points to program, in order to reference these for execution they have to be explicitly converted to kernel object by the clCreateKernel method.

There is no requirement to specifically transfer executable code to the device as this is managed by the OpenCL runtime.

Memory

The memory architecture is described later in section 3.2.1 below, but the host must support the transfer of memory objects from itself to and from the devices within a context. There are two types of memory transfer based on either a generic memory buffer object or a specific image data format; in essence both operate in similar manner with some additional methods to support the image specific requirements.

The standard memory buffer objects are created using clCreateBuffer which are allocated on the device, the buffers may also be mapped into host memory using clEnqueueMapBuffer method. There is also support for sub selection within buffers or images to facilitate partial data transfers such as halo exchanges.
The transfer of data must be processed through the context command queues using the `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` methods with variations for image format data and selective region transfers. These queue based requests are called with a parameter set to either invoke blocking until transfer is complete or to return immediately. Correctly sequenced command instructions can overlap memory transfers with computations where the hardware device supports it e.g. NVidia Fermi based cards.

**Runtime**

OpenCL runtime is based around the execution of commands through the context command queue. The command queue represents a scheduler for the submission of commands to the device; they may be kernel execution, memory, instrumentation or synchronisation commands. Multiple command queues may associate with a single context but there is no mechanism to synchronise them.

**Kernel Execution**

Kernels can be executed to conform to either task parallel or data parallel threading models, see section 3.3.2 below, with the `clEnqueueNDRangeKernel` and `clEnqueueTask` methods. There are also methods to allow execution of native arbitrarily created code on the device.

**Command Queues**

Command queues are created for a given context using `clCreateCommandQueue`, properties of the queue such as execution order must be set at creation.

Commands that are submitted to the queue are always executed in the order of submission. It is not necessarily the case that a second command will wait for completion of the previous before commencing. Some hardware devices support parallel asynchronous or out-of-order execution where there is no enforced execution order between different kernels, this could inadvertent cause memory corruption or deadlock. It is possible however to enforce serialisation of the submitted commands to the command queue as a property at creation time or by barrier methods.

When a command is queued an event object is passed that allows the monitoring of the status of that command. This status information can be used to determine the success or failure reason of many of the standard commands. It is also possible to attach call back notification to events such that specified host handler may be called on a predetermined condition.

In order to understand the execution of kernel execution and memory commands it is possible to retrieve profiling information on an event using the `clGetEventProfileInfo` method. The information returned is timing data in high resolution micro seconds of queuing and kernel execution profiles.
Command Queue Synchronisation

Two types of runtime commands can be used control queue execution flow; queue internal commands which are submitted to the command channel to act as inline controls and external queue maintenance commands.

Instructions can be submitted to a command queue in order to provide flexibility in synchronisation and interaction with certain events. The correct use of overlapping commands can produce performance gains though performing non dependant actions such as performing a data transfer simultaneously as performing calculations. Explicit synchronisation points are enforced when the clEnqueueMarker, clEnqueueBarrier or clEnqueueWaitForEvent commands are issued. The use of event management enables the uses of call back handlers to host code routines which can either deal with exception conditions or host scheduling decisions.

Externally to the queue there are the clFlush and clFinish commands. The flush method which will clear any currently blocking running commands and the finish method will block until the queued commands are completed.

3.3 Software Architecture

Using OpenCL in an application requires two distinct pieces of code to be written the host side code which mainly deals with the setup and management of the devices and the device code which will be executed upon the device. The second piece of code is the actual kernel code that will run on the accelerator hardware.

OpenCL allows for both online and offline compilation of the device target code, the offline method provides a way for developers to distribute pre-compiled binaries for a given architecture. The benefit of online compilation is that the host may be able to better target or optimise the device code based on known attributes of the devices actually fitted to the machine i.e. maximise use of memory etc.

3.3.1 Host Code

The primary purpose of the host code is to initialise and control the execution of code on the accelerator devices. This is done in several stages, firstly the devices within the platform need to be enumerated, from which an execution context can be created.

Once created a context must contain a command queue and may contain a number of memory objects and kernels. The command queue acts in similar manner to that of an instruction pipeline allowing the host code to schedule memory operations, kernel execution and synchronisation events, further there are call back methods allowing the host program to be notified on a particular event.
Devices may be queried to determine the attributes of the installed devices such as the number of computational units etc. as well as any hardware specific capabilities such as double floating point support. This information may be used before compilation of program code to optimise or replace routines as appropriate to the capabilities present.

### 3.3.2 Device Code

OpenCL is based on the ISO C ’99 standard language; there is support for both scalar and vector data types as well specific data types for image related data formats. Further given that OpenCL is for computational purposes there is only embedded support for maths functions, no external libraries can be used.

It is important to note that while OpenCL provides a standardised method to interact with hardware accelerators the programmer still needs to have a full understanding of the devices physical architecture and capabilities.

Full details of the OpenCL C language and its functions can be found in the specification[^4].

### Thread model

The design of OpenCL supports both data parallel and task parallel methods of kernel execution. Data parallel methods are mainly implemented in simple processing cores that can execute on mass in a Single Instruction Multiple Data (SIMD) manner to perform many calculations at high speed, such as in a GPGPU. Where there are many complex processors that are capable of performing dissimilar or variable sized workloads such as in a task farms, this would be considered task parallel. Task parallel capable processors could be based on multi-core CPUs, Cell processors or IP connected compute node clusters.

### Task / Data parallel Programming

In OpenCL the implementation of task and data parallel programming is essentially the same. The command to submit for execution a range based, data parallel kernel is the ‘**clEnqueueNDRangeKernel**’ method. Principally this calls the single kernel to be executed on a number of processing elements within the workgroup. Required, over and above the kernel handle, are two parameters that determine the number and size of the dimensions of the processing elements required to participate. Within the executing kernel there are a number of embedded functions that will return that particular work items position within the workgroup, these functions are:

- `get_group_id`
- `get_global_id`
- `get_local_id`
The task submission method is ‘**clEnqueueTask**’ this essentially performs the same function as `clEnqueueNDRangeKernel` but where the dimensions and range are pre-set to 1 only.

**Memory Model**

Memory management is explicit within OpenCL, given that different computation devices may have limited memory or memory capabilities such as cache, the following memory hierarchy has been defined as below and shown in Figure 3-7 – OpenCL Memory model below.

In addition to statically assigned buffers that can be transferred from and to the device there is

- **Host:**

  Host memory is the main memory of the host machine. Data created here can be transferred to and from the device global memory. It is also possible to Map areas of host memory into device global memory, the transfer still takes place but is managed by OpenCL.

- **Global:**

  The device global memory is shared across all compute units / workgroups, it is the largest of the local device memories but also the slowest. To inform the compiler that an assigned area of memory is global it is prefixed with ‘__global’

- **Constant:**

  Memory items defined as constant, marked as ‘__const’, may be stored in global memory but will be provided to the processing elements as high speed local elements when required. They are read-only in nature.

- **Shared:**

  Shared memory is the fastest memory available that is shared between processing elements, it may be used as cache or explicitly used as shared access memory with asynchronous pre-fetch capabilities. This memory is marked as ‘__local’ as it is only available to the work items within a single compute unit.

- **Private:**

  Available only within a processing element, private storage is the fastest memory in the hierarchy. It is often implemented as registers within the processing element and is therefore highly constrained in size.
The relaxed consistency model of OpenCL memory is applied across the tiers of the memory hierarchy. That is that for a processing element a load or store is always consistent but access within the local or global memory is only consistent at work-group barrier and cannot be guaranteed on an inter-kernel basis.

Figure 3-7 – OpenCL Memory model

Synchronisation

Previously discussed, were command queue barriers that enforced order in the execution of kernels, within the kernel code we must implement barrier functions in order to internally synchronise a work items within a functioning work-group. This is simply implemented with ‘barrier()’. Further we can enforce the ordered access of memory elements by use of memory fences with the ‘mem_fence()’ method. Both these functions take a flag to signify whether a local or global synch is required; this is signified by using either ‘CLK_LOCAL_MEM_FENCE’ or ‘CLK_GLOBAL_MEM_FENCE’ respectively.

3.4 Execution Model

In order to implement a code in OpenCL there are a series of steps we must take in order to initialise the execution environment, as discussed there are dependencies for command queues, platform and devices etc. In Figure 3-8 below, a sample application flow shows the major stages of initialisation, execution and clean-up which demonstrates the key functions that must be performed.

A number of things stand out such as there is implicit transfer of kernel code to the device and that asynchronous execution of queued memory operations and kernel execution could lead to data corruption or system failure. The use of blocking calls will inevitably cause delay or lockup in the host thread if a previous operation takes too long to complete. A possible solution is to use delegate handler to manage
queued events or submit a number of actions with explicit queue barriers inserted between sensitive steps.

Additionally memory transfers between host and target device are over interconnects such as PCI Express, the latency of memory transfers needs to be factored as a cost of execution.

```
# Ifunction.cl
__kernel void add(__global int *a, __global int *b)
{
    private id = get_global_id(0);
    a[id] += b[id];
}
```

**Figure 3-8 – OpenCL sample application flow**
4 Common Language Runtime

The Common Language Runtime (CLR) is a technology that is more commonly known as the Microsoft .Net Framework. Microsoft released the C# and CLR standard to the European Computer Manufacturers Association (ECMA) in 2002. The ECMA since the 1960s have been responsible for maintaining a number of industries created technical standards, further in 2006 the ISO/IEC also adopted the C# and CLR standards as ISO/IEC 23270:2006 and ISO/IEC 23271:2006. Core features of a CLR runtime environment are memory and process management, security and exception handling. The ECMA updated and ISO release of the standards improved on core features and introduced a number of new features such as generics, lambda expressions and parallel threads in the CLR and support for dynamic typing, anonymous methods, iterators and nullable types etc. in the language support.

There are a number of benefits in using a Common Language Runtime, foremost being that the CLR allows for binary compatibility of compiled applications between different platforms and architectures in the form of Common Intermediate Language (CIL) code. Moreover the CLR was designed to support multiple different languages and provide first class interoperability between compiled assemblies from these various sources. The runtime includes a fully featured core library providing language independent support for features such as native library interoperability, text regular expression and IO handling. The core library is extended through additional assembly objects providing functions from web services support to complex machine learning capabilities; many of these libraries are provided as open source or with an open license as well as the commercial packages. Being a virtual execution environment the CLR has the ability to perform run time optimisations that would be unavailable to statically compiled code; these could include intelligent in-lining, dead code elimination or cost estimation.

There are drawbacks in using such a managed virtual execution systems (VES) like the CLR or Java, primarily focussed on interoperability with native, static code. Managed VESs maintain the runtime memory allocation for code and data objects thereby enabling reuse or compaction where appropriate, given that blocks of memory belong to the CLR we must either use memory ‘pinned’ in place or use a marshalled method to transfer between manage and unmanaged allocated memory blocks. There is further a wrapping of the underlying native driver code such that it presents itself within CLR namespace, which will add some calling overhead.

While the stance taken by this project is that of cross platform compatibility, it will always be the case that codes possibly need to integrate with the physical environment, such as operating systems and GUls. Therefore codes may include truly independent platform agnostic sections and platform targeted specific sections to support local capabilities or user experience rules.
4.1 CLR Programmer Support

The C# language is the primary language supported by CLRs as the language specification is part of the open standards. There are, however, a number of other languages \cite{27} from managed implementations of C++, Java, Visual Basic, and FORTRAN etc. as well as dynamic typed languages like Ruby, Python through to modern functional and imperative languages like ML, and F#.

This project will use C# as the implementation language of choice for all CLR codes. Further cross platform tools will be used to ensure development and build processes can be produced across multiple platforms. On Linux and OSX the open source implementation of the CLR is Mono\cite{28} and developer environment is MonoDevelop. Mono provides a complete implementation of the CLI with a near complete runtime system library, it runs on many processor architectures.

The open source development suite provides many of the features of commercial grade IDEs such as Microsoft Visual Studio 2008 but NVidia and AMD GPGPU SDKs currently only provide deep debug and profiling information as plugins to Visual Studio, such as NVidias NSight\cite{29} toolkit. There are alternative cross platform OpenCL debuggers such as gDEBugger\cite{30} from Graphic Remedy which is based on previous OpenGL debugging technology. gDEBugger is still in beta with respect to OpenCL support and the stability at this point was variable, while providing good information while operational. NVidias OpenCL profiler provided as part of their developer SDK \cite{31} and is available for both windows and Linux environments.

More practically, the source deployment process should follow norm by being invoked through ‘make’ and other expected tools. The appropriate command scripts can be hand coded or the MonoDevelop application can be used to generate the appropriate ‘configure’ and ‘make’ scripts. Many other tools such as debuggers and profilers are available in command line or GUI form with open source or commercial license.

4.2 Mono CLR

The most widespread open source implementation of the CLR is Mono, this project is sponsored by Novell and has a number of associated commercial pay to use products. The Mono Project was originally started by Miguel de Icaza, at the time a lead developer on GNOME, as an easier way to implement desktop applications. It has since developed into a fully featured development and execution environment.

Mono can be run on machines of all sizes from iPhones and iPads at the lower scale through to super computers. The runtime is supported on many processors such as x86, x86-64, ARM, MIPS, Power PC etc. and can be built from a source distribution or implemented from pre-packaged installers. The Mono team are closely tied to the SUSE Linux team through Novell and as such Linux installation is automated with
‘automake’ tools, which builds easily on OpenSUSE and derivatives such as Cray compute node Linux. Other than build and package tools the only dependency Mono has is for ‘glib’ a threading library; further libraries to support localised device interfaces, open source or commercial applications etc. can also be installed.

The current release of Mono is version 2.6.7 which can be built with support for the LLVM compiler thereby producing static, non just-in-time (JIT) compiled code. The use of this option removes the opportunities for runtime optimisations and produces a platform specific executable this may though be of use when JIT related factors should be removed from code execution.

While the Mono implementation of new features generally tracks that of Microsoft all major features such as Language Integrated Query (LINQ), web services, dynamic runtime typing etc. are implemented. The Mono CLR supports most but not all of the features of the runtime class library; some methods have not yet been implemented, such as ‘System.Text.RegEx.CompileToAssembly’. There are additional classes such as ‘Mono.Unix’ and ‘Mono.SIMD’ which make available Unix platform specific functions.

The Mono C# compiler ‘gmcs’, can be used to convert C# source files to CLR code or byte code executable on any CLR. Similarly the Microsoft C# compiler ‘csc.exe’ can be used to compile code for execution on the Mono runtime. The mono runtime is invoked to execute a code using ‘mono <application.exe>’ whereas on windows the executable can be run directly.

Standard development tasks such as profiling, step wise debugging, and tracing are all fully supported with various levels of tools being provided.
5 Methodology

This project looks to investigate execution of mathematical functions on accelerator hardware in as cross platform a manner as possible. To do this we will write a number of codes that exercise OpenCL and perform real world calculations in order to determine the feasibility, suitability and reliability of implementing a production code in such a scenario. This chapter looks to describe the specifics of the work undertake to develop a testable solution.

During the initial pre-project investigations a number of codes were selected as candidates for potential conversion to OpenCL kernels. The Java Grande Forum had previously established a number of benchmarks which were used to compare the performance of Java Virtual Machine (JVM) coded mathematical routines with the statically compiled implementations. The Java Grande Test Suite consists of three sections of tests, the first group relate to timing of functions such as arithmetic throughput, casting, maths libraries etc. The second section is mathematical kernels performing LU factorisation, Successive over-relaxation, Fourier Transformations and Sparse Matrix Multiplication etc. Lastly there are a number of applications such as Computational Fluid and Molecular dynamics simulations.

Section 2 represents routines that may be suitable targets for porting to GPGPUs, in particular the LU Factorisation, Sparse Matrix Multiple and Fast Fourier Transformation each present an opportunity and challenge for implementation. These routines contain functionality that well represents the HPC kernel and that can be used to provide statistical analysis of performance improvements.

The project looks to deliver in a two staged approach of producing CPU only test codes and secondary set that and OpenCL based GPGPU accelerated test codes.

The first set of tests looks to further validate the suitability of HPC kernel based programs to be run on manage runtime environments. Further in determining which kernel performs best in native code we can look to target this routine as a best case for speedup in a managed code implementation.

The second set of tests looks to compare the implementation of such kernels by using OpenCL as technology to accelerate both statically compiled and managed runtime variants of the same kernel operations. Comparing the native, managed, native OpenCL and managed OpenCL version of the same code we look to consider how this may provide us speedup and advantage.
5.1 CPU Execution

The reference C and JAVA codes were taken from the Java Grande Benchmark suite and were run unmodified, further an implementation of the chosen kernels was required in C#. The need to re-run the reference C and Java codes was two-fold, firstly the compiler and JVM technologies have significantly advanced in almost a decade since the original Java Grande piece was produced. Secondly, processor technologies have changed considerably with four or five iterations of Moore’s Law occurring, while Intel x86 based systems with high clock rates, large cache and high speed memory has become the current dominant system architecture.

The previous Java Grande Forum work was revisited for the CLR in HPC.Net by Werner Volgels [32] in 2003; a set of codes had been written in C# to execute the Java Grande Benchmark, these codes were released but have since become unavailable. Therefore a new subset of the benchmark was created to act as a reference CLR based implementing having only the following Java Grande Benchmark Section 2 tests:

- LU Factorisation
- Sparse Matrices Operations
- Fast Fourier Transformations

5.1.1 cpuBenchmarks

After an initial view of the original source code, the C version was ported to the C# language, limited changes were made in the construction of program flow i.e. use of class containers, methods, IO etc. and modification of arrays handling. Given the nature of these tests a simple test harness was built to automate the execution these test, that code is cpuBenchmarks.

In managed environments and their execution virtual machines the management of memory is performed by the runtime, as such CLR array data types are notated and implemented differently than in C and more like that of Java. The CLR specification allows for both 2 dimensional arrays ([x,y]) as well as jagged arrays of arrays ([x][y]). The C# code follows the original C code by using 2D notation.

The C benchmarks used an external high resolution timing code that was replaced with the System.Diagnostics.StopWatch library that can be used for reliable timing at the millisecond or system tick level.

Porting and verification of the code was straight forwards with some minor issues arising in the sparse matrix code, whereby an undetermined casting of a float by double multiplication caused lack of precision.
5.2 OpenCL Execution

In order to benchmark the performance of the OpenCL implementation the selected Section 2 Kernel was required to be written as OpenCL based code. Further to porting the kernel source an OpenCL host code would be required to manage the execution of the accelerated code. To determine the overheads when utilising managed runtimes versus non-managed runtimes a reference C host code was also written, as previously discussed, GPGPU technologies have been driving need for OpenCL and will be used as the target architecture for these tests.

Initially a basic code was produced to test basic driver enumeration and gather platform information. A given principal of this project is that a single code should be producible and executable on any platform without modification of this code; however it does not mean that each physical platform provides the basic requirements to run the tests code, such as double floating point precision. A code called **oclEnumerate** was implemented in C# to provide specific platform and device information as well as to test driver connectivity on a number of platforms.

Using OpenCL within a managed runtime, such as CLR or JVM also requires a secondary interoperability layer to enable calling the driver native interface. There are a number of open sourced object libraries available for Java and CLR implementation. We must also look at the overheads incurred by wrapping native code for execution in managed environments.

5.2.1 OpenCL Libraries

From the pre-project investigations, the OpenCL.Net library from SourceForge was used for these tests. Several other OpenCL libraries were tested but the OpenCL.Net library was evaluated to be the cleanest implementation with as little deviation from the OpenCL C Standards API. In addition to the open source OpenTK[^33] and free commercially licensed Hoopoe Cloud OpenCL.Net[^34], during the project a small effort was utilised in the evaluation of another open source library named Cloo[^35] which looked to provide better object oriented features.

Within the selected SourceForge OpenCL.Net library minor amendment were made to the code in order to support the Mono automatic DLL to UNIX library mapper; the author had coded the name of the supporting driver code to be ‘opencl’ all in lowercase. This works fine for Mono and .Net on windows that will look for OpenCL.dll regardless of case, but Unix case sensitive file systems will not find the library. Modifying the library source to ‘OpenCL’ will indeed now enable the Mono runtime on Unix to find the native support libraries with prefix ‘lib’ and postfix ‘.so’. This suggestion was put back to the library author but no further communications were received.

[^33]: OpenTK
[^34]: OpenCL.Net
[^35]: Cloo
In order to determine if the runtime can load a particular library or verify the search paths mono can be called with additional logging data as in Figure 5-1.

```
[s0973635@nessgpu2 dis]$ MONO_LOG_LEVEL=debug mono oclHarness.exe
Mono-Message: gc took 96 usecs
Mono-INFO: Assembly Loader probing location: "/work/s02/s0973635 mono/lib mono/2.0/mscorlib.dll".
Mono-INFO: Image address mscorlib 0x8fd3d70 ->
/work/s02/s0973635 mono/lib mono/2.0/mscorlib.dll 0x8fd3340: 2
Mono-INFO: Assembly Loader probing location: "/work/s02/s0973635 mono/lib mono/2.0/mscorlib.dll".
Mono-INFO: AOT loaded AOT Module for /work/s02/s0973635 mono/lib mono/2.0/mscorlib.dll.
...
Mono-INFO: DllImport attempting to load: 'libOpenCL.so'.
Mono-INFO: DllImport loading location: 'libOpenCL.so'.
Mono-INFO: Searching for 'clGetPlatformIDs'.
Mono-INFO: Probing 'clGetPlatformIDs'.
Mono-INFO: Found as 'clGetPlatformIDs'.
```

**Figure 5-1 – Mono debug output of native OpenCL library initialisation.**

Initial investigations into the use of OpenCL on a number of platforms demonstrated that the order of platforms and devices was not necessarily consistent, for example on the EPCC Fermi0 machine the devices listed in order are Tesla C2050, GeForce 8400, Tesla C2050, Tesla C2050, Tesla C2050. To facilitate the discovery of installed hardware oclEnumerate was written both to test the OpenCL stack and enumerate the installed devices. Output examples were previously given in Figure 3-5 and Figure 3-6.

To build and execute oclEnumerate please see the README enclosed with the distribution.

### 5.2.3 oclHarness

The OpenCL host code for the C and C# programs is called **oclHarness**, the system specific C executable is oclHarness and the CLR bytecode is oclHarness.exe. Both codes are constructed in a similar manner, performing the same series of actions in the same order.

To build and execute oclHarness please see the README enclosed with the distribution.

These codes implements a series of tests that begin by perform a single kernel verification, this is achieved through execution of a parallel add kernel; this is followed by a configurable set of test class objects. Ideally the test selection would be through command line parameters but currently default execution parameters are hardcoded and must be amended and recompiled for testing.
In the C# version the base class for a test function is the TestHarness, this provides a prototype for the **struct benchmarkTimes** and the initialisation and execution methods. There were further implications with the use of a two dimensional array for the matrix within the CLR; for a normal vector the data can be passed from the CLR managed memory to the unmanaged statically allocated buffers and queued to be transferred to the device. This requires a double allocation of memory for both the managed array and the raw array as is transferred to the device, it should be possible to use either device mapped or CLR pinned memory to reduce this to a single allocation. This memory use is further exacerbated for the case of the matrix as the Marshal.Copy method used to transfer between the managed arrays and the unmanaged buffer space lacks an overload method for the case of a two dimensional array, in this case we do a two stage transpose from a two dimensional array to a larger single dimensional array and finally to the unmanaged buffer. This must be repeated when retrieving the data back from the card. The additional memory copy is unnecessarily wasteful and the function could easily be added to the open source system library.

Test methods implemented were for a number of tests but only LU Factorisation codes were completed. The oclHarness code produced acted as the host code and benchmark harness for the OpenCL source files, the developed versions of OpenCL code are called LUFact.cl and LUFactPar.cs with both sources being shared between native and managed code.

### 5.2.4 Kernel Porting

The importance of execution of a complete HPC kernel routine led to a number of variants of a single code being written, rather than the originally intended three kernels focus was directed to the LU Factorisation code.

When porting the LUFact kernel code an initial naïve method of implementing the BLAS matrix **dgefa** factorisation and **dgesl** solver routines with the associated **idamax**, **daxpy**, **dscal** and **ddot** functions was attempted. This first code is a straight port and takes no advantage of the accelerator hardware. A number of minor changes had to be made in order to accommodate the execution in OpenCL.

The two major kernel **dgefa** and **dgesl** routines were marked up with OpenCL directives in order to identify the kernel routines and allocated runtime memory. Since there was no attempt made to use the parallelism of the hardware, the kernels were enqueued to the device as single tasks. This however proved to be unreliable in that the **dgefa** routine would not execute to completion, even for small sizes of n which should complete in little time. This was probably due to all calculations being performed in slower global memory by a single thread so either causing memory deadlocks or increased execution time such that the device driver would abort the job.
In order to investigate the kernel operation and provide the capability of debugging and profiling a modified kernel was created that essentially moved the outer loop from within the dgefa routine to the host code. These modified versions were named dgefaCL and dgeslCL as in Figure 5-2. The OpenCL based codes also took some cognisance of the device architecture and attempted to use local high speed memory for the duration of an execution. The dgefaCL routine was to be called many times for the matrix a for row of k.

All three OpenCL memory types were used, ‘global’ memory buffers for the major parameters of a[n,n], b[n], ipvt[n] and ‘local’ high speed but limited workgroup for temporary vector use. As dynamic memory allocation is not allowed, workgroup memory allocations for col_k[n] and col_j[n] have to be passed into the kernel as parameters. Additional parameters were used for control and debugging purposes.

The main loop of the code is executed by iterating through the row index on the dgefaCL routine and finishing with the dgeslCL routine as demonstrated in Figure 5-3. Given there exists a data dependency between executions of the dgefaCL kernel, EnqueueBarrier() directives are used to ensure that subsequent kernelDgefa tasks are not started before the previous cycle is complete, which may be possible on some hardware.

```c
__kernel void dgefaCL(__global double* a, int c, __global int* ipvt, int n, int k,
__local double* col_k,__local double* col_j)
__kernel void dgeslCL(__global double* a, __const int lda, __const int n, __global
int* ipvt, __global double* b, int job, __local double* col_k, __local double* loc_b)

for (int k = 0; k < n - 1; k++)
{
    kernelDgefa.SetArg(5, k);
    cl.CQs[devIndx].EnqueueTask(kernelDgefa);
    cl.CQs[devIndx].EnqueueBarrier();
}
cl.CQs[devIndx].EnqueueTask(kernelDgesl);
cl.CQs[devIndx].Finish();
```

Figure 5-3 – Main host code execution loop.
The initial runs of the naïve code demonstrated the key issues that hinder the development processes such as driver function, profiling and debugging facilities. Several profilers were tried with the NVidia OpenCL Visual Profiler v1.1.20 providing the most reliable results and from which some diagrams in this document were created. gDEBugger and AMD tools were also tried but were found to be inadequate in stability and performance, both are still in beta so representable as a finished product.

Initial runs of the code took in excess of two to three seconds, after which on both Windows and Linux this caused the display driver to reset aborting kernel execution, with dedicated computational cards such as the NVidia Tesla series this reset does not happen. With the dgefaCL kernel modified to only externalise the iterations across the rows of the matrix, this allowed for a profiling run to execute demonstrating the program work plan as in Figure 5-4 below.
Figure 5-4 – Profile of LUFact.cl for LU factorial n=250
We can now clearly see that over the repeated execution of the dgefaCL routine over the rows of the matrix, the cyan area, that as the triangular area progresses the columns of data get shorter. This gives us a variable time to execution based on the size of n, which is O(n). Many of the functions with dgesICL would perform well if the code could be run in a SIMD, data parallel manner; vector scale and parallel reductions are known to be efficient.

After consideration of the first code a second set of kernel routines where then implemented, where the partitioning of the code between host and device was modified to allow parallel optimisations to be made in the support routines of idamax, daxpy, dscal and ddot. BLAS level 1 routines such as dscal and daxpy are simple arithmetic operations performed on a vector which may easily be ported for data parallel execution. The idamax function returns the index of the largest absolute value within a vector; this must be coded as a parallel reduction to achieve performance gains of a GPGPU.

For the purposes of execution these were potentially wrapped and called in the kernel routines of idamaxCLND, scaleCLND, elimCLND and solveCLND, prototypes of which are in Figure 5-5.

```c
__kernel void idamaxCLND( int k, __global double* a, int sz, int incx,
__global int* ipvt, __local double* dmaxv, __local int* lclReduce)
__kernel void scaleCLND( int k, __global double* a, int sz, __global int* ipvt, __local double* col_k)
__kernel void elimCLND(int j, int k, __global double* a, int sz, __global int* ipvt, __local double* col_j)
__kernel void solveCLND( __global double* a, __const int lda, __const int n, __global int* ipvt, __global double* b, __local double* loc_b)
```

**Figure 5-5 – Prototypes of kernels idamaxCLND, scaleCLND, elimCLND and solveCLND**

The execution of these new CLND kernels required that steps in the outer loop of dgefa be queued by the host code in sequence, as in Figure 5-6. This did allow for more scope in the implementation of parallel algorithms and execution of the kernels in parallel with the EnqueueNDRangeKernel option. This was called with a workgroup sizing in one dimension with a size of n, this would cause exec failure if n was greater than the physical device maximum; in this case 1024. There was no consideration given to further decomposition to multiple workgroups.
The idamaxCLND kernel was modified to perform a parallel reduction method inspired by NVidia’s OpenCL programming overview [36]. This approach involved utilising two local structures to maintain the maximum value and original index of the values in a vector while trying to avoid memory bank conflicts during access. The kernel code could now be called with a data parallel execution model with half the threads required to execute for each iteration of idamax. In Figure 5-7 below the two temporary data structures lclReduce and dmaxv hold the local reduction index and double precision maximum value, these are initialised as id and column[0] value respectively.

```c
for (int k = 0; k < nm1; k++){
    kernelIdamaxCLND.SetArg(0, k);
    kernelScaleCLND.SetArg(0, k);
    kernelElimCLND.SetArg(1, k);

    // find l = pivot index
    cl.CQs[devIndx].EnqueueNDRangeKernel(kernelIdamaxCLND, dims, null,
                                           globalWorkSize, null);
    cl.CQs[devIndx].EnqueueBarrier();

    // interchange if necessary
    cl.CQs[devIndx].EnqueueNDRangeKernel(kernelScaleCLND, dims, null,
                                           globalWorkSize, null);
    cl.CQs[devIndx].EnqueueBarrier();

    // row elimination with column indexing
    for (int j = k + 1; j < n; j++){
        kernelElimCLND.SetArg(0, j);

        cl.CQs[devIndx].EnqueueNDRangeKernel(kernelElimCLND, dims, null,
                                               globalWorkSize, null);
        cl.CQs[devIndx].EnqueueBarrier();
    }
    cl.CQs[devIndx].Finish();
}
```

Figure 5-6 – Host code implementing dgesl using separate idamax, scale and eliminate kernels

The idamaxCLND kernel was modified to perform a parallel reduction method inspired by NVidia’s OpenCL programming overview [36]. This approach involved utilising two local structures to maintain the maximum value and original index of the values in a vector while trying to avoid memory bank conflicts during access. The kernel code could now be called with a data parallel execution model with half the threads required to execute for each iteration of idamax. In Figure 5-7 below the two temporary data structures lclReduce and dmaxv hold the local reduction index and double precision maximum value, these are initialised as id and column[0] value respectively.
Through the iterations of the idamaxCLND the parallel implementation selectively moves the maximum value and its index to index 0 of the arrays. This is implemented using a strided access to memory and using half the number of threads performing a single comparison. In each step the functioning threads are shaded in the lclReduce row as blue, the value stored by the reduction is selected by the green arrows.

Further the scaleCLND and elimCLND codes were also modified to execute a number of parallel threads where the column index was the thread id. The appropriate memory synchronisation barriers were also inserted. The scaleCLND was a simple example of each thread performing a single calculation rather than a loop. elimCLND was slightly more complex as it had loops of master only and parallel code, memory fences were used as a barrier to ensuring memory consistency.

These improvements aided stability and ability to debug the inner actions of the code. The profile of this code executing is in Figure 5-8 below.
Figure 5.8 – Profile of LUFactPar.c for LU Factorisation n=250
The LUFactPar.cl execution profile shows a significantly different profile for the former LUFact.cl code. The optimised idamaxCLND, elimCLND and scaleCLND routines execute extremely quick compared to the previous iterations of dgefaCL, order of 5us compared to 15000us. This however is balanced by the fact that there were over 30k method calls compared to 255 for LUFact.cl example.

The solver routine was not itself optimised for parallel execution but did make use of the parallel daxpyCLND routine which had implemented data parallel execution. This is demonstrated in comparison of LUfactPar.cl solveCLND compared against LUFact.cl dgeslCL execution profiles, solveCLND taking roughly 550us and dgeslCL taking approximately 25000us.

The LUFactPar code still has a number of issues relating to GPGPU execution, currently the code routines will correctly solve an 8x8 matrix, higher values appear to have memory bank access issues. Additionally the decomposition of work is based on the use of a single workgroup / compute unit, this further restricts the upper bounds of execution to around n=250. Values greater than n=250 causes kernel failure with the OpenCL ‘Out_Of_Resource’ error, this is believed to be due to excessive use local memory and would further work on the workgroup decomposition method.

In sizes of a[,] over n=8 we see that the pivot vector ipvt[] has incorrect values, once the wrong value is selected the matrix data is pivoted and scaled incorrectly. This was originally believed to a fault in the idamaxCLND code, in that it appeared from the ipvt vector that the routine had incorrectly calculated the maximum value in the column. After some overly complex and time consuming manual debugging the fault was found not in the idamaxCLND kernel but in the previous row iteration, the correctly calculated columns were not pivoted correctly. This missing column swap and subsequent incorrect scale corrupted the subsequent matrix data. It is possible that given the problem occurs as n equals a power of 2 that again a memory bank access issues might be at fault, there was not further time to investigate.

Although the LUFactPar.cl code gave incorrect results at n>8 it did perform the appropriate calculations and performance would be indicative of that of a correctly operating code, however the upper bounds of n were defined by the kernel failing out of resources at n > ~250. For comparisons purposes these codes will be test with n=250.

During final analysis a third variation of the code was produced, this re-introduced the dgefa routine in OpenCL code as kernel dgefaCLND. This essentially reducing the number of OpenCL calls to O(n), where dgefaCLND executed faster as k tends towards n. This final variation is profiled in Figure 5-9 below.
Figure 5-9 – Profile of LUFactPar.cl using dgefaCLND kernel

Although this has the same shape as the profile of the execution dgefaCL in Figure 5-4 the run time of each dgefa method has dramatically decreased.
6 Results

The execution of the code created and performance metrics garnered over the implementation and testing period of the project, looks to demonstrate the operational capability and suitability for the given test platforms to execute OpenCL based HPC kernel code.

6.1 Methods

There are a number of factors that will influence the performance characteristics for the tests, from physical factors such as PCI Express bus through to other background tasks the host machine may be running. Therefore a number of different machines were tested for compatibility and capability but for stability and reproducibility the main results were taken by using the Ness machine and submitting jobs to the backend job engine. Where GPGPU testing is required the ness-gpu1 backend was used for the execution of OpenCL code but later in the project local machines were used. The new EPCC fermi0 machine was tested but an unresolved issue exists where the creation of a context appears to take several seconds, this appears to be an infrastructural related issue.

The intended tests were each executed with follow problem sizes:

<table>
<thead>
<tr>
<th>n =</th>
<th>Size A</th>
<th>Size B</th>
<th>Size C</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU Factorial</td>
<td>500</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>Sparse Matrix</td>
<td>50000</td>
<td>100000</td>
<td>500000</td>
</tr>
<tr>
<td>FFT</td>
<td>2097152</td>
<td>8388608</td>
<td>16777216</td>
</tr>
</tbody>
</table>

Figure 6-1 – Tests and problem size.

Each code was executed 5 times and the average time to execute being used as the reference time. Full run results are provided in the Appendix B.

Where codes could not executed or had a limiting upper bound factor then all codes were sampled with the maximum values that would work across all codes.

6.1.1 Environment

The following machines were used in development and testing various software configurations:
<table>
<thead>
<tr>
<th>Machine</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Ness</td>
<td>Nyx</td>
<td>SUSE</td>
</tr>
<tr>
<td>CPU</td>
<td>Type</td>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td>2x AMD Opteron</td>
<td>Intel i7 860</td>
<td>AMD Athlon x64</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4 HT</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.6GHz</td>
<td>2.8GHz</td>
<td>2.4GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>Interconnect</td>
<td>Interconnect</td>
<td>Interconnect</td>
</tr>
<tr>
<td></td>
<td>PCIe 2.0</td>
<td>PCIe 2.0</td>
<td>PCIe 1.0</td>
</tr>
<tr>
<td>OS</td>
<td>Linux</td>
<td>Windows 7</td>
<td>Linux</td>
</tr>
<tr>
<td>GPU</td>
<td>Manufacturer</td>
<td>NVidia</td>
<td>NVidia</td>
</tr>
<tr>
<td></td>
<td>Model</td>
<td>Model</td>
<td>Model</td>
</tr>
<tr>
<td></td>
<td>Tesla C1060</td>
<td>GTX 480</td>
<td>FX 5800</td>
</tr>
<tr>
<td></td>
<td>4GB DDR3</td>
<td>1.5GB DDR5</td>
<td>4GB DDR3</td>
</tr>
</tbody>
</table>

**Figure 6-2 – Test hardware**

An Apple Macbook Pro was quickly eliminated from the OpenCL based codes as the software stack worked correctly but the on boards NVidia GPGPU does not support double precision operations or the Apple OpenCL implementation does not support the `cl_khr_fp64` directive.

Latterly in the project the new EPCC tesla 200 series machine became available for access, this had issues with creation of an OpenCL context taking in excess of several seconds, the root cause was not established but it was the only platform presenting these issues.

The software environment to build the codes is listed below with any optimisation directive information.

C Compiler: gcc v4.4.1 was used with the original –fast compiler option replaced with the –O3 optimisation option.

Java Virtual Machine: Sun Java 1.6.0, the –server option was to ensure optimal runtime settings.

Common Language Runtime: Mono v 2.6.7 x64 on Linux and Microsoft .Net 3.5 on windows. Where ever possible default optimisations were used, the mono runtime was built with large array support but without the llvm compiler infrastructure as this was causing the errors in the OpenCL interoperability layer.
6.2 CPU Based Codes

Each of the three benchmark routines were executed on several machines for comparison, with the performance graphs shown in Figure 6-3 below.

The limited tests show similar result to that of the previous studies in 2003, with the native code still outperforming the managed code. Overall there is much less discrepancy between the managed runtimes.

Figure 6-3 – CPU execution of Java Grande Section 2 kernel codes, LU Factorial, Sparse Matrix Multiply and Fast Fourier Transformation
6.3 OpenCL Based Codes

The porting of the OpenCL kernels was more involved that at first planned, as discussed in section 5.2.4 above there were a number of pain points in terms of creating and verifying OpenCL GPGPU code. This led to the selection of a common testable problem size of n=250.

The OpenCL codes LUFact.cl and LUFactPar.cl were executed from both C and C# host code, with n=250. The following results were established:

<table>
<thead>
<tr>
<th>n=250</th>
<th>CPU</th>
<th>LUFact.cl</th>
<th>LUFactPar.cl</th>
<th>dgefaCLND</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>346</td>
<td>1.592</td>
<td>10.034</td>
<td>46</td>
</tr>
<tr>
<td>C#</td>
<td>165.2</td>
<td>5.2</td>
<td>16.308</td>
<td>212</td>
</tr>
<tr>
<td>Java</td>
<td>174</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-4 – Performance in MFlops of CPU and OpenCL codes n=250

The LUFact.cl from C showed a significant drop in performance from the CPU based codes, from the initial naïve code which failed to run. This was still lower than expected but verification from the sum of profiled kernel times in around 6 seconds compared to 0.5s for the CPU demonstrated that both the dgefaCL and dgeslCL running as single threads were the root cause of the performance demonstrated.

The LUFactPar.cl code from both host codes gave a similar performance, still less than that of CPU codes. From the original kernel profiles in Figure 5-4 and Figure 5-8 we can see that instead of a few long calls we execute many short calls. This will incur additional costs in terms of OpenCL and managed language overheads. Interestingly the manage C# version was consistently faster that native C version by around 50%, the runtime may have been performing native interoperability optimisations based on the large number of calls.

Investigation into the profile of the host code, in Figure 6-5 below, data confirmed that the LUFactPar.cl code execute 31624 kernel executions. This is represented in the host code profile in Figure 6-6. Of those, 31623 calls being to idamaxCLND, scaleCLND and elimCLND executing at roughly 3us each plus a final solveCLND taking 551us, as in Figure 6-7. This combined code execution is only around 96ms out of total of around 635ms, kernel runtime from the C# profiler demonstrated that the code was now being constrained by the OpenCL calling mechanism. A simple calculation of total kernel time – profiled kernel execution time leaves us the overhead time of 540ms which for 31624 calls gives an individual EnqueueNDRangeKernel call a timing of 0.01ms or 10us which is roughly 3 times longer than the card takes to execute it.
The insight towards the balance of OpenCL calls and execution method informed the modification of the LUFactPar.cl code to include the `dgefaCLND` kernel. This...
reduction in overheads indeed produced improvements with the C version now executing 46MFlops and the C# version pulling 212 MFlops. This is due to both the reduced kernel parallel execution times as well as the lower call overheads. To further investigate the scalability of the code it was found that the dgefaCLND code remained stable up to a value of n=750 on some cards. The in following Figure 6-8, the demonstrated performance of the dgefaCLND based code for a number of sizes of n.

<table>
<thead>
<tr>
<th>n =</th>
<th>250</th>
<th>500</th>
<th>750</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFlops - CPU</td>
<td>165.2</td>
<td>217.4</td>
<td>220.8</td>
</tr>
<tr>
<td>MFlops - OpenCL</td>
<td>212.76</td>
<td>363.89</td>
<td>459.998</td>
</tr>
<tr>
<td>Speed Up</td>
<td>x1.2</td>
<td>x 1.6</td>
<td>x 2.08</td>
</tr>
</tbody>
</table>

**Figure 6-8 - Performance of C# implementation of dgefaCLND based LU Factorisation code for variable n.**

When graphed we can see an increase in performance as in Figure 6-9, this will be due to Gustasfon’s law where the increased workload has offset the invocation overheads. Given an OpenCL code that scales for larger problem sizes we would reasonably expect that similar increase could be realised until physical limitations such as memory bandwidth and accelerator throughput were reached.

**Figure 6-9 – Graph of performance improvements as n increases in dgefaCLND based code.**
To determine the execution time of individual sections of code, instrumentation was added to record separately the initialisation, transfer and execution sections of the code as in Figure 6-10.

<table>
<thead>
<tr>
<th>method</th>
<th>#Calls</th>
<th>GPU usec</th>
<th>%GPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>dgefaCLND</td>
<td>749</td>
<td>593475</td>
<td>99.34</td>
</tr>
<tr>
<td>solveCLND</td>
<td>1</td>
<td>2580.77</td>
<td>0.43</td>
</tr>
<tr>
<td>memcpyHtoDasync</td>
<td>3</td>
<td>8.064</td>
<td>0</td>
</tr>
<tr>
<td>memcpyDtoHasync</td>
<td>3</td>
<td>1317.41</td>
<td>0.22</td>
</tr>
</tbody>
</table>

**Figure 6-10 – Console output of kernel run.**

From the profiler information in Figure 6-11, we can see the two computational kernels were invoked the required number of times and that together they accounted for 99.7 of GPU profile time. Further that the host to device and device to host memory copies were insignificant by comparison although they were completed out with the kernel timing phase of execution.

For the C# implementation, the initialisation routines takes 237ms this includes compilation of OpenCL source and memory buffer allocation, the main kernel executes in 613ms. From the OpenCL profiling run, the combined kernel execution time of 596ms is observed, leaving 17ms as the overhead from the 751 method calls in managed runtime and OpenCL stack. This calculates to an instruction overhead of 0.02ms per invocation.
7 Project Work Plan

The work progressed according to plan for the initial phases but soon into the OpenCL GPGPU programming it became apparent that execution and infrastructural considerations would dominate time consumption. A copy of the original project plan is provided in Appendix A.

Phase one of the plan was to implement and benchmark the CPU based codes from three source languages. There were a number of minor issues that initially caused some delays such as verification of the C# sparse matrix code; all issues were resolved within the project phase and results recorded.

Phase two involved the investigation and creation of a number of OpenCL based codes and appropriate host harnesses. While the principals of the requirements were understood the actual implementation details were part of the investigation and as such were subject to flexible resource allocation. This flexibility was exercised and focus was shifted from the creation of many kernel codes to a correct implementation of a single kernel. Specifically the LU Factorisation code was primarily addressed and two major variations were produced, some extensions were added to the second code to improve overall performance. As an investigatory experience the learning curve in programming GPGPUs though OpenCL was much harder than anticipated, the lack of suitable tooling and the abstract documentation left many of the real world details to be discovered through trial and error efforts. There is a number of manufacturer reference materials such as NVidia OpenCL Programming for CUDA\textsuperscript{36} that address the hardware specifics of programming the latest generation of GPGPU hardware.

The process of debugging and fault finding was tedious but only possible through the use of parallel running of the CPU and GPU based codes. In order to examine the state of data within the OpenCL runtime it was necessary to pass additional variable to each kernel and use these additional data sets as temporary debug information. These debug values could then be compared to the values garnered from the CPU code and where appropriate loop sizes could be modified to find the exact point of error.

While not all development options were exercised there was sufficient scenario data to perform a basic comparison of the execution of OpenCL and non OpenCL based codes.

The original pre-project risk register included a number risks and issues that during the execution of the project provided some additional insights. Changes in the standards did occur but had lower impact than anticipated. OpenCL kernels were not as portable between execution environments as would have been preferred. Most significantly the time required to complete the task was considerably more than was anticipated, however the mitigation of review and focusing on core tasks enabled an appropriate conclusion to the piece.
8 Conclusions

In testing the execution of a numeric algorithm in different runtime situations we have shown that performance increases can be gained from using accelerator technology such as GPGPUs. This does however come at some cost, it is implicit that the code must be written such that it can utilise facilities of hardware to work efficiently.

As can be seen from the performance of the CPU codes in Figure 6-3, across the range of algorithms performance was similar in order between the native code and managed versions, but that native code was invariably faster. In a native execution, resources such as cache and processor cycles are applied solely to the problem and not to the virtual runtime, optimisation decisions are made statically at compile time and are fixed. The lack of variance between the managed runtime environments observed in previous studies is likely due to maturity of these environments over the last 7 or so years.

The performance differential was lessened in the case of executing OpenCL based kernels, where the execution of the kernels on external hardware performed at the same rate regardless of host code. The implication of the last column in Figure 6-4 is that the overheads in calling OpenCL from managed code is less than that of native C, this is possibly through runtime optimisation and further analysis would look to identify the weakness in the native implementation. This would require further verification to ensure the results scale for larger size of n, which would present a more computationally intensive execution. Whatever the cost of OpenCL invocation, we can clearly see that a balance must be struck between cost of kernel calls and kernel execution time. Bias towards a few longer running kernels will leader to greater performance, only where device accelerator optimised code is used.

The goal of platform independent code running in truly heterogeneous environments is laudable; there are however, a number of technical problems imposed by physical characteristics of real hardware. Code written for generic hardware architectures must account for device specific limitations this thereby hinders true portable of accelerator code. C and FORTRAN remain the primary languages for development of HPC codes, the inclusion of managed runtimes could be considered for a number of software engineering advantages such as speed of development and interoperability. More over where a computation is performed by committed hardware high-level object oriented languages can simplify the creation of host code.

The OpenCL API and SDK provide an ideal opportunity to encapsulate HPC kernel algorithms for heterogeneous platforms. The platform is still in an early stage of release but clearly there are a number of items requiring redress to align with peer schemes like CUDA such as the capability to develop and debug code. Further OpenCL is designed for an extensible list of devices including multi-core CPUs, it must be realised that a generic approach to programming devices does not eliminate
the need to have a proficient understanding of the actual device architecture. Even when a generic approach can be taken to programming a device class, like a GPGPU, the accelerator specific implementation often limit how generic a code can be. This may be addressable through just-in-time kernel code modification and recompilation but may be better rectified by a standardisation of device architectures.
9 Bibliography


<table>
<thead>
<tr>
<th>ID</th>
<th>Task Name</th>
<th>Duration</th>
<th>Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initiate</td>
<td>3.68 days</td>
<td>Mon 05/04/10</td>
</tr>
<tr>
<td>2</td>
<td>Create documentation pack</td>
<td>1 day</td>
<td>Mon 05/04/10</td>
</tr>
<tr>
<td>3</td>
<td>Verify environment</td>
<td>2.6 days</td>
<td>Tue 06/04/10</td>
</tr>
<tr>
<td>4</td>
<td>Phase 1 - CPU Kernels</td>
<td>25 days</td>
<td>Tue 06/04/10</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>7 days</td>
<td>Wed 07/04/10</td>
</tr>
<tr>
<td>6</td>
<td>Dense Linear Algebra</td>
<td>2 days</td>
<td>Wed 07/04/10</td>
</tr>
<tr>
<td>7</td>
<td>Sparse Matrix</td>
<td>2 days</td>
<td>Fri 09/04/10</td>
</tr>
<tr>
<td>8</td>
<td>Fast Fourier Transforms</td>
<td>2 days</td>
<td>Wed 14/04/10</td>
</tr>
<tr>
<td>9</td>
<td>C#</td>
<td>6 days</td>
<td>Fri 16/04/10</td>
</tr>
<tr>
<td>10</td>
<td>Dense Linear Algebra</td>
<td>2 days</td>
<td>Fri 16/04/10</td>
</tr>
<tr>
<td>11</td>
<td>Sparse Matrix</td>
<td>2 days</td>
<td>Tue 20/04/10</td>
</tr>
<tr>
<td>12</td>
<td>Fast Fourier Transforms</td>
<td>2 days</td>
<td>Thu 22/04/10</td>
</tr>
<tr>
<td>13</td>
<td>Java</td>
<td>6 days</td>
<td>Mon 26/04/10</td>
</tr>
<tr>
<td>14</td>
<td>Dense Linear Algebra</td>
<td>2 days</td>
<td>Mon 26/04/10</td>
</tr>
<tr>
<td>15</td>
<td>Sparse Matrix</td>
<td>2 days</td>
<td>Wed 28/04/10</td>
</tr>
<tr>
<td>16</td>
<td>Fast Fourier Transforms</td>
<td>2 days</td>
<td>Fri 30/04/10</td>
</tr>
<tr>
<td>17</td>
<td>Benchmark</td>
<td>5 days</td>
<td>Tue 04/05/10</td>
</tr>
<tr>
<td>18</td>
<td>Document Phase 1</td>
<td>20 days</td>
<td>Tue 06/04/10</td>
</tr>
<tr>
<td>19</td>
<td>Phase 2 - OpenCL Kernels</td>
<td>33.2 days</td>
<td>Mon 24/05/10</td>
</tr>
<tr>
<td>20</td>
<td>Test Harness creation</td>
<td>7 days</td>
<td>Mon 26/05/10</td>
</tr>
<tr>
<td>21</td>
<td>Kernel</td>
<td>9 days</td>
<td>Thu 03/06/10</td>
</tr>
<tr>
<td>22</td>
<td>Dense Linear Algebra</td>
<td>3 days</td>
<td>Thu 03/06/10</td>
</tr>
<tr>
<td>23</td>
<td>Sparse Matrix</td>
<td>3 days</td>
<td>Mon 07/06/10</td>
</tr>
<tr>
<td>24</td>
<td>Fast Fourier Transforms</td>
<td>3 days</td>
<td>Fri 11/06/10</td>
</tr>
<tr>
<td>25</td>
<td>Benchmark</td>
<td>10 days</td>
<td>Thu 17/06/10</td>
</tr>
<tr>
<td>26</td>
<td>Document Phase 2</td>
<td>33.2 days</td>
<td>Mon 21/05/10</td>
</tr>
<tr>
<td>27</td>
<td>Project Closedown</td>
<td>5 days</td>
<td>Mon 26/07/10</td>
</tr>
<tr>
<td>28</td>
<td>Finalise documentation</td>
<td>5 days</td>
<td>Mon 26/07/10</td>
</tr>
</tbody>
</table>
11 Appendix B – Raw data
### CPU based Codes, C implementation

<table>
<thead>
<tr>
<th>Machine A</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Max</th>
<th>min</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU Fact A</td>
<td>428.7729</td>
<td>431.988</td>
<td>430.1615</td>
<td>431.2037</td>
<td>492.4015</td>
<td>LU Fact A</td>
<td>492.4015</td>
<td>428.7729</td>
</tr>
<tr>
<td>LU Fact B</td>
<td>342.77889</td>
<td>342.9967</td>
<td>342.9983</td>
<td>342.7052</td>
<td>398.1756</td>
<td>LU Fact B</td>
<td>398.1756</td>
<td>342.7052</td>
</tr>
<tr>
<td>LU Fact C</td>
<td>429.4503</td>
<td>369.3584</td>
<td>429.8318</td>
<td>429.5724</td>
<td>369.622</td>
<td>LU Fact C</td>
<td>429.8318</td>
<td>369.3584</td>
</tr>
<tr>
<td>Sparse B</td>
<td>57.2928</td>
<td>65.4047</td>
<td>57.4766</td>
<td>57.337</td>
<td>57.3668</td>
<td>Sparse B</td>
<td>65.4047</td>
<td>57.2928</td>
</tr>
<tr>
<td>FFT A</td>
<td>571388.8701</td>
<td>914444.3833</td>
<td>763723.8178</td>
<td>759798.3147</td>
<td>761203.7108</td>
<td>FFT A</td>
<td>914444.3833</td>
<td>571388.9</td>
</tr>
<tr>
<td>FFT B</td>
<td>616883.6646</td>
<td>743405.5235</td>
<td>742484.9208</td>
<td>617327.0586</td>
<td>616385.9183</td>
<td>FFT B</td>
<td>743405.5235</td>
<td>616385.9</td>
</tr>
<tr>
<td>FFT C</td>
<td>549512.8012</td>
<td>558701.3306</td>
<td>557720.837</td>
<td>557807.2105</td>
<td>557632.507</td>
<td>FFT C</td>
<td>558701.3306</td>
<td>549512.8</td>
</tr>
<tr>
<td>Ness</td>
<td>Run 1</td>
<td>Run 2</td>
<td>Run 3</td>
<td>Run 4</td>
<td>Run 5</td>
<td>Max</td>
<td>Min</td>
<td>Average</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>-------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>(Mflop/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LU Fact A</td>
<td>269.5605</td>
<td>280.3790</td>
<td>268.6966</td>
<td>271.3053</td>
<td>268.6966</td>
<td>280.3790</td>
<td>268.6966</td>
<td>271.7276</td>
</tr>
<tr>
<td>LU Fact B</td>
<td>225.2920</td>
<td>224.2343</td>
<td>252.7085</td>
<td>224.1591</td>
<td>222.8888</td>
<td>252.7085</td>
<td>222.8888</td>
<td>229.8565</td>
</tr>
<tr>
<td>LU Fact C</td>
<td>244.7009</td>
<td>244.1194</td>
<td>244.8692</td>
<td>329.2647</td>
<td>279.9147</td>
<td>329.2647</td>
<td>244.1194</td>
<td>268.5738</td>
</tr>
<tr>
<td>(Iter/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse A</td>
<td>98.6193</td>
<td>107.9331</td>
<td>96.7118</td>
<td>110.6806</td>
<td>112.2334</td>
<td>112.2334</td>
<td>96.7118</td>
<td>105.2356</td>
</tr>
<tr>
<td>Sparse C</td>
<td>4.2720</td>
<td>4.2736</td>
<td>4.2676</td>
<td>4.4832</td>
<td>5.2196</td>
<td>5.2196</td>
<td>4.2676</td>
<td>4.5032</td>
</tr>
<tr>
<td>(Samp/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT A</td>
<td>386857.0300</td>
<td>277327.7000</td>
<td>279285.1200</td>
<td>279396.7500</td>
<td>278247.6000</td>
<td>386857.0300</td>
<td>277327.7000</td>
<td>300222.8400</td>
</tr>
<tr>
<td>FFT B</td>
<td>225730.8000</td>
<td>225579.0500</td>
<td>225360.8800</td>
<td>225840.1900</td>
<td>225821.9500</td>
<td>225840.1900</td>
<td>225360.8800</td>
<td>225666.5740</td>
</tr>
<tr>
<td>FFT C</td>
<td>204495.4500</td>
<td>204855.0200</td>
<td>205087.9000</td>
<td>205125.5200</td>
<td>288958.4400</td>
<td>288958.4400</td>
<td>204495.4500</td>
<td>221704.4660</td>
</tr>
</tbody>
</table>
### CPU based Codes, C# implementation

<table>
<thead>
<tr>
<th>Ness</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Max</th>
<th>min</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Mflop/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LU Fact A</td>
<td>266.140</td>
<td>266.140</td>
<td>266.140</td>
<td>266.140</td>
<td>266.140</td>
<td>266.140</td>
<td>265.300</td>
<td>265.8040</td>
</tr>
<tr>
<td>LU Fact C</td>
<td>249.560</td>
<td>249.560</td>
<td>249.560</td>
<td>249.560</td>
<td>249.560</td>
<td>249.560</td>
<td>248.330</td>
<td>248.9500</td>
</tr>
<tr>
<td>(Iter/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse A</td>
<td>130.290</td>
<td>126.820</td>
<td>132.360</td>
<td>130.460</td>
<td>132.630</td>
<td>132.630</td>
<td>126.820</td>
<td>130.5120</td>
</tr>
<tr>
<td>Sparse C</td>
<td>5.7300</td>
<td>5.7300</td>
<td>5.7300</td>
<td>5.7200</td>
<td>5.7300</td>
<td>5.7300</td>
<td>5.7200</td>
<td>5.7280</td>
</tr>
<tr>
<td>(Samp/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT A</td>
<td>310873.4100</td>
<td>375228.4800</td>
<td>377865.2300</td>
<td>377389.2400</td>
<td>377933.3200</td>
<td>377933.3200</td>
<td>310873.4100</td>
<td>363857.9360</td>
</tr>
<tr>
<td>FFT B</td>
<td>250945.5500</td>
<td>313487.3500</td>
<td>315919.4100</td>
<td>314545.2800</td>
<td>315741.0000</td>
<td>315741.0000</td>
<td>250945.5500</td>
<td>302127.7260</td>
</tr>
<tr>
<td>FFT C</td>
<td>235400.3200</td>
<td>293785.6300</td>
<td>296459.1500</td>
<td>294182.2900</td>
<td>296579.6800</td>
<td>296579.6800</td>
<td>235400.3200</td>
<td>283281.4140</td>
</tr>
</tbody>
</table>
### Various implementations of LU Factorial with n = 250, MFlops

<table>
<thead>
<tr>
<th>Machine</th>
<th>n = 250</th>
<th>C</th>
<th>C#</th>
<th>Java</th>
<th>C ocl LUFact</th>
<th>C ocl LUFactPar</th>
<th>C# ocl LUFact</th>
<th>C# ocl LUFactPar</th>
<th>C ocl dgeslCLND</th>
<th>C# ocl dgeslCLND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>295</td>
<td>184</td>
<td>146</td>
<td>1.592</td>
<td>10.106</td>
<td>5.2</td>
<td>16.32</td>
<td>46.045</td>
<td>215</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>363</td>
<td>159</td>
<td>142</td>
<td>1.592</td>
<td>9.8</td>
<td>5.2</td>
<td>16.14</td>
<td>46.764</td>
<td>215.14</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>367</td>
<td>125</td>
<td>245</td>
<td>1.592</td>
<td>10.09</td>
<td>5.3</td>
<td>16.52</td>
<td>46.757</td>
<td>210.83</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>352</td>
<td>210</td>
<td>191</td>
<td>1.592</td>
<td>10.07</td>
<td>5.3</td>
<td>16.19</td>
<td>46.554</td>
<td>210.83</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>353</td>
<td>148</td>
<td>146</td>
<td>1.592</td>
<td>10.106</td>
<td>5.1</td>
<td>16.37</td>
<td>45.357</td>
<td>212</td>
</tr>
<tr>
<td>max</td>
<td></td>
<td>367</td>
<td>210</td>
<td>245</td>
<td>1.592</td>
<td>10.106</td>
<td>5.3</td>
<td>16.52</td>
<td>46.764</td>
<td>215.14</td>
</tr>
<tr>
<td>min</td>
<td></td>
<td>295</td>
<td>125</td>
<td>142</td>
<td>1.592</td>
<td>9.8</td>
<td>5.1</td>
<td>16.14</td>
<td>45.357</td>
<td>210.83</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td>346</td>
<td>165.2</td>
<td>174</td>
<td>1.592</td>
<td>10.0344</td>
<td>5.22</td>
<td>16.308</td>
<td>46.2954</td>
<td>212.76</td>
</tr>
</tbody>
</table>
Performance of dgefaCLND vs CPU, Machine C

<table>
<thead>
<tr>
<th></th>
<th>n = 250</th>
<th></th>
<th>n = 500</th>
<th></th>
<th>n = 750</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c - ocl</td>
<td>c# - ocl</td>
<td>CPU</td>
<td>c - ocl</td>
<td>c# - ocl</td>
</tr>
<tr>
<td>1</td>
<td>46.045</td>
<td>215</td>
<td>184</td>
<td>364.59</td>
<td>252</td>
</tr>
<tr>
<td>2</td>
<td>46.764</td>
<td>215.14</td>
<td>159</td>
<td>362.91</td>
<td>248</td>
</tr>
<tr>
<td>3</td>
<td>46.757</td>
<td>210.83</td>
<td>125</td>
<td>362.91</td>
<td>199</td>
</tr>
<tr>
<td>4</td>
<td>46.554</td>
<td>210.83</td>
<td>210</td>
<td>367.69</td>
<td>178</td>
</tr>
<tr>
<td>5</td>
<td>45.357</td>
<td>212</td>
<td>148</td>
<td>361.35</td>
<td>210</td>
</tr>
<tr>
<td>max</td>
<td>46.764</td>
<td>215.14</td>
<td>210</td>
<td>367.69</td>
<td>252</td>
</tr>
<tr>
<td>min</td>
<td>45.357</td>
<td>210.83</td>
<td>125</td>
<td>361.35</td>
<td>178</td>
</tr>
<tr>
<td>avg</td>
<td>46.2954</td>
<td>212.76</td>
<td>165.2</td>
<td>363.89</td>
<td>217.4</td>
</tr>
</tbody>
</table>

1.287893  1.673827  2.083324