University of Huddersfield

University of Huddersfield Repository

Dafinoiu, A.A.

Investigation of heterogeneous computing through novel parallel programming platforms

Original Citation

Dafinoiu, A.A. (2016) Investigation of heterogeneous computing through novel parallel programming platforms. Masters thesis, University of Huddersfield.

This version is available at http://eprints.hud.ac.uk/id/eprint/31698/

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

http://eprints.hud.ac.uk/

INVESTIGATION OF HETEROGENEOUS COMPUTING THROUGH NOVEL PARALLEL PROGRAMMING PLATFORMS

ANDREI-ALEXANDRU DAFINOIU

A thesis submitted to the University of Huddersfield
in partial fulfilment of the requirements for
the degree of Masters of Science by Research

Supervised by Dr. Violeta Holmes

The University of Huddersfield

September 2016

Abstract

The computational landscape is dominated by the use of a very high number of CPU resources; this has however provided diminishing returns in recent years, pushing for a paradigm shift in the choice for computational systems.

The following work was aimed at determining the maturity of heterogeneous computer systems in terms of computational performance and their possible integration within High-Performance Computing resources through the use of the OpenCL parallel programming platform.

An introduction is given in the existing hardware architectures targeted by the OpenCL platform, existing literature regarding the integration of heterogeneous systems for computational applications, and the OpenCL platform as a development environment.

A number of applications are developed to benchmark the capabilities of the framework in multi-architecture environments, the results of which show up to 160 times performance gain when targeting GPU architectures, as opposed to CPU, for matrix multiplication algorithms.

Based on this, an extensive test-bench is designed targeting the HTCondor resource pool for a Fast-Fourier Transform application. Results from these machines once again showed a significant performance increase against CPU systems, while also enabling the expansion of the HTCondor system and the uncovering of 30 Teraflops of dormant computing power.

The FPGA architecture is also investigated for its potential in OpenCL computational acceleration, with a focus on the platforms ease of use. It is determined that the framework is mature enough for FPGA application development.

Acknowledgements

I would like to express my thanks to Dr. Violeta Holmes, and the High-Performance Computing Research Group for their wisdom, support, and friendship. Their input has been invaluable to me during my time at the University of Huddersfield, and without them this work would have not been completed on time.

Table of Contents

Final Word Count: 17,945

List of Figures

List of Tables

Chapter 1: Introduction

Generally, the world of computing has accessed two methods of manipulating and interpreting information; firstly, hard-coded designs in the form of ASICs, where data-paths and algorithms are fixed in hardware, resulting in high performance applications that cannot be altered after creation, accomplishing only the task they were designed for. And secondly programmable systems, (meaning CPUs, and more recently GPUs), where algorithms are implemented after production via the use of software, the data path in a programmable system is also fixed, however it implements primitive generics so that it may be used in multiple algorithms, resulting in a higher degree of reusability at the cost of performance. (Altera, 2007)

Fixed implementations like the ASIC are so complex in terms of design and manufacture that the user base for such devices is limited to companies that can afford the time and resource investment. Also with technology moving forward so fast, the overall production time of such a device might end up being longer than the time it takes for new generations of hardware to be created, making ASICs a very niche market. Because of this, the majority of applications use programmable systems based on CPU architecture.

As requirements grow, programmable devices need to improve in order to keep up with the computational demands of users. This is done, generally through three different trends. The first of these trends is the frequency scaling of said systems. However, higher frequencies require higher voltage, making it more difficult to increase frequency without also increasing power consumption. This issue is known as the “power wall” and it refers to the point where increasing frequency would require so much more power that it becomes impractical. (Schaller, 1997)

It can be observed in Figure 1 that CPU clock frequency has not seen any major improvements since 2004, and had actually dropped when vendors decided to embrace multi-core architectures.

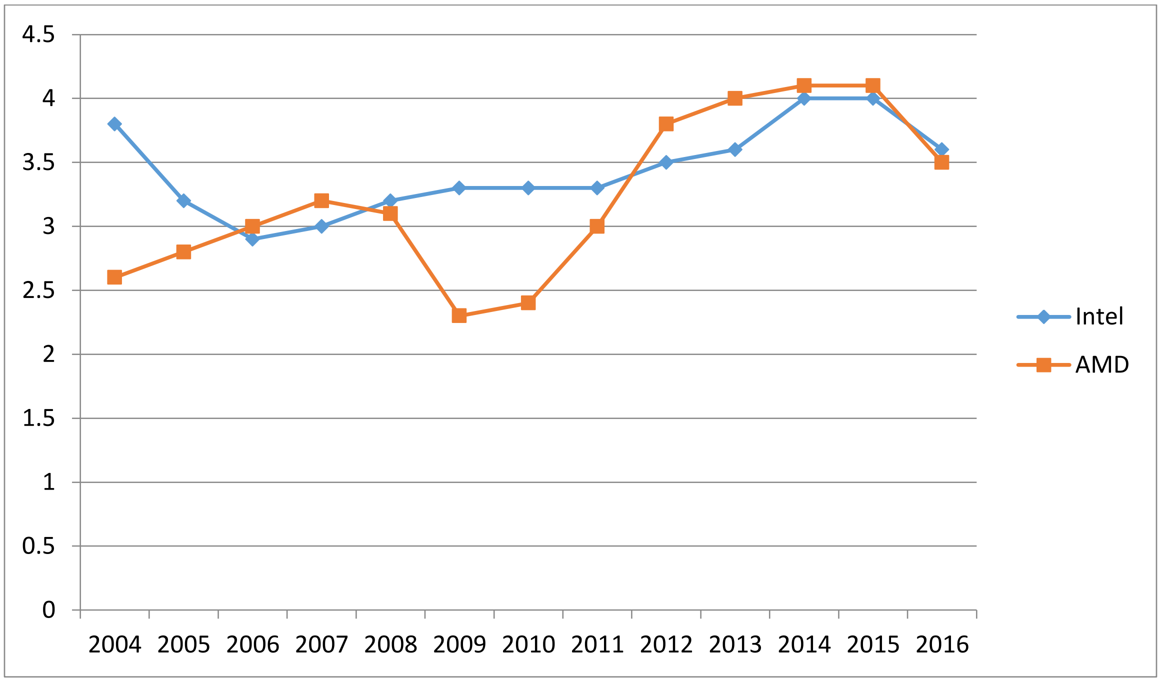


Figure 1: CPU Frequency Evolution (GHz)

A second trend involved reducing the size of transistors in programmable systems, thus increasing the amount of components fitted on the same amount of space. This trend is governed by Moore’s Law, an observation stating that the number of transistors in an IC doubles approximately every two years. The observation has stood the test of time since 1975; however the development pace of this trend is also diminishing, with smaller sized transistors taking increasingly more time to develop. Also, this trend is reaching its physical limits, with current technologies offering 14nm chips. The expected end-date of Moore’s Law is set for 2025.

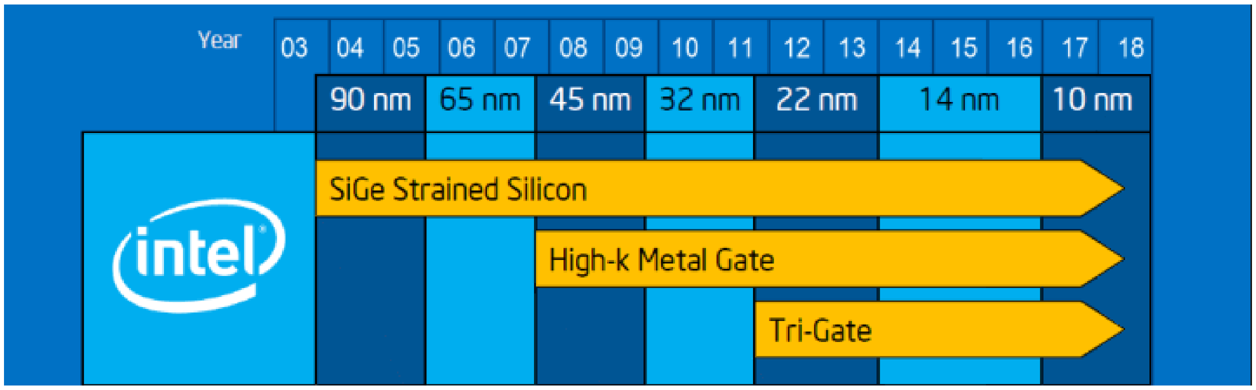


Figure 2: Intel CPU Architecture Size Evolution (Intel)

Taken together, the above two are closely related in limiting CPU frequency, since cooling capabilities are not scaling at a fast enough rate to allow for the maintaining of high frequencies in more dense systems. This has led to a stagnation, and even decrease, in CPU frequency during the last decade.

The third trend used to improve performance relied on the development of more complex hardware, capable of converting the sequential logic of programming into instruction-level parallelism. Also, because, in software programming, the memory latency of programs is not considered, this task falls on hardware once again, meaning that larger chunks of hardware must be dedicated to managing memory, and extracting parallelism from the code. Over time, the improvements to hardware in programmable systems have seen diminishing returns. (Schaller, 1997)

One attempt to avoid the issues described above was the emergence of Multi-Core processing which involves utilizing more compute units running at slower clock speeds and parallelising the process as to exploit multiple nodes at the same time. To offer an example, the

Intel Pentium 570J of 2004 offered a max clock speed of 3.8 GHz and a single core, and the current generation Intel i7-6700k offers a max clock speed of 4.2 GHz with four cores.

Since the benefits of following the three aforementioned trends are diminishing, emphasis is shifted towards creating parallelism at code level instead of relying on hardware to extract it at instruction-level. This means that the developer is tasked with defining parallelism and that the hardware can focus more on the computation and less on interpretation. (Garland & al., 2008)

1.1 Heterogeneous Computing

Any system that uses more than one processor type to handle computational requirement is referred to as a heterogeneous system. The addition of specialized coprocessor to accelerate specific computational tasks as opposed to simply increasing the number of processors is the “definition” of heterogeneous computing. (Kalinov, Lastovetsky, & Robert, 2005)

Heterogeneous systems have found their way into every corner of everyday life, with the CPU-GPU combination being the most common. These are today found in the most so-called “smart” devices, such as phones, tablets or watches. And although devices such as these were not designed with the intent to benefit computing, recent work by the Mont-Blanc project has shown that embedded and mobile devices can be used to power a fully functional supercomputer, with the aim of creating a supercomputing environment that is more energy efficient. (Perez, Bosque, Stafford, & Beivide, 2016)

1.2 The OpenCL heterogeneous framework

OpenCL is a development framework that is platform-independent and emphasises parallel computing. This framework is compatible with many platforms, with commercial suppliers such as Intel, NVidia, and AMD all offering support for OpenCL on their hardware.

OpenCL is a programming language derived from ISO C99 that adds API in order to extract parallelism from an otherwise serial programming language. This allows OpenCL to expand the number of applications that can run on an FPGA, and opening it up to a variety of programmers that had no way of using it before. (Stone, Gohara, & Shi, 2010)

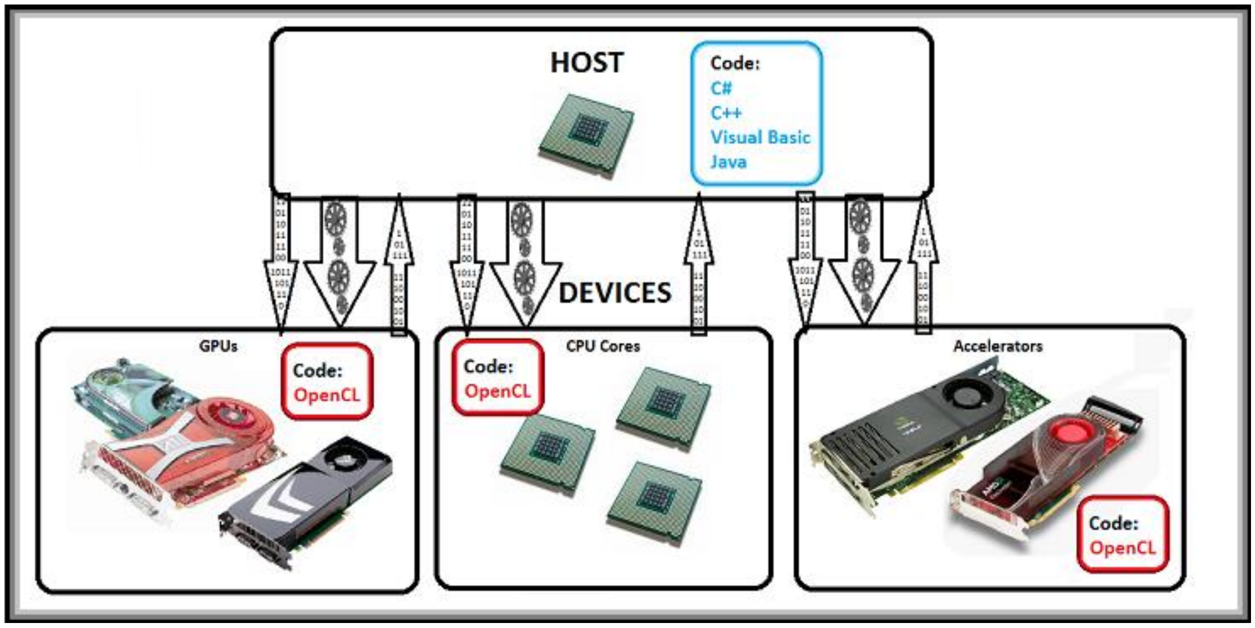


Figure 3: Host and Various Accelerators (CMSoft)

The standard use model for OpenCL is split in two parts:

The Host code; a sequential C code written with OpenCL API required to communicate with the chosen platform. This code is compiled into an executable that gets ran on the host CPU, and is responsible for controlling the entire system, from start to finish.

The Kernels; each function ran on the platform is written as a kernel, using OpenCL syntax. This then gets compiled using the SDK offered by the manufacturer of the platform, generating a second executable. The kernel executable is used by the host to programme the platform during run-time.

1.3 Reconfigurable Computing

The term reconfigurable computing refers to the act of performing computations via the use of spatially (field) programmable architectures such as FPGAs. This merges a multitude of disciplines, including hardware design, digital signal processing, computer aided design, and sequential and parallel computing. Over the past 25 years, a community dedicated to building and programming these new systems has emerged, and the foundation for large scale reconfigurable computing is being laid.

The FPGA became an attractive solution in the computing world because dedicated hardware was always much faster than its software counterpart. However due to the high design cost and development time of ASIC solutions made it viable only for a select few. Moore's Law also meant that in some cases a faster microprocessor was created before the hard-coded solution meant to outperform it was implemented. FPGAs offered similar hardware specific computational speeds without the development and manufacturing costs or lead times of traditional ASIC solutions. (Tessier, Pocek, & DeHon, 2015)

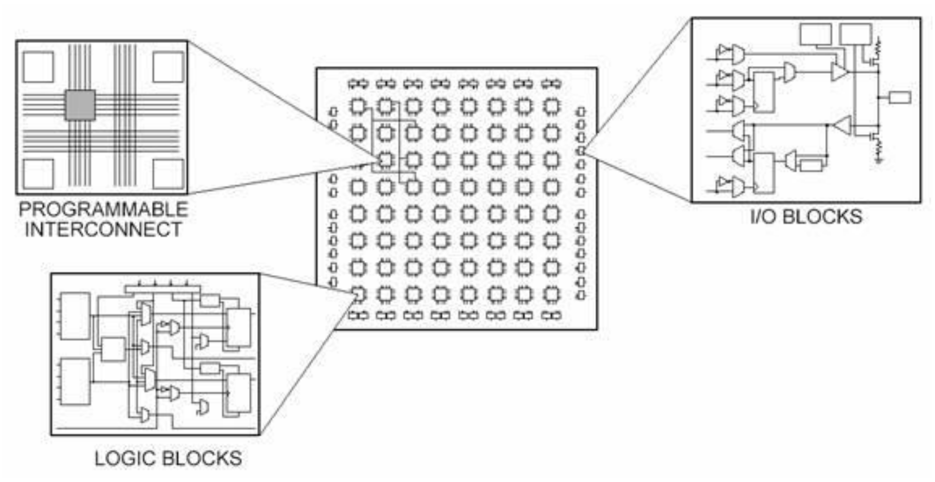


Figure 4: FPGA Basic Outline (Mazsola)

While the FPGA, through its massive parallel computation capabilities, flexibility, and low energy consumption, provides opportunities for computational acceleration, it also comes with an exponential increase in development time. CPUs and GPUs are programmed using high-level languages, C and CUDA as an example, which lowers the development time. In contrast, FPGA are programmed in HDL, which in software development terms, is similar to the Assembler language, a low-level language that makes development more time-consuming. FPGA developers also need to take into account hardware design, RTL programming and timing optimisations. This, in turn, requires domain experience for optimal design and implementation.

In order to alleviate some of the barriers that prevent FPGA based computing to take root multiple tools have been developed to reduce development time by allowing users to write code in high-level languages, such as C or Java, and having it converted into HDL.

It has been shown that FPGAs offer similar computational power to GPUs in regards to optical flow algorithms, however the development time of such applications is 12 times slower on the FPGA than it is on a GPU. The difference in development time was attributed to the complexity of hardware design and simulation which use the process of “edit/compile/simulate followed by edit/synthesize/place-and-route/execute” as opposed to the software development which requires only an “edit/compile/execute” process. However, taking into account the lower power consumption of the FPGA and its higher affinity to parallelism, it can be assumed that FPGAs have the ability to perform better than GPUs, however the trade-off in development time make it a more situational solution. As such, the introduction of OpenCL development could bring down the development time of such applications. (Bodily, Nelson, Wei, Lee, & Chase, A Comparison Study on Implementing Optical Flow and Digital Communications on FPGAs and GPUs, 2010)

In recent years, interest has changed from using HDL to HLS, standing for High-Level Synthesis, an approach to producing logic circuits that avoids using HDL when possible. This means that HLS tools convert a software based design to a circuit made up of control logic and data path. Parallelism in HLS is achieved through scheduling; multiple instructions are performed during the same clock cycle. This, however, is not the best approach when using FPGAs. These devices benefit a lot from their ability to manage pipelined applications, however current programming languages, like C, are unable to express pipelining and as such, the full potential of the FPGA is not unlocked. Also, HLS is not traditionally used to create an entire system, only small parts of it; this means that the need for a competent HDL developer is not bypassed.

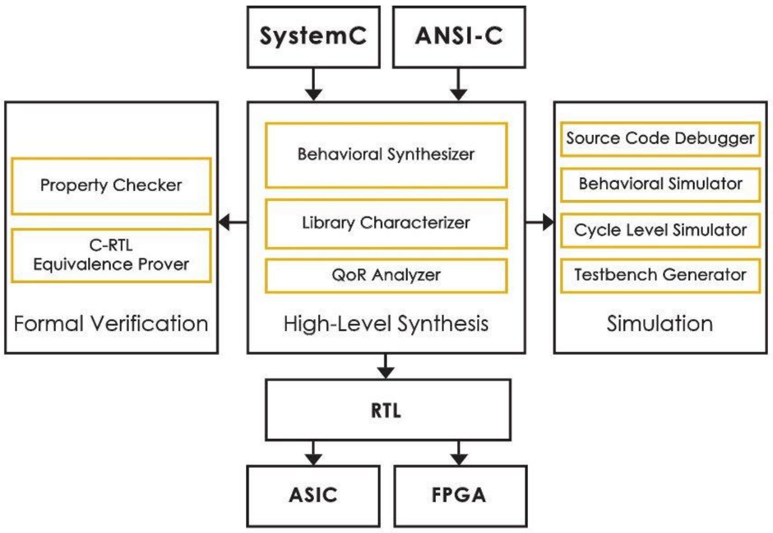


Figure 5: C to RTL Converter Using HLS (Aldec)

OpenCL addresses most of the issues posed by HLS by using a host connected to multiple kernels. Each kernel runs independently of the other and the host manages communications. The host part of the system sets up the data to be processed and runs threads on a kernel. Threads are executed by "…reading arguments, loading data from global memory, processing it, and storing the results in global memory." By controlling the OpenCL application through the host file, the designer is able to avoid going into hardware design, removing the need for experience in that domain and allowing for faster development times.

A recent work by Altera Corporation showed that OpenCL based implementations provided comparable if not better results than the HDL-coded alternatives, with much lower development times. This suggests that OpenCL could allow for the development of high-quality computational solutions based on FPGAs much faster than traditional methods. (Altera, 2011)

1.4 Methodology

To determine the efficiency of heterogeneous computing, using an OpenCL based programming model, several test-benches will be designed. These will be based on algorithms or compute-heavy tasks that can benefit from the increased native parallelism available in heterogeneous systems. Systems, such as these, would allow for the breakdown of operations between resources in order to maximise performance, for example, assigning serial tasks to CPUs and parallel tasks to GPUs.

These implementations will then be compared with computing solutions offered on the existing systems, which utilise CPU based computation, in order to determine whether heterogeneous computing provides a speed-up factor worthy of consideration. Comparison will not be made solely on runtime speed-up but also on development time, development complexity and power-usage.

An attempt will be made to improve the performance of the High Throughput Computing environment at the University of Huddersfield by taking advantage of the readily available General-Purpose GPUs in the HTCondor pool. This diverse ecosystem spans multiple computer architectures, various operating systems, and a significant variation of compute units, varying from low end CPUs to high-end GPGPUs. Currently, the university exploits the idle CPU time of available machines by assigning them computational tasks, however the GPU resources in these systems are unused. No configuration exists to allow for the allocation of tasks to the GPU component of available computers, and as such their capabilities are wasted during their idle periods.

With OpenCL being a platform independent tool, a single implementation is developed to exploit the entire heterogeneous system pool, and HTCondor offers a means to access it. While performance is not inherently sought after in HTC, the ability to accelerate computing without requiring hardware changes, or physical intervention, is still desirable and increases the CPU hours generated by the system. It also enables researchers to use more complex applications that are would normally be too time-consuming when ran on CPUs alone.

A third study will investigate the use of OpenCL for developing FPGA applications aimed at computational applications. The aim is to determine the efficiency of OpenCL design as opposed to traditional HDL design in terms of development time, difficulty of porting applications from CPU to FPGA, and the speed-up obtained when using reconfigurable computing.

1.5 Research Questions:

- Does heterogeneous computing provide enough benefits to warrant a change from traditional systems?
- Is the OpenCL heterogeneous platform mature enough to encourage a shift in development environment used for High-Performance Computing?
- Can FPGAs be used to accelerate computing using the OpenCL platform?

The remainder of this thesis is structured as follows. Chapter II offers insight into the different architectures types that can be exploited for computing purposes, and a review of existing work done in this area. Chapter III introduces the OpenCL platform with a focus on the programming model and usage. Chapter IV covers the implementation of an OpenCL benchmark for use across CPUs and GPUs, with a detailed design process and resulting performance. Chapter V covers the implementation of a different OpenCL benchmark, over a HPC resource composed of hundreds of machines. Chapter VI presents the FPGA related benchmark design and execution, while also discussing the SDK offered by the manufacturer. Chapter VII discusses further research topics in this subject area. Chapter VIII represents the conclusion of this thesis.

Chapter 2: Literature Review

As introduced above, heterogeneous computing refers to the use of multiple types of processors to accelerate the execution of computations within a system. The following pages will detail some of the existing accelerators used in conjunction with CPUs to improve performance.

2.1 Many-Core Architectures

Due to the recent improvements in CPU architectures, the distinct line separating GPUs from CPUs is becoming increasingly blurred. It is due to the emergence of Many-Core architectures that previous boundaries need to be re-evaluated. Many-Core architectures are systems which contain multiple CPU cores within a singular unit, allowing for heavier parallelism at CPU level. This is different from simply connecting multiple CPUs together since it offers much faster memory transfer speeds, and more complex optimisations for parallel execution, at the expense of individual thread performance, and it is in this aspect that Many-Core architectures are similar to GPUs.

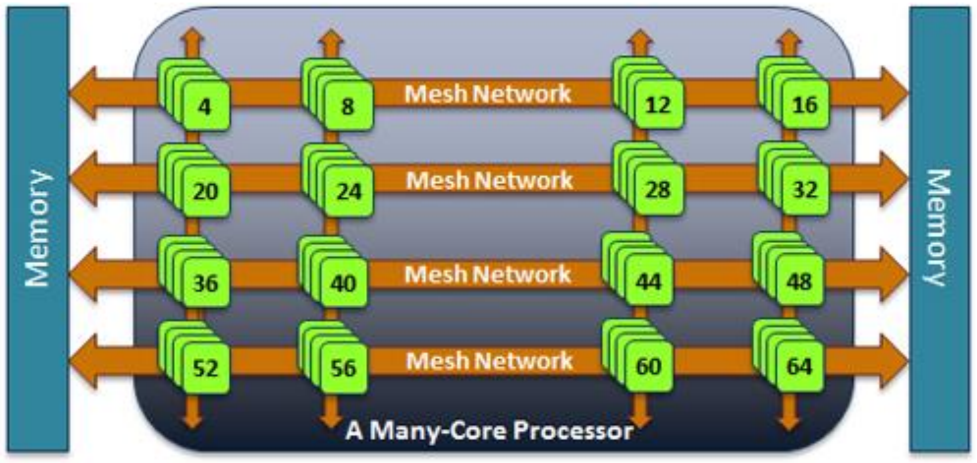


Figure 6: Many-Core Processor architecture (Embedded.com)

One such system is the Intel Xeon Phi, a coprocessor unit comprised of up to 72 specialized CPU cores that can be connected to a computer via a PCI-E bus. The Xeon Phi functions, from a programming perspective, as a CPU. It is fully compatible with existing CPU applications that exploit parallelism. The goal of these sorts of architectures is to offer GPU level parallel performance without the inherent drawbacks of GPU based computational design and programming, or the bottlenecks generated by off-chip data transfers. 23 of the top 500 supercomputers are based on the Xeon Phi architecture, including the former number one supercomputer Tianhe-2, the current fastest supercomputer; Sunway TaihuLight also uses many-core processors with 260 cores per unit. (Top500, 2016)

2.2 Graphical Processing Units

The Graphical Processing Unit (GPU) is a specialized IC designed for rapid manipulation of data, primarily used in computer graphics and image processing. The GPU, as architecture, contains large amounts of parallel processors, which, while unable to match the frequency of a CPU processor, have demonstrated superiority in tasks that involve parallelism, be it data or task parallelism. However, among the major drawbacks of using GPU accelerators are, the difficulty of programming parallel based applications with fundamentally different approaches to solving, and, on a hardware level, the bottleneck resulting from the need to communicate with a host CPU, that results in abysmal performance when there is limited data to be computed. (Owens, 2008)

A GPU processor is specialized in the sense that it is designed with the following considerations in mind:

1. Computational requirements are extensive;
2. Operations are massively parallel;
3. Latency is not as important as throughput;

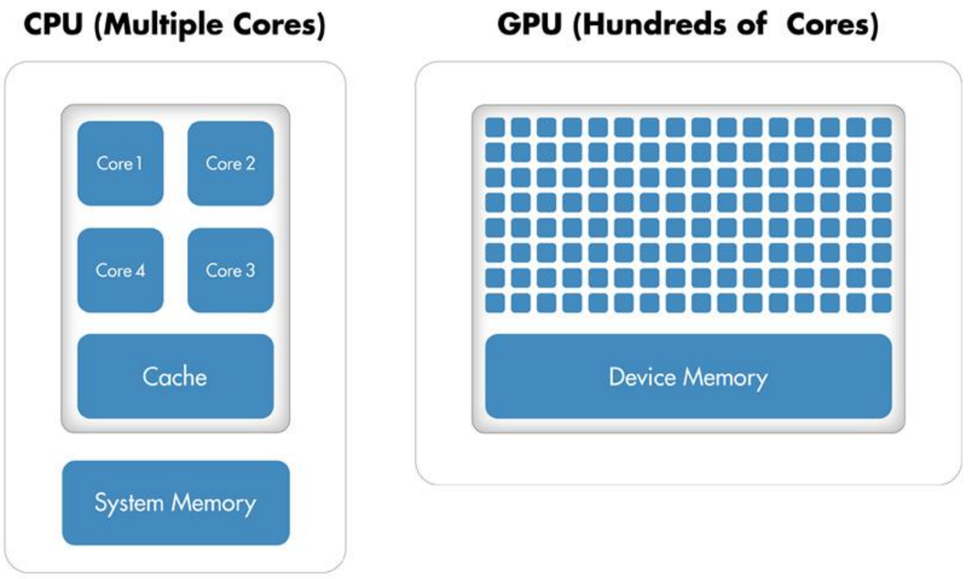


Figure 7: CPU - GPU Core Count

Constant advances in hardware and programming API's for GPUs have led to an explosion of GPU based computations, with 66 of the top 500 supercomputers being fitted with GPU accelerators. (Top500, 2016)

2.3 Field-Programmable Gate Arrays

These devices present a combination of the hardware efficiency found in hard-coded designs and the re-configurability of programmable systems. Initially developed for replacing multiple transistor-transistor logic devices with a single device, the FPGA was used in connecting a micro-controller to peripherals, interfacing devices, or managing memory banks. It was designed as a low-cost prototyping solution, and as such was not considered for computational acceleration. Following the fall in transistor costs, and with it the increase in FPGA power, these devices gained ground in the field of verification, rapid prototyping and also low-volume production where ASIC solutions were deemed impractical.

The ever-growing costs of designing and masking ASICs have led to a higher demand for FPGA solutions, increasing, in turn the interest in developing faster and stronger FPGAs. (Altera, 2007)

2.4 Existing Applications

2.4.1 Radar Processing: FPGAs or GPUs?

A white paper by the ALTERA Corporation that discusses the efficiency of FPGA usage in floating-point operations with regards to their usage in radar systems. The reasoning behind this investigation is that CPUs are unable to keep up the pace with current generation processing requirements, and as such are the significant bottleneck in such systems. (Altera, 2013)

The idea of peak FLOP (Floating-Point Operations per Second) as a measure of performance is discussed and dismissed since it represents an indication of the theoretical maximum capability of the device rather than the actual performance in real-world applications. The article then moves on to show that FPGAs are capable of outperforming GPUs when working with small sized algorithms. One given example is the Fast Fourier Transform (FFT), which in radar systems oscillates in length between 512 and 8,192, in general. In this case GPU solutions are ineffective due to overhead and power usage, with FPGAs offering similar computational speeds. The paper stats that GPUs become efficient solutions for FFTs that are

"[…] several hundred thousand points […]" in length. Based on this, the paper proposes benchmarking solutions based on typical applications.

Following several algorithm based benchmarks, it is concluded that FPGAs can provide lower latency and higher performance than processors; however the advantage of using FPGAs is expected to increase dramatically with the introduction of HPC-optimized FPGAs.

2.4.2 A Comparison Study on Implementing Optical Flow and Digital Communications on FPGAs and GPUs

A study made in (2010) set out to determine the performance of both FPGAs and GPUs in signal, and image processing applications. The article studies raw performance as well as design and development effort for both platforms. (Bodily, Nelson, Wei, Lee, & Chase, A comparison study on implementing optical flow and digital communications on FPGAs and GPUs, 2010)

Implementation of the FPGA system was done using a number of readily available IP cores, which limited the system clock rate, and resulted in raw performance approximately 4 times slower than the GPU solution while also having a much higher development effort. The paper also introduces design enhancements for the FPGA that would, in theory, bring the computational performance to values similar to those generated by the GPU.

The study found that while the GPU solution consumed around 200-300W of power, the FPGA consumption bordered on 10W. This allows for FPGAs to be implemented in embedded systems applications where power constraints exist. In terms of speed, the GPU outperformed the

FPGA, however it required large data block sizes to do so, this in term generated large latency issues that aren't encountered in the FPGA implementation.

The development time was approximated to by 12 times higher for the FPGA than the GPU, due to the difficult nature of debugging HDL based applications.

2.4.3 Performance Comparison of GPU, DSP and FPGA implementations of image processing and computer vision algorithms in embedded systems

A master's thesis from 2013 studied the implementation of template matching on both FPGAs and GPUs for use in embedded, real-time systems. (Fykse, 2013)

Template matching is a process that requires multiple scans of the same image, for different sizes and orientations of the sought object. For this reason the only viable solutions for real-time applications are GPUs and FPGAs, due to their inherent parallelism.

The author chose to implement the solution from scratch on the FPGA and by using an open-source model for the GPU. Details are given on all steps of the design process, and FPGA testing is done in software, through the use of test-benching, with accuracy determined via comparison with a MATHLAB implementation. For the GPU implementation the OpenCV library is used, allowing for fast and straightforward implementation of the desired system.

When compared, from a development effort stand point, the author debates that even with the use of Intellectual Property and HLS, the FPGA development is a lot more complicated than the GPU one. There is mention of OpenCL as a means of facilitating GPU implementations (however, due to the "age" of the paper, OpenCL is not considered for FPGA implementation).

In regards to power consumption, the FPGA far outclasses the GPU, however the FPGA consumption is based on software approximation as hardware testing was not done. Finally, as a pure performance comparison, the GPU performs slightly better than the FPGA at all but the smallest of implementations. The thesis concludes that when faced with real-world projects the higher performance of the GPU must be weighed against the lower power consumption of the FPGA.

2.4.4 Accelerating High-Performance Computing With FPGAs

Published in 2007, this white paper by Altera presents the improvements offered by FPGAs as coprocessors in multiple High-Performance Computing applications. The introduction shows that HPC requirements are increasing at a much faster rate than processors, creating a technology gap. With Moore's Law being outpaced by HPC requirements, the need for specialized coprocessors was introduced. (Altera, 2007)

From a business perspective, higher performance means higher profits (from lower time to market, for example), and as such the need for performance that exceeds Moore's Law is understandable. As processor performance increase is slowing down, and development becomes cost and energy inefficient, application-specific processors are introduced. Ethernet controllers, Graphical processing units and Digital Signal Processors are a few of these solutions; however they are not the answer to the technology gap introduced above, since they only address a single aspect of the problem.

The ideal coprocessor is proposed as providing “specific hardware acceleration for key processes within the application”, being scalable in performance to keep up with demand and having high-bandwidth, low-latency interfacing to the main processor and system memory.

Apart from these, the paper introduces what it calls the” “four Ps” of HPC market needs: performance, productivity, power, and price.” In short, performance refers to the acceleration of the whole system, productivity refers to the ease of configuring the system to run existing software, power refers to the consumption of such systems, which is generally linked to either utilized space or dissipated heat; and finally price, which requires no explanation.

As HPC is shifting away from Massively Parallel Processing toward cluster computing, the coprocessor design needs to be easily integrated into commodity standard architectures “with a cost similar to adding another node in the cluster.”

The FPGA is introduced as a solution that satisfies all “four Ps” of HPC needs. Examples are given of FPGA performance increase of standard CPU architectures ranging from 10x to 360x. From a productivity perspective, compilers that convert C to HDL are introduced, thus removing the need for a user to have prior experience with FPGAs in order to use them. For power, the inherent parallelism of FPGAs allow them to greatly reduce operating time compared to sequential systems, resulting in higher performance at slower clocks, in turn resulting in lower power consumption. The final “P”, price is also covered by the FPGA which has a cost comparable to a CPU of similar specifications.

2.4.5 OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems

Published in 2010 this work is a very thorough introduction to the OpenCL framework, covering the reasoning behind its development, its predecessors and describing the functionality of the platform. (Stone, Gohara, & Shi, 2010)

The shift toward heterogeneous computing created a need for software development frameworks in the form of parallel programming languages and libraries. Several toolkits were developed targeting multi-core processors and GPUs, namely, OpenMP, CUDA, and others. OpenCL is described as an industry standard for parallel computing targeting heterogeneous systems that, unlike its predecessors, targets a vast majority of hardware devices, and offers a unified environment for development.

The paper describes the OpenCL programming model, device management, development facilitating features of the framework, and memory related aspects of programming. OpenCL is described as targeting architectures that have, up to this point, been poorly supported by vendors in terms of programming tools or libraries. Among the targeted architectures of OpenCL, this work enumerates and expands on multi-core CPUs, GPUs and the IBM Cell processor.

The work offers an in-depth description into the implementation of an application used in bio-molecular science, presenting the different speed-up capabilities of the aforementioned architectures.

2.4.6 A Comprehensive Performance Comparison of CUDA and OpenCL

This 2011 conference paper investigates the performance of both CUDA and OpenCL programming platforms for GPU execution of highly parallel algorithms. This work sets out to determine if using OpenCL sacrifices performance for portability, and if so, identify the trade-offs of using OpenCL as opposed to CUDA. (Fang, Varbanescu, & Sips, 2011)

The work focuses on investigating the performance of CUDA and OpenCL applications for 16 different applications from three different benchmark suites. The tests run initially in this work reveal that CUDA outperforms OpenCL in almost all applications by a margin of up to 30%. However, this is due to the lack of optimizations in the OpenCL applications and the much more mature complier in CUDA.

It is further shown that when developing an application in OpenCL rather than porting it from CUDA, equivalent performance is achieved. OpenCL portability is also investigated with the use of an AMD GPU, an Intel CPU and a Cell/BE accelerator. This revealed that GPU performance remains equivalent when porting but CPUs are limited by the small number of available compute cores and accelerators are not mature enough to support most memory requirements.

The paper proposes the creation of an automated application for optimizing OpenCL applications to different hardware devices and platforms.

2.5 Conclusion

It is inferred, based on investigated literature, that no single device, or architecture, is able to outperform the rest in every single aspect of computation, that there is no single fastest device or fastest architecture. These two titles are highly dependent on the task at hand. As such, it is proposed, that rather than being based on a single device, or architecture, a system able to claim the title of fastest computational engine would be comprised of multiple devices and architecture types. It is for this reason that OpenCL, which promises a platform independent framework, for developing applications targeting heterogeneous systems, was selected as the development environment for this work.

Chapter 3: The OpenCL Platform

The Open Computing Language framework is a standard that offers a common environment for developing and executing programs on heterogeneous systems, composed of diverse computational devices, such as CPUs, GPUs, FPGAs, and DSPs. OpenCL was initially developed by Apple, together with other large companies like AMD, IBM, NVIDIA, and Intel, which together formed the Khronos Group. The first public release of the OpenCL standard was in 2009 with OpenCL 1. (Stone, Gohara, & Shi, 2010)

OpenCL provides a set of abstractions and programming APIs designed to allow a developer to easily access multiple hardware architectures and devices. The framework defines both a core set of features available to all compliant devices, such as memory management, target device identification, data transfers, or execution queuing, and a more complex extension mechanism that allows device vendors to expose features unique to individual devices, add additional interfaces, or provide device specific optimizations. In doing this, OpenCL allows users to efficiently port applications between different architectures, without losses in features or accuracy.

The framework can be used to exploit heterogeneous systems by allowing a user to match execution segments to the computational hardware architecture most suited to carry them out. It is up to the developer to decide how to divide the application between the various available compute architectures in order to maximize performance.

3.1 Programming Model

Since OpenCL is a platform-independent programming environment, OpenCL based applications will run on compliant system regardless of available hardware. However, it is up the developer to provision the application in such a way that it will execute on multiple device architectures. For example, an application targeting GPU execution will fail to start unless a GPU device is found within the system. As such, the user is expected to design the application with regards to the system it will be executed on. However, this is not the only solution, as the user is also able to design the application with features that allow for device selection, or that prioritize execution on available accelerators. (Stone, Gohara, & Shi, 2010)

In terms of design, the user is expected to define the targeted computational devices, memory allocations, data management and others within the control segment of the application, namely the host file. The flow of operations during both design and execution can be divided into 5 distinct sections, as evidenced below.

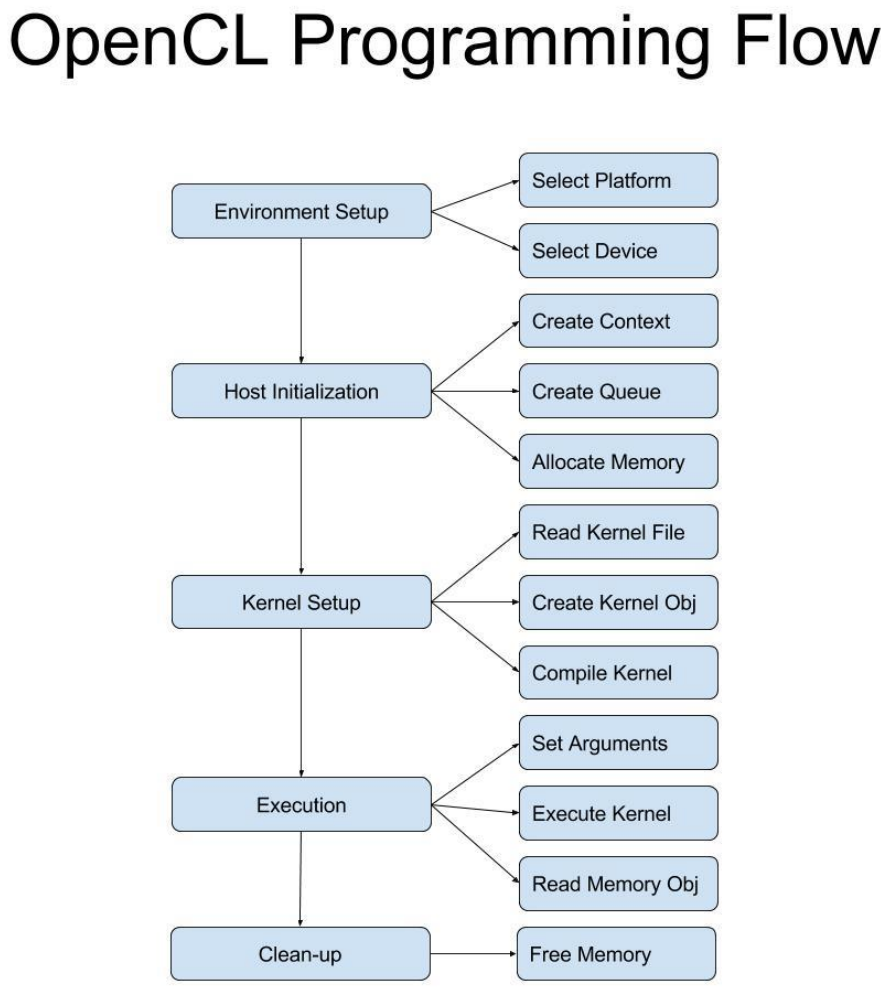


Figure 8: OpenCL Programming Flow

3.1.1 Environment Setup

- Identifying a platform;

A platform is composed of a single host and one or more OpenCL compliant devices. A single computer may have multiple platforms, generally sharing the same host (unless multiple CPUs are available), with each platform being linked to a different OpenCL implementation.

- Select device;

The device is the component that will run computation; multiple types can be called (CPUs, GPUs, FPGAs, or other accelerators). With the inherent heterogeneity of OpenCL, a variety of devices may be available at runtime. The application can be designed with a specific architecture in mind, or setup in a way that allows the host to pick which of the available architectures is the fastest.

3.1.2 Host Initialization

- Create Context;

The OpenCL context, created based on selected platform and devices, manages the objects and resources available to the environment, where objects are allocations that enable communication between the host and the compute devices, and allow for management of memory, command queues, objects and execution. A context may contain one or more devices of the same platform.

- Create command queue;

The command queue is the means through which the host sends commands to the device, with each device requiring its own command queue. Commands include device memory allocations, data transfers, kernel executions, and profiling. Commands are queued in the order that they are coded in the program but can be executed out-of-order by flagging them for asynchronous execution.

- Create memory objects;

Memory objects are blocks of OpenCL data that can be transferred between host and device. A kernel executing on a device is only able to access data stored within the memory of said device, for this reason the device needs to allocate memory to an object where this data can be stored. Memory allocations can only be created and managed by the host. A memory object, thus, allows the host to access a chuck of memory on the device.

3.1.3 Kernel Setup

- Read kernel file;

The code executed on the computational device is contained as a separate entity, written in a manner that exploits parallelism, using OpenCL specific functions. The host executes this kernel and as such must first read it into memory, and where needed, compile the kernel for device execution. There are multiple ways to pass a kernel file to the host, such as reading it in from an external file or reading in a precompiled binary.

- Create program object;

The program object contains the source or binary for the kernels, a built executable, along with the information required to compile the executable at run time, and the list of devices compatible with the program. Program objects may be created with precompiled kernel binaries or with source codes. Precompiled kernels allow for much faster runtime setup however it limits cross-device compatibility.

- Compile kernel;

This optional step creates a binary file for the program object from the source code at runtime, where the precompiled binaries have not been provided within the application, while reducing setup time for execution, this allows an application to target multiple devices without impacting program size or development time.

- Create kernel object;

Based on the program object, a kernel object is instantiated, each containing a kernel function and the argument values used in said function.

3.1.4 Execution

- Set kernel arguments;

As the name suggests, this step handles the arguments passed to the kernels, for examples this could be memory size limits, and pointers to the values used in the function. This is done since the host must handle all calls, queues, and executions.

- Execute kernel (Enqueue task);

In order to execute a kernel on the compute device, it needs to be queued in the command queue, and as mentioned before, this can be done either synchronously, in which case commands are executed in order or asynchronously, where commands are executed independently of one-another.

- Read memory object;

After execution, data from the device memory object must be read back into the host. This can be synchronous (kernel execution is stopped during data transfer) or asynchronous (device keeps computing while data is being transferred).

3.1.5 Clean-up

- Free objects;

After all kernels are executed, the host must free the memory objects it has created, or risk crashing the application once device memory has been filled.

3.2 On-going Improvements

OpenCL 2.0 has recently introduced major improvements to the standard. As the environment matures, more and more features are added to the APIs. In newer releases of the OpenCL standard, a couple of features stand out due to the improvements they bring to not only the capabilities of the application, but also the reduction of design complexity.

3.2.1 Shared Virtual Memory

The first of two major improvements brought forth in OpenCL 2.0 is the addition of shared virtual memory. Before its existence, the user had to manage host memory, device memory and communication between the two; this took up a lot of time in design and space in programming. (AMD, 2014)

With the introduction of shared virtual memory, this management is no longer required, there is no need to track buffers and copy information from one point to the other. Shared pointers have been introduced to fix this exact issue.

OpenCL 2.0 introduces two different types of shared memory: Coarse-grain SVM and Fine-grain SVM.

These two types are differentiated by the synchronization points used in updating the buffers, with coarse-grain being updated when the buffers are called, when the kernel is launched, and when it finishes its operation, and fine-grain including the same synchronization points but also at atomic operations. Atomic operations are those operations that are completed in a single time step, relative to other operations, meaning that no other thread can observe an atomic operations execution. The operation is thus indivisible and irreducible, so it can appear to the system as if happening instantaneously.

Coarse-grain only offers a small benefit to programming as it removes the need for individual calls to buffers, but the real improvement can be seen in the (not yet hardware supported) fine-grain SVM, because using this system, buffer mapping/unmapping is no longer required and since buffers update more often, the system can be altered to use data prior to a kernel finishing its main operation.

3.2.2 Device Kernel Enqueue

With OpenCL 2.0 the device is now able to enqueue kernels, without having to communicate with the host programme. Together with the pipe system, which allows for kernels to exchange data between them, the system will be able to run at much faster speeds, effectively removing the current bottlenecks constituted by device-host communication speeds. (AMD, 2014)

With kernels given to ability to create new kernels without the use of the host programme, new possibilities arise, where an algorithm can adapt itself without having to transfer data back and forth with the host, limiting device to host communication, one of the main bottlenecks in such a system, to a minimum. One such example is found in networking, where GPUs and FPGAs can be used for much faster network encryption/decryption. The accelerator is able to manage data inputs and outputs, without relying on the host CPU.

3.2.3 Standard Portable Intermediate Representation (SPIR-V)

SPIR-V is a standard developed by Khronos, the developers of OpenCL, to facilitate application portability and performance. It is a programming language environment, situated between high-level and low-level languages, which allows for the development of standardized applications for OpenCL drivers. This removes the need to integrate high-level language compilers into device drivers, reducing driver complexity, and improves portability across multiple hardware implementations. (Khronos Group, 2016)

SPIR-V is an attempt to remove the need to precompile binaries for each individual hardware device, leading to a much faster runtime compilation and a smaller development effort.

3.3 Conclusion

The use of familiar programming languages and the massive amount of targeted platforms of the OpenCL framework make it a promising solution for developing heterogeneous applications, with a much shorted development cycle and an increased resilience to aging. The ability to easily alter an application so that it targets a different architecture, the ability to increase performance "under-the-hood" via vendor specific optimizations, and the ability to expose features unique to individual devices offer any application developed with the OpenCL framework a much longer lifespan. This also allows for a much faster adoption of newer hardware architectures, without the need to shift to a different development framework, learn a new programming language, and redevelop the application.

Chapter 4: OpenCL Multi-Architecture Application Development

This chapter introduces the utilized test-bench applications developed for testing the efficiency of the OpenCL platform on various workstations containing CPUs and GPUs. The goal was to utilize applications which could operate on different device platforms with minimal changes, and without device specific optimizations, in order to reveal baseline performance, or rather, the worst expected performance of the given systems.

These applications were developed on a Windows based machine using Visual Studio 2013 and the AMD APP SDK version 2.9, chosen based on the specifications of the development machine, although, the choice of development environment did not affect the design of the applications since no device specific optimizations were desired. Applications targeted both the CPU and GPU architecture either in the same package or as separate instantiations of the same application.

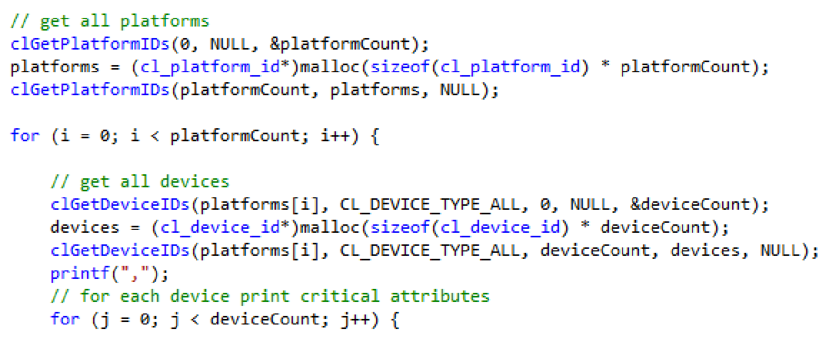
In order to test the usability, efficiency and heterogeneity of the OpenCL framework, a benchmarking system was designed based on applications that could exploit the use of massively parallel hardware architectures offered by specialized architectures.

CPU execution was aimed at providing a comparison baseline for all further testing. GPU execution, aimed at both AMD and NVIDIA devices was chosen because the GPU is the most widely available accelerator available. The FPGA was chosen as the second targeted accelerator architecture for OpenCL execution in order to assess both the effectiveness of FPGA based computing for engineering applications and the duration and complexity of OpenCL based designs targeting the FPGA architecture; however this is covered in a separate chapter.

4.1 OpenCL System Detection

An application was designed to poll the system for compliant OpenCL devices and list their respective features, including core count, clock speed, and maximum memory allocation size. This application lacks OpenCL device specific functionality and thus can report if a system has OpenCL drivers installed or nor and following that what OpenCL devices are identified.

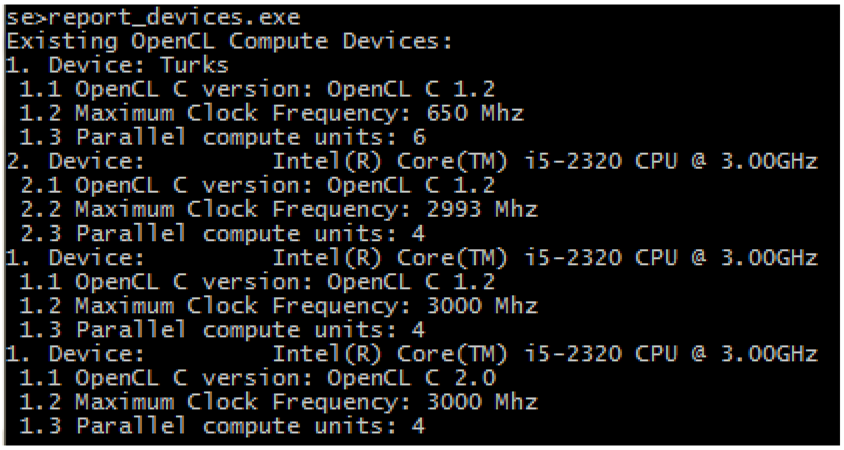
By using this setup it can easily be determined if a system is able to run OpenCL applications or not, and if not, whether the issue is related to available hardware or missing software drivers.



```
// get all platforms
clGetPlatformIDs(0, NULL, &platformCount);
platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id) * platformCount);
clGetPlatformIDs(platformCount, platforms, NULL);

for (i = 0; i < platformCount; i++) {

    // get all devices
    clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, 0, NULL, &deviceCount);
    devices = (cl_device_id*)malloc(sizeof(cl_device_id) * deviceCount);
    clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, deviceCount, devices, NULL);
    printf(",");
    // for each device print critical attributes
    for (j = 0; j < deviceCount; j++) {
```

Figure 9: Excerpt of Device Detection Application

As see in Figure 9 the application reads all OpenCL platforms, and for each, queries every available device for information. Memory sizes for buffers holding output data are calculated right before data acquisition. This is because the number of platforms and devices is unknown at the design stage and as such pre-allocating memory becomes difficult. The CL_DEVICE_TYPE_ALL parameter ensures that all OpenCL compliant devices are called, and can be altered so that the application only reports CPU, or GPU, or accelerator devices.



```
se>report_devices.exe
Existing OpenCL Compute Devices:
1. Device: Turks
 1.1 OpenCL C version: OpenCL C 1.2
 1.2 Maximum Clock Frequency: 650 Mhz
 1.3 Parallel compute units: 6
2. Device:         Intel(R) Core(TM) i5-2320 CPU @ 3.00GHz
 2.1 OpenCL C version: OpenCL C 1.2
 2.2 Maximum Clock Frequency: 2993 Mhz
 2.3 Parallel compute units: 4
1. Device:         Intel(R) Core(TM) i5-2320 CPU @ 3.00GHz
 1.1 OpenCL C version: OpenCL C 1.2
 1.2 Maximum Clock Frequency: 3000 Mhz
 1.3 Parallel compute units: 4
1. Device:         Intel(R) Core(TM) i5-2320 CPU @ 3.00GHz
 1.1 OpenCL C version: OpenCL C 2.0
 1.2 Maximum Clock Frequency: 3000 Mhz
 1.3 Parallel compute units: 4
```

Figure 10: Device Report for Development PC

The application was designed in two variations, regular and basic output. The regular output reported the most important features of the scanned devices, information that is helpful in determining the performance of the device and certain design parameters, such as maximum workgroup size or memory allocations.

The basic output variation of the application simply returns the device name and compiler version, and is designed to be executed in conjunction with the other applications, to identify the targeted device.

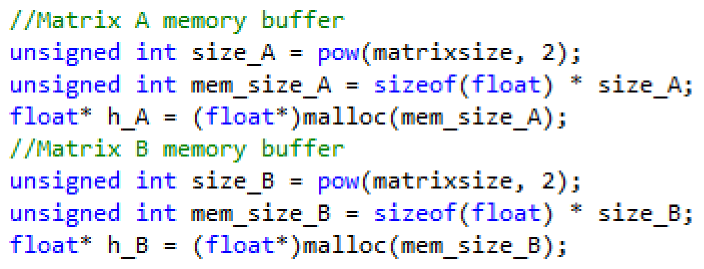
4.2 Application Design

This application used a basic, non-optimized matrix multiplication operation using two same-sized matrixes populated with random data at runtime, and reported execution time using OpenCL profiling tools by measuring duration between start of computation until end of data transfer from compute device to host. This is done to account for the communication overhead generated by different workgroup sizes, and delays in transfer caused by slower bus speeds for GPU to CPU communication.

Multiple matrix sizes were employed, ranging from 2^8 up to 2^12, on multiple workgroup sizes, namely 64, 256, and, where available 1024. The automated allocation of a workgroup size at runtime by the compiler was also utilized, by passing the argument 0 to the workgroup size, allowing the application to determine the best size allocation.

4.2.1 Host Code

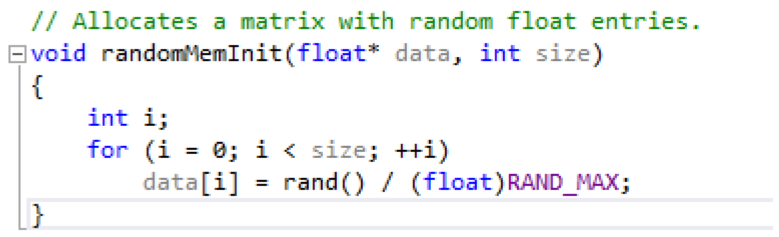
The development process began with allocating the memory buffers that will hold the compute elements and resulting data. This is done by determining the size of the matrixes based on the number of elements, as seen in Figure 11. Because the application is designed with square matrixes in mind, number of elements is determined by squaring the number of rows/columns. With the matrix elements being of type float, the necessary memory can easily be determined using the "sizeof" function.



```
//Matrix A memory buffer
unsigned int size_A = pow(matrixsize, 2);
unsigned int mem_size_A = sizeof(float) * size_A;
float* h_A = (float*)malloc(mem_size_A);
//Matrix B memory buffer
unsigned int size_B = pow(matrixsize, 2);
unsigned int mem_size_B = sizeof(float) * size_B;
float* h_B = (float*)malloc(mem_size_B);
```

Figure 11: Memory Buffer Allocation

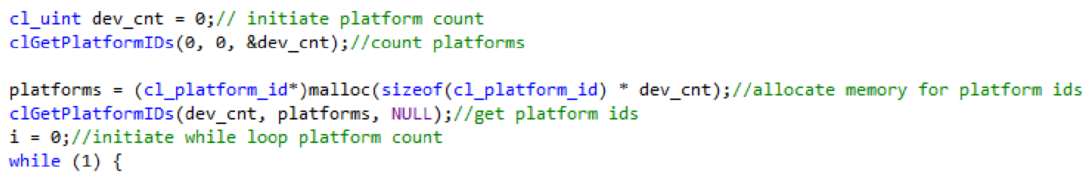
The memory buffers are filled with randomly generated numbers based on a predefined seed making use of C's rand function. Figure 12 illustrates the basic function employed in element allocation.



```
// Allocates a matrix with random float entries.
void randomMemInit(float* data, int size)
{
    int i;
    for (i = 0; i < size; ++i)
        data[i] = rand() / (float)RAND_MAX;
}
```

Figure 12: Function for Matrix Element Allocation

The first step in the aforementioned OpenCL flow is the allocation of a compute platform. In this case, the application allows the user to determine which platform to use for the computation; this is done as a target system may have multiple OpenCL implementations or different compute devices.

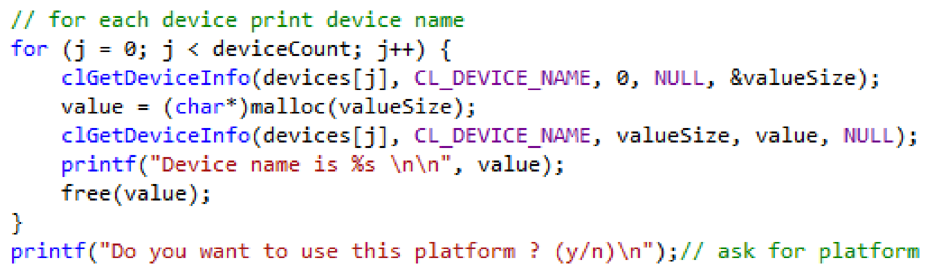


```
cl_uint dev_cnt = 0;// initiate platform count
clGetPlatformIDs(0, 0, &dev_cnt);//count platforms

platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id) * dev_cnt);//allocate memory for platform ids
clGetPlatformIDs(dev_cnt, platforms, NULL);//get platform ids
i = 0;//initiate while loop platform count
while (1) {
```

Figure 13: Identifying Available Platforms.

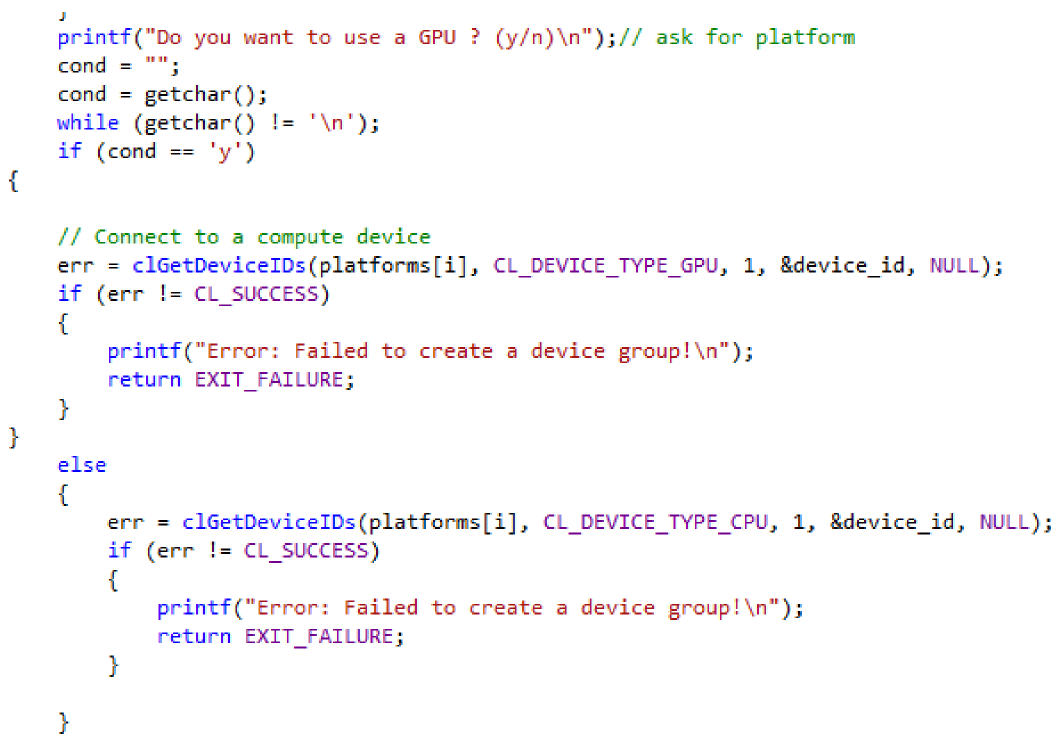
As such, the application must first determine how many platforms are available, allocate memory for them, and finally store platform information in memory, as seen in Figure 13. A while loop is created past this that enables a user to pick a targeted platform based on the compute devices existing within the platform, Figure 14. The user is then asked to pick between targeting a CPU device on the platform or a GPU device, Figure 15.

Note: The application is not optimized to work with platforms that contain multiple devices of the same type and will always pick the first one detected.



```
// for each device print device name
for (j = 0; j < deviceCount; j++) {
    clGetDeviceInfo(devices[j], CL_DEVICE_NAME, 0, NULL, &valueSize);
    value = (char*)malloc(valueSize);
    clGetDeviceInfo(devices[j], CL_DEVICE_NAME, valueSize, value, NULL);
    printf("Device name is %s \n\n", value);
    free(value);
}
printf("Do you want to use this platform ? (y/n)\n");// ask for platform
```

Figure 14: Device Cycle Loop

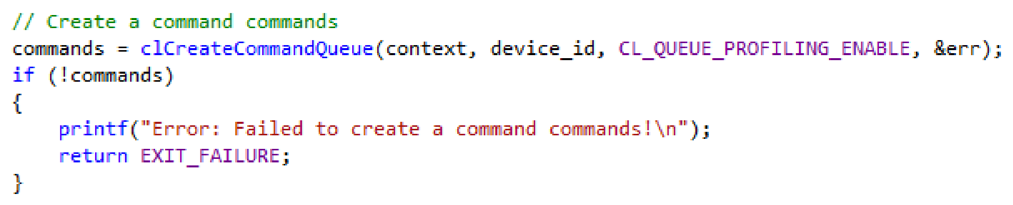


```
    }
    printf("Do you want to use a GPU ? (y/n)\n");// ask for platform
    cond = "";
    cond = getchar();
    while (getchar() != '\n');
    if (cond == 'y')
{

    // Connect to a compute device
    err = clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
    }
}
    else
    {
        err = clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);
        if (err != CL_SUCCESS)
        {
            printf("Error: Failed to create a device group!\n");
            return EXIT_FAILURE;
        }

    }
```

Figure 15: CPU/GPU Decision Point

Once user input has been finished the application creates the OpenCL context based on the chosen device’s ID. It can be noted in Figure 15 that each is called with error checking in place.

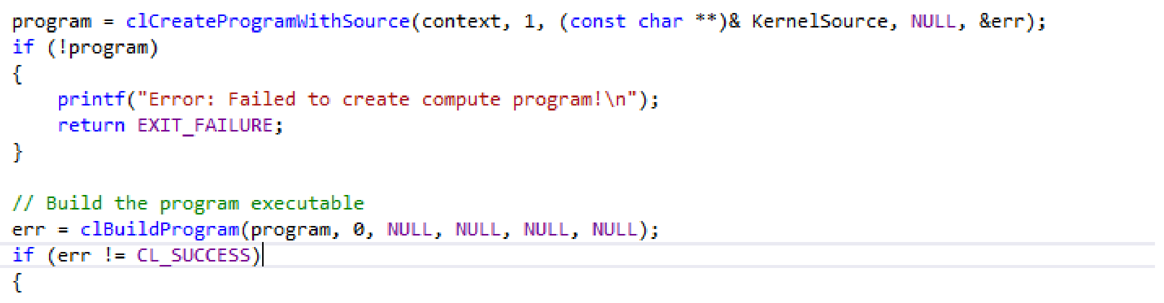
The command queue is initiated, with OpenCL profiling enabled, in order to determine total execution time of kernels, Figure 16.



```
// Create a command commands
commands = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &err);
if (!commands)
{
    printf("Error: Failed to create a command commands!\n");
    return EXIT_FAILURE;
}
```

Figure 16: Command Queue Initialization

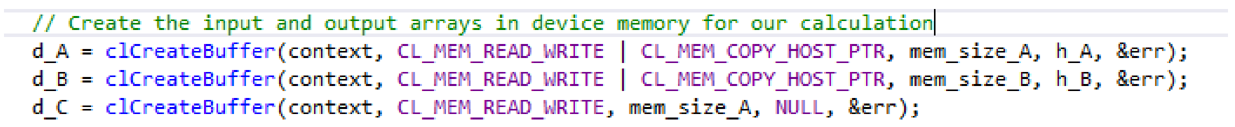
The OpenCL kernel is loaded from a separate file that is read into memory during runtime and then compiled into an executable based on the chosen architecture. This allows for device portability however it does not affect profiling times, Figure 17.



```
program = clCreateProgramWithSource(context, 1, (const char **)& KernelSource, NULL, &err);
if (!program)
{
    printf("Error: Failed to create compute program!\n");
    return EXIT_FAILURE;
}

// Build the program executable
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
```

Figure 17: Building Program Executable

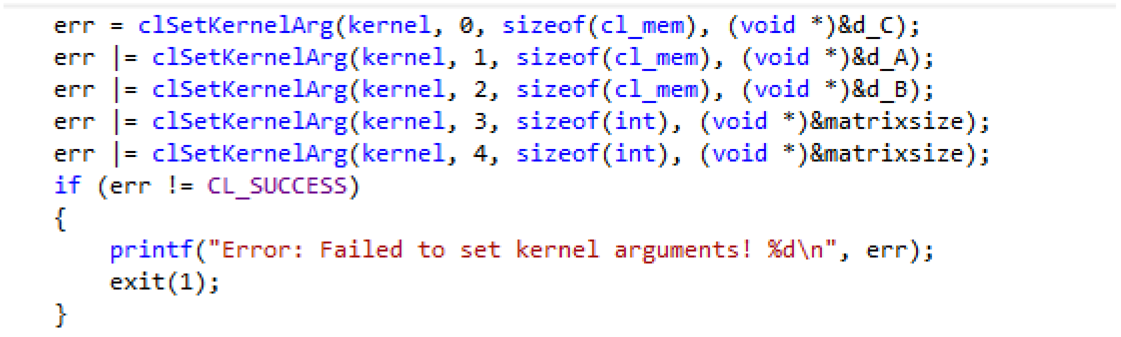
Once the compute kernel is creates, memory must be allocated on the device to contain all three matrixes, the first two are copied from the host, and the first is merely instantiated, as it will contain the result of the matrix multiplication, Figure 18.



```
// Create the input and output arrays in device memory for our calculation
d_A = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, mem_size_A, h_A, &err);
d_B = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, mem_size_B, h_B, &err);
d_C = clCreateBuffer(context, CL_MEM_READ_WRITE, mem_size_A, NULL, &err);
```

Figure 18: Allocating Device Memory

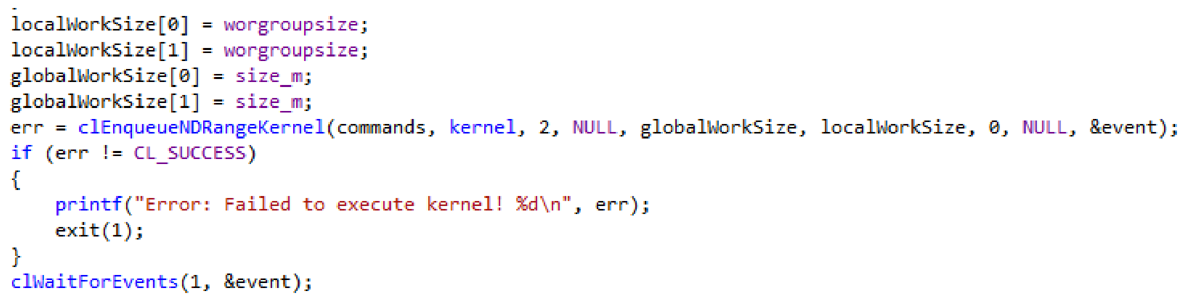
The kernel arguments are then passed; they contain the memory buffers and matrix sizes, Figure 19



```
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&d_C);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&d_A);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&d_B);
err |= clSetKernelArg(kernel, 3, sizeof(int), (void *)&matrixsize);
err |= clSetKernelArg(kernel, 4, sizeof(int), (void *)&matrixsize);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to set kernel arguments! %d\n", err);
    exit(1);
}
```

Figure 19: Kernel Argument Passing

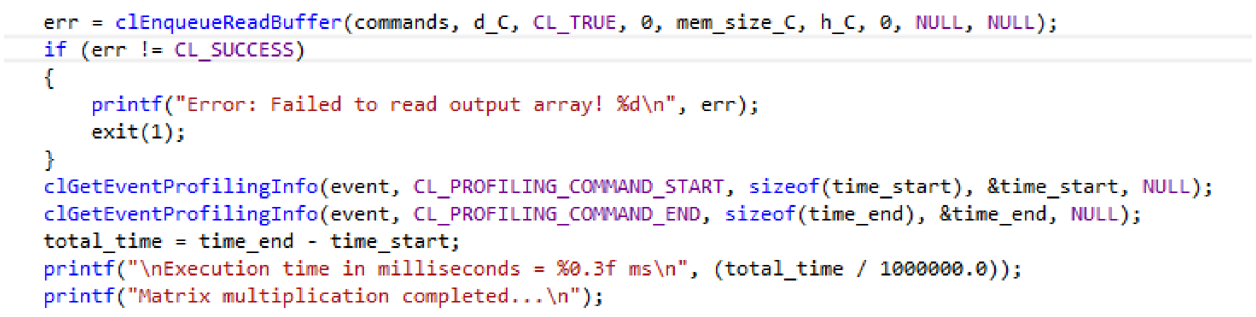
At this stage, Work-group and Work-item sizes are set and the kernel is queued in the command queue for execution, and the application waits for the kernel cu finish, Figure 20.



```
localWorkSize[0] = worgroupsize;
localWorkSize[1] = worgroupsize;
globalWorkSize[0] = size_m;
globalWorkSize[1] = size_m;
err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL, globalWorkSize, localWorkSize, 0, NULL, &event);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to execute kernel! %d\n", err);
    exit(1);
}
clWaitForEvents(1, &event);
```

Figure 20: Command Enqueue

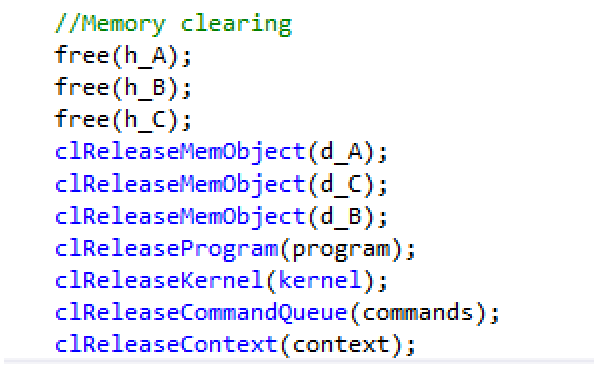
Finally, the contents of the calculated matrix memory buffer are read back into the host and profiling data is called in order to determine execution duration. This duration is calculated using built-in profiling tools offered by the OpenCL framework, and take into account the duration between the first and last command executed by the kernel on the compute device, Figure 21.



```
err = clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0, mem_size_C, h_C, 0, NULL, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to read output array! %d\n", err);
    exit(1);
}
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(time_start), &time_start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end), &time_end, NULL);
total_time = time_end - time_start;
printf("\nExecution time in milliseconds = %0.3f ms\n", (total_time / 1000000.0));
printf("Matrix multiplication completed...\n");
```

Figure 21: Data Retrieval and Profiling

Last but not least, all memory allocations are cleared and the application terminates, Figure 22.

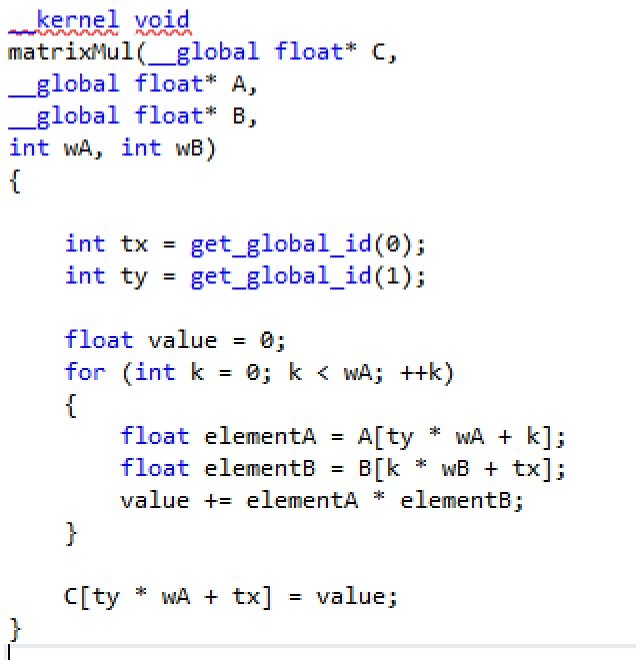


```
//Memory clearing
free(h_A);
free(h_B);
free(h_C);
clReleaseMemObject(d_A);
clReleaseMemObject(d_C);
clReleaseMemObject(d_B);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(commands);
clReleaseContext(context);
```

Figure 22: Memory Clearing

4.2.2 Kernel Code

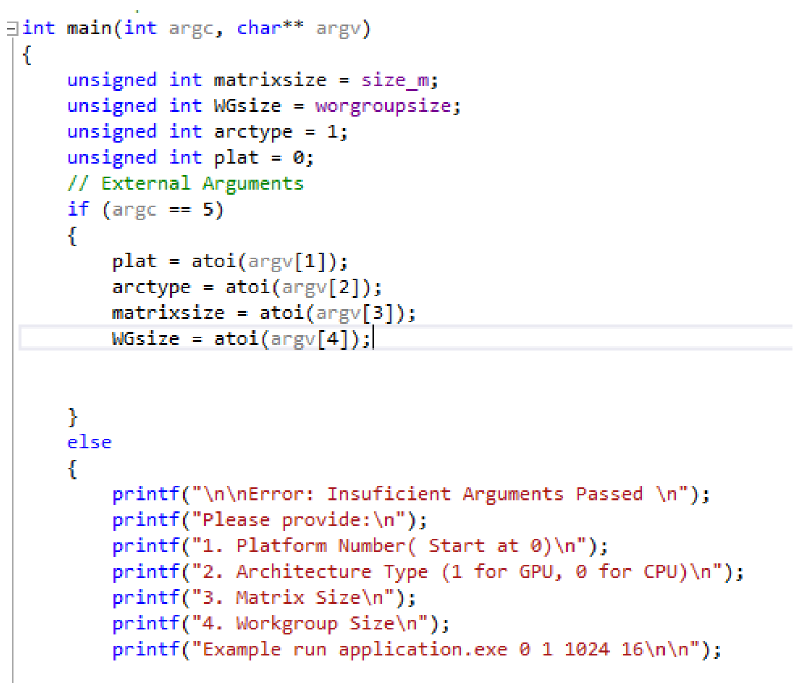
The kernel code is fairly straightforward, it takes in the global buffers containing the two populated matrixes, the buffer containing the output matrix and the number of rows and columns of the matrixes. It defines the two working dimensions using a work-item ID call, “get_global_id”, and based on this information calculated the value of each individual element of the resulting matrix, Figure 23.



```
__kernel void
matrixMul(__global float* C,
__global float* A,
__global float* B,
int wA, int wB)
{

    int tx = get_global_id(0);
    int ty = get_global_id(1);

    float value = 0;
    for (int k = 0; k < wA; ++k)
    {
        float elementA = A[ty * wA + k];
        float elementB = B[k * wB + tx];
        value += elementA * elementB;
    }

    C[ty * wA + tx] = value;
}
```

Figure 23: OpenCL Kernel Code

Later versions of the application used external arguments as opposed to user input in order to facilitate batch execution. The information passed externally was, platform number, device type, matrix size and workgroup size, Figure 24.



```
int main(int argc, char** argv)
{
    unsigned int matrixsize = size_m;
    unsigned int WGsize = worgroupsize;
    unsigned int arctype = 1;
    unsigned int plat = 0;
    // External Arguments
    if (argc == 5)
    {
        plat = atoi(argv[1]);
        arctype = atoi(argv[2]);
        matrixsize = atoi(argv[3]);
        WGsize = atoi(argv[4]);


    }
    else
    {
        printf("\n\nError: Insuficient Arguments Passed \n");
        printf("Please provide:\n");
        printf("1. Platform Number( Start at 0)\n");
        printf("2. Architecture Type (1 for GPU, 0 for CPU)\n");
        printf("3. Matrix Size\n");
        printf("4. Workgroup Size\n");
        printf("Example run application.exe 0 1 1024 16\n\n");
```

Figure 24: External Argument Code Snippet

4.3 Test Bench Environment

This section will cover the execution of the developed test-bench applications for OpenCL on available compute systems in the form of workstations. The main focus is to determine a performance baseline for execution based on standard CPU execution time and compare that against a GPU unit.

4.3.1 System Specifications

Table 1: System Specifications

Specifications	System I	System II	System III	System IV
CPU	Intel i5-2320	Intel i5-2310	Intel i5-3470	Intel i7-3770
Frequency	3.00 GHz	2.90 GHz	3.20 GHz	3.40 GHz
Compute Units	4 threads	4 threads	4 threads	8 threads
Workgroup Size	1024	1024	1024	1024
GPU	AMD HD 6570	AMD HD 6570	NVIDIA 750 TI	AMD HD 6450
Frequency	650 MHz	650 MHz	1.02 GHz	625 MHz
Compute Units	6 SM	6 SM	5 SM	2 SM
Workgroup Size	256	256	1024	256

The number of compute units refers to the amount of processors available to any device, in CPUs this is equal to the number of threads however in GPUs it refers to the Stream Multiprocessors. Stream Multiprocessors consist of multiple stream processors, the specialized processing computational resources used in graphical processing. In the case of AMD GPUs each SM accesses 80 processing elements, while for the NVIDIA each SM contains 128 processing elements. (Asano, Maruyama, & Yamaguchi, 2009)

A workgroup is a collection of computations all executed on a single compute unit. Since each computation unit hands a work-group, increasing the size of these work-groups allows for the exploitation of inherent parallelism at device level and reduces communication overhead at the expense of device memory.

System I was the development computer and had the development kit installed.

System II was chosen as an almost identical system to the 1st however without any specific software installed, to determine if efficiency can be affected by the presence or lack of OpenCL development software.

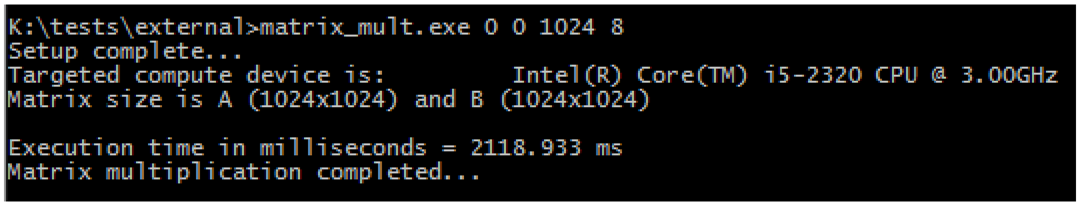
System III was chosen in order to test functionality over a different GPU hardware provider in the form of an NVIDIA GPU.

Systems IV was chosen in order to determine the CPU performance increase for a CPU with twice as many cores as System I which was considered the baseline.

4.4 Application Execution

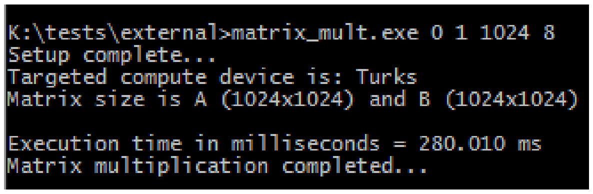
The first tests using OpenCL applications were based on the Matrix Multiplication application described in the previous section. The used application was not optimized for any architecture and featured OpenCL profiling for kernel execution duration reporting. The application was compiled on a Windows machine using Visual Studio 2013 and the AMD APP SDK.

Execution was done using the windows command line interface and later on batch scripts which queued all executions and logged results to a file. This minimized any effect user interaction might have on overall execution time.



```
K:\tests\external>matrix_mult.exe 0 0 1024 8
Setup complete...
Targeted compute device is:         Intel(R) Core(TM) i5-2320 CPU @ 3.00GHz
Matrix size is A (1024x1024) and B (1024x1024)

Execution time in milliseconds = 2118.933 ms
Matrix multiplication completed...
```

Figure 25: CPU Matrix Multiplication



```
K:\tests\external>matrix_mult.exe 0 1 1024 8
Setup complete...
Targeted compute device is: Turks
Matrix size is A (1024x1024) and B (1024x1024)

Execution time in milliseconds = 280.010 ms
Matrix multiplication completed...
```

Figure 26: GPU Matrix Multiplication

Each matrix size/workgroup combination was iterated 5 times, the results logged and averaged below in Table 2, the best performing workgroup size was highlighted in green. The duration was reported in milliseconds, however, due to the large execution times, Table 2 also shows the duration in minutes for the larger sized matrixes.

4.4.1 System I

System I represents the development machine, on which all applications were designed or modified, it includes a suite of development software kits that allow for debugging and monitoring of applications and as such contains 4 different OpenCL platform environments.

Table 2: CPU Execution System I

Execution Time ms	Workgroup			
Matrix Size	0	64	256	1024
512	72.45	62.78	61.75	63.54
1024	2,320.77	2,204.63	2,183.47	2,218.83
2048	33,519.61	28,889.39	28,484.52	35,120.07
4096	(5 m)347,118.99	(4 m)277,252.29	(4 m)263,592.12	(5 m)328,294.72
8192	(47 m)2,766,358.04	(38 m)2,336,417.39	(38 m)2,333,558.53	(45 m)2,729,901.38

Although the GPU in System I operates at a much lower frequency than the CPU, its compute units have access to 80 Stream Processing Units each, for a total of 480 SPUs to be used in algorithmic acceleration. It is worth mentioning that porting this application from CPU to GPU involved the alteration of one argument in the host code. Also, for the GPU implementation, a workgroup size of 1024 could not be allocated as the maximum permitted by local memory is 256.

Table 3: GPU Execution System I

Execution Time ms	Workgroup		
Matrix Size	0	64	256
512	23.67	42.01	25.35
1024	212.89	278.06	141.13
2048	1,711.68	2,209.27	1,110.44
4096	13,732.36	17,709.84	8,917.50
8192	(1.5 m) 106,116.71	(2 m) 142,785.23	(1 m) 71,308.35

It can be easily noted in Table 3 that the GPU outperforms the CPU even at the smallest execution sizes used; however, the speed-up becomes more apparent as it goes from x2 to x32 depending on the number of computed elements. This is both because of the increased number of parallel executions and the reduced impact of data transfers.

This data shows that an entry-level GPU is able to achieve a speed-up of up to 30 times that of its CPU counterpart, where speed-up is proportional to the size of the calculated matrix. Development effort for application porting and performance increase is minimal, however with

GPU specific optimizations for memory usage and transfers higher speed-up values could be obtained.

Automatic workgroup allocation at runtime by the OpenCL compiler leads to the slowest execution on CPU; however this is not the case for GPU implementations. This leads to the conclusion that while automatic allocations is not a good design practice for applications, it can be used in highly heterogeneous systems where applications would otherwise have to be designed with the specifications of the weakest system in mind.

Full CPU benchmarking, with 5 iterations, resulted in around 16 hours of compute time, for the GPU execution using the same parameters the compute time was reduced to around 30 minutes. However, for a more fair comparison, the 1024 workgroup execution would need to be excluded from compute time, leading thus to a duration of approximately 12 hours.

4.4.2 System II

This computer system is one of the workstations available in the University of Huddersfield computer labs. Specification wise it is almost identical to the development unit, however it lacks any form of development kit for OpenCL or similar drivers. However, OpenCL drivers required for execution are available within the basic Intel CPU drivers, and as such benchmarking should not be affected. Also, it is expected that GPU execution will also be guaranteed by Intel drivers.

CPU execution results remained mostly consistent to those from the previous system, with performance being at most 10% slower on System II compared to System I.

GPU execution however failed passed matrix sizes of 1024, leading to the inability of running the full benchmark. When executing the application with any matrix size over 1024, the GPU driver crashed and was restarted by Windows. This reset the GPU while the kernel was still execution, leading to a seemingly unending execution time, Table 4.

This issue was traced to the timer watchdog within Windows, its purpose is to monitor GPU execution and stop any application that appears to be stuck. This is done to ensure that the user does not lose access to the computer if an application gets stuck in an endless loop. Since OpenCL uses the entire GPU during execution, when the GPU used also drives the display, the latter freezes, preventing the user from issuing further commands until the program finishes. According to Microsoft specifications, the default wait time for this watchdog is 2 seconds; however this was not the case for the development machine since it had no issues completing the execution. (Microsoft)

Table 4: GPU Execution System II

Execution Time ms	Workgroup		
Matrix Size	0	64	256
512	19.22	36.61	17.60
1024	175.25	271.39	138.63
2048	1,454.87	2,153.37	1,086.38
4096	13,457.71	17,444.19	8,712.4
8192	103,994.37	141,357.37	70,773.537

Further investigation revealed that System I had a timeout of 180 seconds, while System II did not have a defined register key for the timeout function, thus reverting to the default 2 second value. It is assumed that this registry is created and managed by the GPU drivers on the system, based on specifications offered by the GPU vendor. Since System II did not have any proprietary drivers installed and was running off of the base windows drivers, the register entry increasing the GPU timeout did not exist.

Based on this evidence it becomes obvious that certain alterations are needed in the application design that would ensure host-device communication within the allocated timeout period, as to avoid triggering the timer watchdog.

4.4.3 System III

Following unsuccessful execution on a driverless system, and in order to investigate performance on a different GPU architecture, a NVIDIA based system was chosen as a third target. On this machine, OpenCL fails to identify the GPU device as part of the base platform since NVIDIA does not share compilers with other manufactures, as such, in order to execute OpenCL applications that target NVIDIA GPUs the proprietary GPU drivers need to be installed. Since previous work has shown lack of proprietary drivers prevents complete benchmark execution, they were installed on this system prior to benchmark execution.

Table 5: CPU Execution System III

Execution Time ms	Workgroup			
Matrix Size	0	64	256	1024
512	72.04	74.40	78.35	72.63
1024	2,486.66	2,964.48	2,547.38	2,403.90
2048	26,245.34	27,082.33	25,748.42	24,648.62
4096	253,705.94	247,687.54	244,500.25	241,906.53
8192	2,021,904.59	2,087,273.88	2,164,539.81	2,011,549.21

For the CPU implementation, speed-up compared to the initial test system is once again unnoticeable for all except the highest matrix values. At matrix values of 4096 and 8192 the CPU in System III, which is one generation newer than the previous ones displays a performance increase of up to 20%. This increase is likely due to improvements in data transfer protocols and bus speeds becoming relevant only at large data sizes.

Table 6: GPU Execution System III

Execution Time ms	Workgroup			
Matrix Size	0	64	256	1024
512	3.15	5.39	3.68	3.13
1024	27.41	44.75	30.20	30.20
2048	331.65	349.38	254.96	203.05
4096	3,072.67	2,836.21	2,195.49	1,630.01

8192	25,304.24	23,421.24	17,857.63	13,260.44

The GPU benchmark on System III displayed a performance increase over the GPUs in the previous systems; this performance increase is attributed to the much newer architecture inside the NVIDIA GPU, increased frequency and number of processing units.

The GPU timeout error identified earlier was still present initially on the system, however it could be easily altered within the NVIDIA Nsight control panel, ultimately this issue would be resolved at software level instead of relying on workarounds.

Table 7: GPU Speed-up Against Baseline

GPU speed-up	GPU		
Matrix Size	System I	System II	System III
512	2.4	3.5	19.7
1024	15.5	15.8	72.3
2048	25.7	26.2	140.3
4096	29.6	30.3	161.7
8192	32.7	33	176

Table 7 shows the speed-up obtained by targeting GPU devices instead of CPUs, compared to the CPU baseline on System I, and displays an increase of up to 30 times on a GPU architecture that is part of the same generation as the CPU and upwards of 170 times for a much newer and faster GPU architecture.

4.4.4 System IV

CPU execution on System IV was aimed at determining the performance increase on the CPU when the number of available compute units was doubled; GPU execution was excluded since the device was found to be out-dated.

Table 8: CPU Execution System IV

Execution Time ms	Workgroup			
Matrix Size	0	64	256	1024
512	59.45	62.76	58.22	57.58
1024	1,779.39	1,853.35	1,794.15	1,778.48
2048	16,716.48	17,343.45	16,108.97	16,090.35
4096	193,401.82	194,583.96	187,702.54	187,492.11
8192	(20 m)1,192,011.20	(24 m)1,467,616.96	(25 m)1,490,003.31	(19 m)1,165,134.26

Table 8 shows a number of differences compared to Table 2, a direct comparison shows a performance increase between 20% and 120%, however running a comparison based on the best performing workgroup size of each CPU shows that performance actually varies between 15% on lower matrix sizes, and 80% at higher matrix sizes.

4.5 Conclusions

CPU performance in parallel heavy applications is influenced more heavily by number of available threads than it is by the speed of individual units. GPU performance was shown to increase with computational size due to the decreased impact of overhead in data communication, and the ability to better parallelize execution across more elements.

The OpenCL environment is fragmented, with device vendors implementing platforms that only function on a certain devices and that are based on various versions of OpenCL. This hinders the development of fully heterogeneous systems and suggests that the system is not yet fully mature.

Development effort for architecture porting is minimal for the CPU and GPU architecture for applications that do not feature device specific optimizations. Automated workgroup allocations at runtime are not viable unless the targeting multiple types of device simultaneously.

In order to ensure kernel execution without the interference of the GPU watchdog timer on Windows machines certain modifications must be made within the application to ensure that host-device communication takes place at regular intervals regardless of kernel execution duration. On slower systems, where the kernel execution might take over two seconds, the lack of such a system leads to kernel failure and prevents execution. While this could also be solved by decreasing the complexity of kernels and increasing their numbers, it would also affect the performance on faster systems.

Drawing from the findings of the above work a much larger test-bench was envisioned. One that would span hundreds of systems simultaneously, and if successfully implemented, would also expand the computational capabilities of the University of Huddersfield. A different application was desired for this test-bench, one that featured better optimizations for GPU computing, and could tackle the watchdog issues described above. This new test-bench, and the application selected for it, is described in the following chapter.

Chapter 5: OpenCL framework implementation over HTCondor

Based on promising results from initial testing, together with the existence of unused GPU resources within the compute infrastructure of the University, a project was proposed to exploit idle general purpose GPU resources for compute applications. A prime candidate for this was identified in the field of High-Throughput Computing.

In High Throughput Computing emphasis shifts from job execution rate to discrete job parallelism, meaning that the speed of individual compute resources is not as relevant as their number, availability and overall throughput. Prioritising throughput over frequency allows HTC to exploit opportunistic environments, where the number of available resources is constantly changing. One such environment is a University campus, where workstations can be used for computational purposes when otherwise idle, a process referred to as cycle-stealing. (Livny, Basney, Raman, & Tannenbaum, 1997)

A tool created specifically for this purpose is HTCondor, a workload management system for heterogeneous, opportunistic environments. In HTCondor tasks are distributed among available resources; where the term resources refers to workstation PC’s that have been idle for a set period of time. The system incorporates job execution queues, scheduling, prioritization, resource discovery, and of course, resource management. Another important feature, and one closely tied to the cycle-stealing mechanism, is a checkpoint system that prevents total work loss in the event of a workstation being removed from the resource pool, be it due to hardware failures or idle state being broken by used activity. For example, if a user returns to his workstation during job execution, the job is then migrated to a different workstation. With applications that support check-pointing, the data loss can be kept at a minimum.

Submitted jobs are matched to resources by using the ClassAd mechanism, a framework that allows both jobs and machines to specify requirements and or preferences in regards to resource allocation. The system actively scans for resource changes, and removes workstations that have been taken offline, or have not been available for long periods of time.

The University of Huddersfield implements an HTCondor pool, with an approximate 2300 workstations on campus, the resource pool totals at around 7000 CPU cores. However, due to the opportunistic nature of this system, peak availability is never achieved, with daily reports averaging between 700 and 3000 available nodes at any given time. The system is part of the Queens Gate Grid, the supercomputing resource created to support the research community at the University of Huddersfield. (Gubb, 2013)

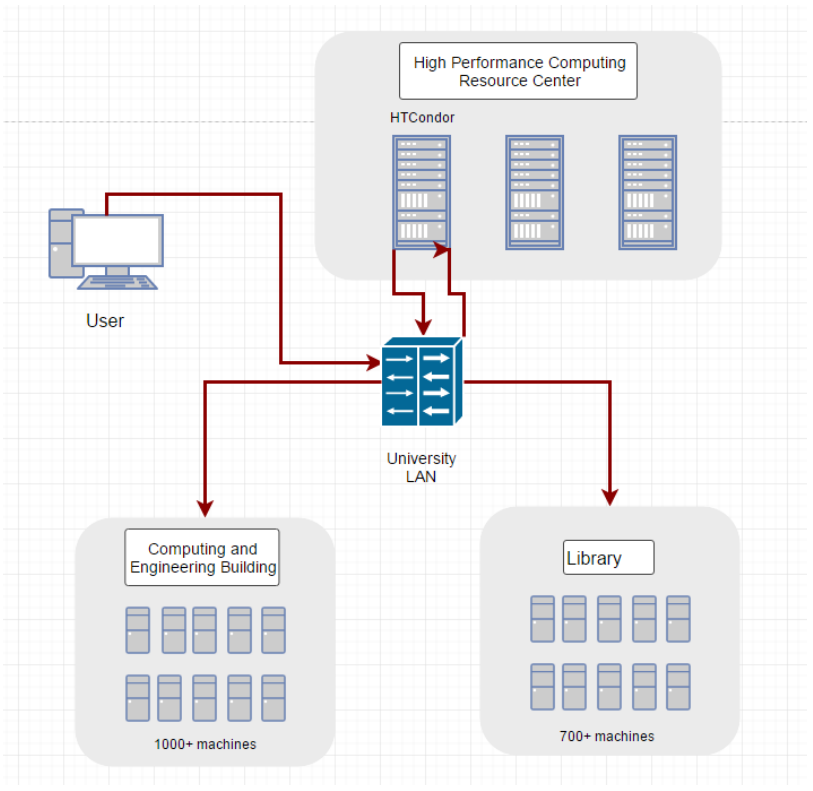

High Performance Computing Resource Center
HTCondor
User
University LAN
Computing and Engineering Building
1000+ machines
Library
700+ machines

Figure 27: Condensed Path of HTCondor Access

The goal of this research was the integration of the existing GPU resources within the HTCondor pool, supplementing the existing CPU based implementation without introducing any changes that would affect existing end-users. The resulting GPU resources would be used to supplement the dedicated GPU cluster. Another goal of this was a case-study of the effectiveness, flexibility and ease-of-use of the OpenCL framework across a highly heterogeneous resource pool. The chosen benchmarking application was based on Fast-Fourier Transforms.

While a significant number of Universities across the UK deploy HTCondor pools within their campuses, there is limited research output indicating GPU compute integration within these pools. (Gubb, 2013)

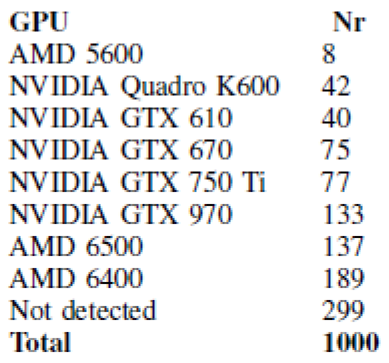
GPU detection within an HTCondor system is facilitated by two built-in detection software applications. These are:

1. CUDA based detection;
    - Software detects CUDA compliant GPUs, returns device name, memory limits and core count;
2. OpenCL based detection;
    - Software detects OpenCL compliant GPUs, returns device family and memory.

Relying on a CUDA based approach limits the use-case to only NVIDIA GPUs, and the built-in OpenCL algorithm does no return enough relevant information about the existing resources. For this reason, a different OpenCL program was designed, to poll a target computer for all available OpenCL devices (be it CPU or GPU) and record information. As OpenCL is compatible with CUDA devices, there is fragmentation when using it, thus trading platform specific optimizations for increased flexibility.

Device detection was executed over the live environment, where normally, the opportunistic environment works against benchmarking or individual node execution. This was overcome via the use of script based generation for ClassAds, targeting individual machines. 1000 units were randomly selected from the pool to partake in the benchmarking. GPU discovery is evidenced in Table 9.

Table 9: GPU Landscape



GPU	**Nr**
AMD 5600	8
NVIDIA Quadro K600	42
NVIDIA GTX 610	40
NVIDIA GTX 670	75
NVIDIA GTX 750 Ti	77
NVIDIA GTX 970	133
AMD 6500	137
AMD 6400	189
Not detected	299
Total	**1000**

As can be seen, although the system is very diverse, around 30% of polled machines failed to execute the GPU detection software. Following a brief investigation it was determined that a number of machines did not have video drivers installed for the dedicated GPU cards, thus preventing OpenCL execution. Vendor distribution is even, at around 50% each, not accounting the failed reads. Age-wise, the devices are fairly old in terms of GPU architectures, being released 2-3 generations ago. The performance gap between GPU generations is made evident by the performance graphs showing in this work.

5.1 Fast Fourier Transforms

It was decided that the previously designed application, for matrix multiplication, was not optimal for this new test-case, and a real world engineering application was needed to more accurately portrait the performance to be expected within such a system.

One such application, presented in investigated literature as frequently used in digital signal processing is the Fast-Fourier Transform, an algorithm for converting signals for representing time-domain signals in frequency domain. The FFT operation can be broken down into three major steps. (Brigham, 1974)

First, a multi-point time-domain signal is decomposed into multiple time-domains signals containing a single point. For example a 16 point signal is decomposed into 16 signals with one point each. The decomposition process, in this case, takes 4 stages to complete, each stage doubling the number of signals while halving the number of points per signal, thus resulting in 2, then 4, then 8 and finally 16 signals. Also, the decomposition process is interlaced, meaning that the signal is split into odd and even numbered samples, as observed in Figure 28. Note; this is generally done via bit reversal sorting, by flipping the binary value of the signal number. The number of stages needed to complete the operation is equal to the

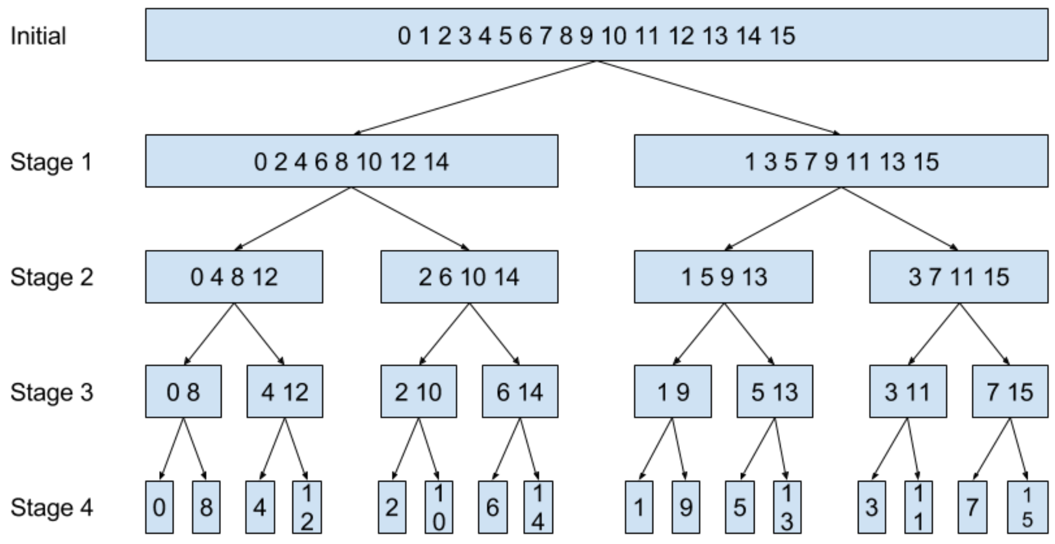


Figure 28: FFT Interlaced Decomposition

The second step lies in calculating the frequency spectrum for each resulting time-domain signal. This is the easiest task, as the frequency spectrum of a 1 point signal is equal to itself.

The final, and most complex step, involved combining the frequency spectra generated into a single spectrum. This is done in the reverse order of the original decomposition, which in the example case would be again 4 stages, yielding a 16 point frequency spectrum. The computational elements required to create these spectra are known as butterfly calculations.

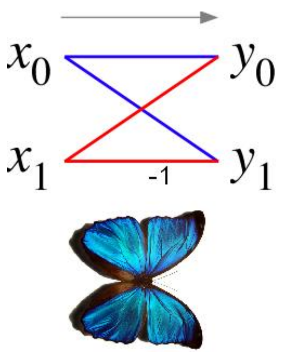


Figure 29: FFT Butterfly Calculation

An application for OpenCL based FFT computation optimized for GPU execution, primarily on the AMD architecture, that also support CPU execution for heterogeneous computing was discovered in the form of the clFFT library. This library offers a set of functions that can be used to create applications aimed at FFT execution on GPU devices. (AMD, 2016)

The clFFT library also offers a benchmark application, called clFFT client, which allows for the rapid assessment of device performance across multiple environments. It is a thoroughly designed application that includes device specific optimizations for both CPU and GPU architectures. As such it was chosen as the benchmark application, as opposed to designing a similar application that would serve the same purpose.

The application was compiled from source to both allow for the understanding of the library functionality and to allow further development of FFT applications based on the clFFT library.

5.2 Benchmark Execution

GPU benchmarking was accomplished over the same live environment as the previous set, namely over the 700 machines detected as having GPU components during. The chosen application for GPU benchmarking was Fast-Fourier Transform, a computation that is both widely used in the field of engineering and makes use of the massively parallel GPU architecture. Each of the 700 units executed single-dimensional FFTs over 17 sizes, with 1000 iterations per size, to ensure benchmark precision. This resulted in approximately 12 million FFTs, and roughly 28,000 CPU hours. It would take a single computer, fitted with a regular 4 core CPU, more than 3 years of constant work to accomplish this task. (Dafinoiu, Higgins, & Holmes, Accelerating High-Throughput Computing through OpenCL, 2016)

When operating in a heterogeneous environment as HTCondor, knowing what devices are available prior to execution is challenging, thus an application aimed at such a system should incorporate means of dynamically optimizing resource allocation during run-time. This however is being the scope of this work, and as such, the lowest common denominator was used when optimizing the application performance.

CPU performance was established for comparison purposes, on a standard Intel i5 CPU, as seen in Figure 30. The metric used to measure the performance of the system was the Giga

FLOP. The formula used to derive the GFLOPs is shown in the HTCondor Related Scripts section.

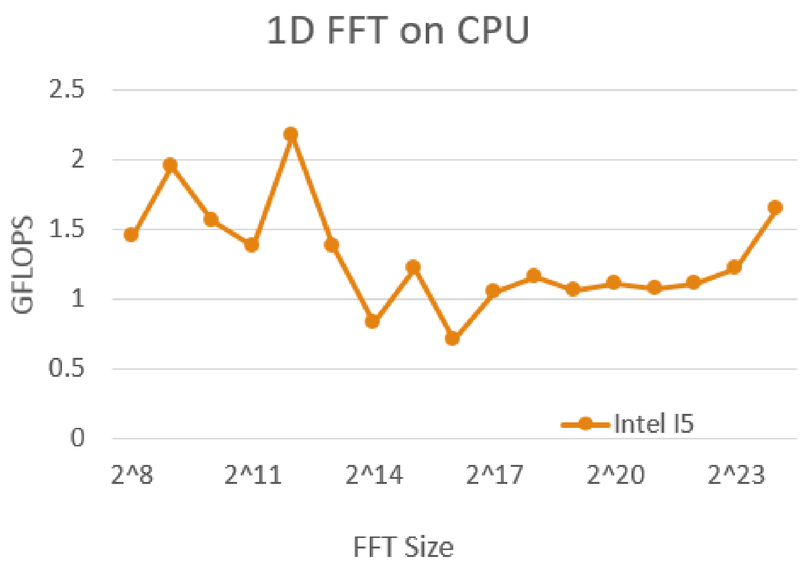


Figure 30: CPU Benchmark

Executed in a controlled environment, the benchmark takes between 60 and 75 minutes on an average system to complete. On the live environment however, execution duration was greatly affected by resource usage during the day. This resulted in full benchmarking taking around 48 hours, with most of the systems being able to complete their work at night. This is due to having fixed allocations for targeted computers, which is only the case during benchmarking.

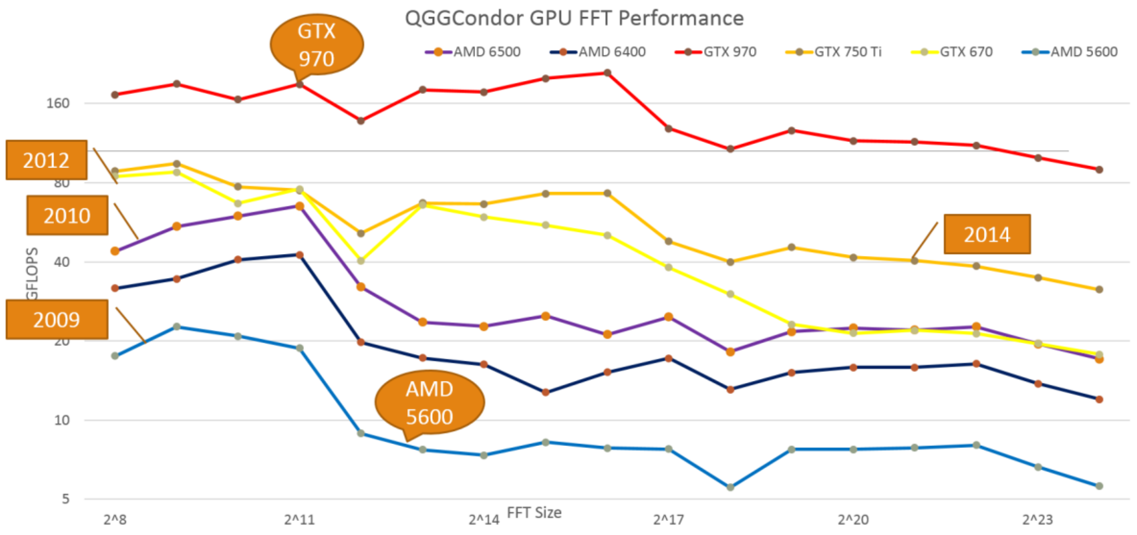


Figure 31: GPU FFT Performance

Detailed performance is showcased in the above chart, with a more detailed breakdown given below.

The chart shows extremely varied performance across the system, with results ranging between 7 and 180 GFPLOPS of performance. One can observe that with each generation of GPUs, as evidenced by the release year referenced in Figure 31, performance is increased by a significant amount. This becomes more evident as one inspects the clock-speeds and compute units available in each GPU generation, factors which influence the overall performance of parallel based computations.

Performance-wise, the newest GPU card, the NVIDIA GTX 970, is the best performing GPU, while also being the 3rd most used GPU in the pool. Due to university policies, computer systems are upgraded every few years, leading to a reduction in underperforming GPUs within the HTCondor pool. For example, the worst performing GPU in the system, the AMD 5600, is on average 15 times slower than the maximum, but it also only exists in 8 of the 700 computers, as it is slowly replaced by newer hardware.

However, the University employs a dedicated GPU cluster for massively parallel applications like the FFT, and as such, any new contender needs to be compared against it, in order to determine its effectiveness. This cluster is comprised of two NVIDIA C2050 computing processors, these GPUs are purpose-built with parallel computation in mind, incorporating error correction codes, high memory, and asynchronous, high-speed, memory transfer. Released in 2010, these GPUs advertise 448 cores operating at 1.15 GHz. In perspective, the GTX 970 cards advertise 1664 cores operating at 1.05 GHz. (Dafinoiu, Higgins, & Holmes, Accelerating High-Throughput Computing through OpenCL, 2016)

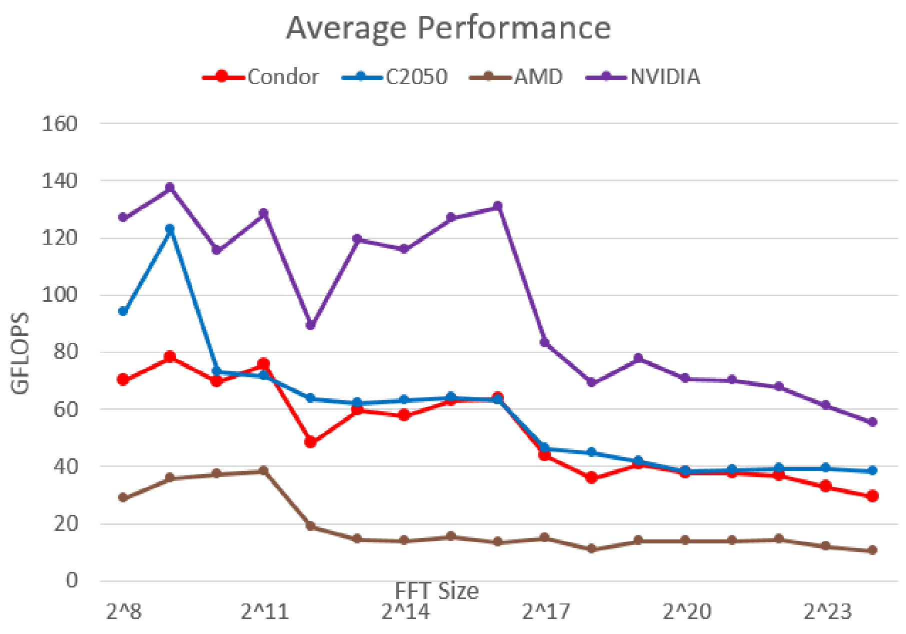


Figure 32: GPU Cluster Comparison

As can be seen in Figure 32, HTCondor average performance closely matches that of a single dedicated GPU card. This however is on a per-node basis, meaning that there are hundreds of more nodes available on HTCondor that on the dedicated cluster, capable of handling GPU computation.

It can also be noticed that the average performance of NVIDIA GPUs, or better said, cards newer than 2010, greatly outperforms the C2050. A noteworthy mention is that the GTX 970 has twice the compute units of the C2050 while also operating at almost twice the clock speed.

This benchmark revealed an untapped resource of approximately 30 Teraflops computational power on just the 700 nodes, extrapolating the results to the 2200+ nodes on campus, the peak performance of the HTCondor system reaches 90 Teraflops.

OpenCL integration within the HTCondor resource pool has revealed a number of GPU resources that can be exploited in order to increase system performance for parallelizable applications. OpenCL has proven to be a highly versatile framework easily adaptable to a highly heterogeneous environment.

This work has shown that newer generation general purpose GPUs are able to match the performance of older dedicated GPU resources, offering a much better price/performance ratio. However, maximizing performance over a heterogeneous system, such as HTCondor is extremely difficult, and requires changes to both the application and the system itself. The OpenCL framework, through its flexibility and ease-of-use, is a valid candidate for developing heterogeneous applications over such systems. A conference paper based on this work was published with the Emerging Technologies Conference in June 2016, where it received positive feedback from reviewers and peers. (Dafinoiu, Higgins, & Holmes, Accelerating High-Throughput Computing through OpenCL, 2016)

5.3 HTCondor Related Scripts

The HTCondor heterogeneous resource pool was not designed to allow for benchmarking of individual units inside the pool. As such, picking a set number of units and executing a given application over each individual unit is not supported by the system, and accomplishing this task required the use of automation scripts. Also, the computation of 17 FFT executions over 700 computers results in a large amount of generated files that need interpreting. The 1000 iterations are managed inside the application, and as such do not count towards the number of generated files.

Also, the formula used to derive the GLFOP performance of each individual system, used by the clFFT client is:

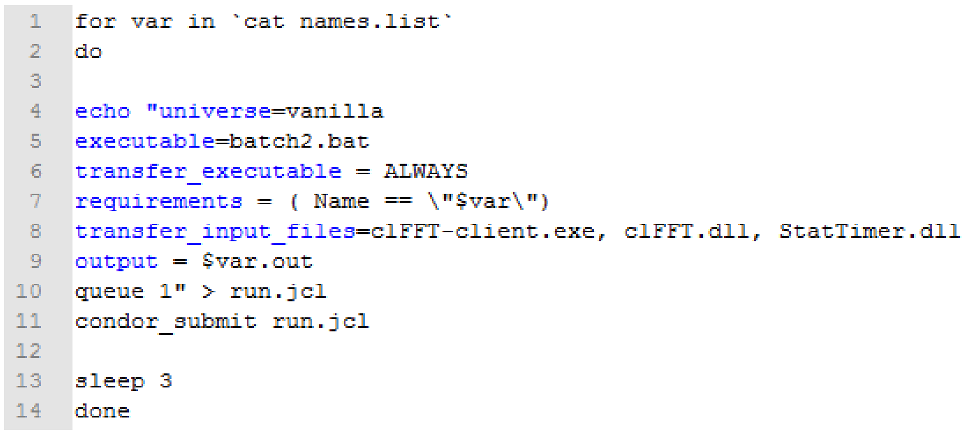
$$GLOPS = (batch\_size * (5 * dimension\_size * (log\,(dimension\_size)\ / log\,(2)))) / (1000000 * walltime).$$

Where 5 is a constant for real FFTs calculations, and walltime is the duration of the execution. (AMD, 2016)

5.3.1 Condor Individual Unit Execution

In order to ensure that applications execute on each of the 700 targeted units only once, no matter how many are available or how many times the job is restarted, the only solution identified was by demanding from HTCondor a specific machine for each job.

As such a script was written, this scripted created a job-file to be submitted to HTCondor based on each line within a predefined file. This file contained the hostname of each of the 700 computers on a separate line. This script was written for the Linux shell environment used on HTCondor, Figure 33.



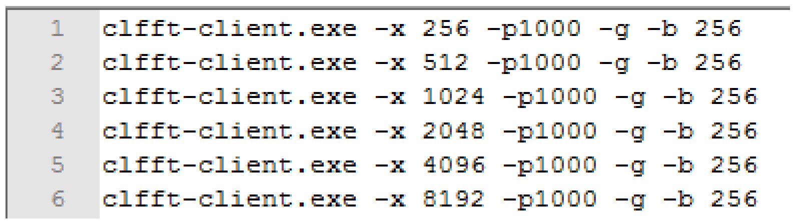
```
for var in `cat names.list`
do

echo "universe=vanilla
executable=batch2.bat
transfer_executable = ALWAYS
requirements = ( Name == \"$var\")
transfer_input_files=clFFT-client.exe, clFFT.dll, StatTimer.dll
output = $var.out
queue 1" > run.jcl
condor_submit run.jcl

sleep 3
done
```

Figure 33: Shell Script for Execution

The above script contains all the standard job parameters of an HTCondor job (minus architecture requirements, for simplicity) and as can be seen creates and queues a job iteration for each machine, asking that the output file resulting from the said job be named after the executing machine. This generated output file contained the 17 executed FFT outputs.

5.3.2 Individual Machine Benchmarking

It can be noted in Figure 33 that the executable passed to HTCondor is not the clFFT-client application, but rather a batch script written for windows. This script executes 17 different instances of the application each with different parameters. The arguments passed were related to the size of the computed FFT, the number of iterations run, the chosen device architecture, and the so-called batch size, which reflected the size of the used arrays, similar to the workgroup size used in the Matrix Multiplication benchmark. It is worth mentioning that maximum batch size was determined by a mix of used FFT size and device memory, leading in lower batch sizes being used for larger FFTs. Note; the batch script also executed the application with an argument that returned the name of the targeted device, in order to determine what type of GPU executed the application.



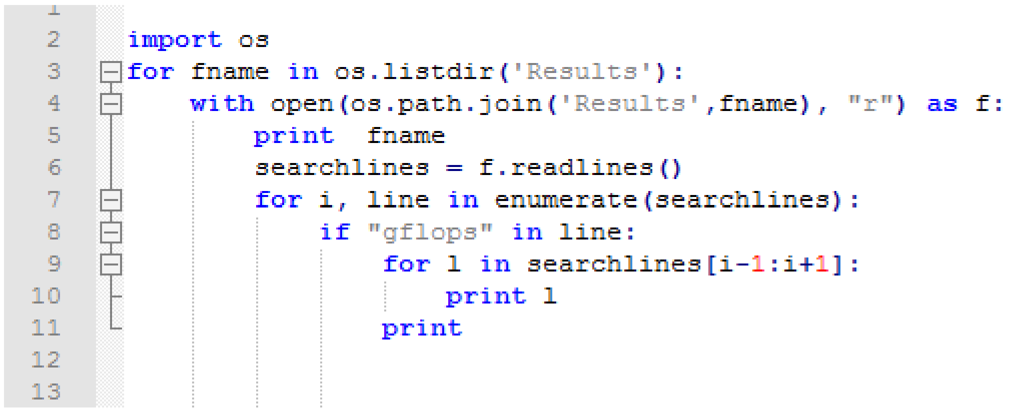
```
clfft-client.exe -x 256 -p1000 -g -b 256
clfft-client.exe -x 512 -p1000 -g -b 256
clfft-client.exe -x 1024 -p1000 -g -b 256
clfft-client.exe -x 2048 -p1000 -g -b 256
clfft-client.exe -x 4096 -p1000 -g -b 256
clfft-client.exe -x 8192 -p1000 -g -b 256
```

Figure 34: Fragment of Windows Batch File

5.3.3 Data sorting and processing

The resulting data was processed and sorted on the development machine, using a batch script and a python script. The batch script moved all files relevant to an FFT size into an intermediate folder, and then executed the python script for that folder, logging the output into a separate file, and then moving the files from the intermediate folder to a separate location for storage. This was iterated for each FFT size, and resulted in 17 files containing only the relevant timing information needed for benchmarking, Figure 35. The remaining files the ones containing the device name. They were moved to a separate file.

Performance sorted by GPU device used was retrieved in Linux, using the GPU names to determine which files to target, then the “grep” command to retrieve the execution duration and GFLOPS.



```
import os
for fname in os.listdir('Results'):
    with open(os.path.join('Results',fname), "r") as f:
        print  fname
        searchlines = f.readlines()
        for i, line in enumerate(searchlines):
            if "gflops" in line:
                for l in searchlines[i-1:i+1]:
                    print l
                print
```

Figure 35: Python Script for Data Sorting

Chapter 6: OpenCL FPGA Acceleration

The FPGA, introduced in a previous chapter, is a type of integrated circuit designed specifically to be altered after manufacturing. The FPGA is based on configurable logic blocks connected via programmable interconnects, these features allow for the “field-programmable” element of the FPGA, altering the design after manufacturing.

Over the past years FPGAs have found their way into many different markets, from image processing to networking, from audio applications to aerospace applications, the versatility of the FPGA and the much lower development costs associated with its usage have allowed it to flourish.

FPGAs come in many different shapes and sizes, depending on their intended purpose. Those relevant to networking industries come in rack-mounted chasings with network connectivity and increased RAM. Those dedicated to image processing may come in a PCI-E format, and could lack network connectivity entirely. And while all these FPGAs are different, in terms of I/O connections, memory, and size, they are all programmed in the same way. This is because the low-level allocations on the actual chip are handled by the compiler. The user is able to make low-level allocations to further tune and optimize the device, however it is not mandatory. From a programming perspective, the FPGA is like a blackboard; the user draws upon it the schematic of the desired design, making sure not to exceed the size of the board. Alterations to the design are possible, and as easy as wiping chalk off the board. However, unlike the blackboard, which can only be used to prototype a design, the FPGA allows for the full implementation of the system, and further functional testing and redesign. (Dafinoiu, FPGA Based Implementation of Offset PPM, 2015)

The FPGA takes a parallel approach to instruction execution, limited by the amount of logic blocks and interconnects available. Whereas a CPU is only able to execute one instruction at a time, or a small number of them on a multi-core CPU, the FPGA is able to spawn as many processes as it can fit within its size for the desired task.

One of the main drawbacks of using FPGAs, especially for computational purposes, has been the steep learning curve of the HDL environment, as well as the need for a parallel approach to computing. This is why this chapter aims to investigate the usability of the OpenCL framework for FPGA design.

6.1 Hardware Description Language

VHSIC HDL, or VHDL for short, is a hardware description language used to describe digital or mixed-signal systems in electronic design. Unlike more familiar programming languages which run instructions sequentially, VHDL runs operations in parallel, making it a dataflow language. Parallel execution is done via the use of processes that are able to run independently of one-another, being executed when a predefined criteria is met. VHDL allows for the text description of a logic circuit which is then synthetized, simulated and placed onto a chip to create a working design. Being an IEEE industry standard for FPGA programming, VHDL is easily ported between different FPGA devices, given viable hardware configurations. (Pellerin, 1997)

One of the main disadvantages of using VHDL is the design complexity of projects, since VHDL is not as high-level a language as C/C++. There are also many different software development kits offered by FPGA manufactures, each utilizing a completely different set of tools, and features, that further encumber the design process. Also, because the FPGA targets a more niche market than conventional programming languages, there is a lack of available instructions on the use and optimization of FPGA designs, with many designs being offered as ready to use Intellectual Properties.

6.2 Altera SDK for OpenCL

Because of the way an FPGA is designed, the approach to using OpenCL to program it is not as straightforward as the ones for GPU or CPU. This is because prior to being designed with a task in mind the FPGA is a blank slate, as such unable to be exploited for computational needs. In order to program, or flash, the FPGA to be used as an OpenCL accelerator, a compiler is needed to turn OpenCL kernel code into the binary coded used to flash the FPGA. (Altera, 2011)

Altera, being one of the leading FPGA developers, as well as one of the founding members of the Khronos group, developing OpenCL, offers a SDK for OpenCL, known as AOCL, to allow for the creation of FPGA based OpenCL applications. In essence, the SDK is easy to use; it simply takes an OpenCL kernel code and converts it into a file that can be used to flash the FPGA so that it can be used as an accelerator.

The OpenCL SDK uses its own OpenCL calls in the host and kernel code, since the FPGA operates differently from previously discussed systems. Also, FPGA design greatly benefits from properly optimized algorithms due to its much higher compute unit count. Altera

OpenCL designs are based on the Altera specific ”utils.h” library rather than the “cl.h” library. The utils library is not a stand-alone implementation of OpenCL, but rather a set of functions specifically created for FPGA use that sit on top of the OpenCL library. (Altera, 2011)

The Altera OpenCL SDK, unlike CPU or GPU OpenCL SDKs is not free, following the trend in FPGA development environments, which are not inherently available to the general public, making FPGA OpenCL development more of a niche market than the FPGA one. The SDK is however offered for free as part of the Altera University programme, and was acquired towards the end of the project. Hence, data gathered for the FPGA development and benchmarking sections is limited; a more detailed and in-depth research will be conducted as part of future work.

6.2.1 Altera Offline Compiler

The AOC is used to compile .cl kernel codes into hardware configuration file, containing the FPGA image to be used in a binary format. This is used by the host, at runtime, to execute the kernel applications. The AOC generates the files needed to program the FPGA and execute the kernel application during runtime, Figure 36. (Altera, 2016)

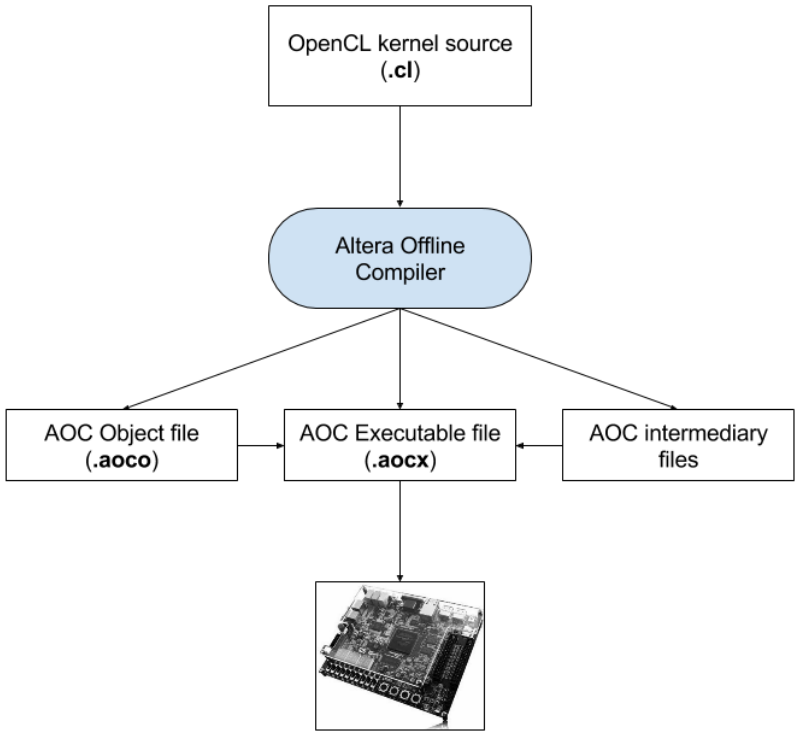


Figure 36: AOC Flowchart

The AOC employs two types of compilation; a one-step compilation for simple kernels that feature minimal compiler optimizations. It is a simple procedure involving a single command that generates an .aoco file containing intermediate information and used in generating the next file, the .aocx. This second file contains the binary for the hardware configuration of the FPGA and is used by the host during runtime to create and execute the kernels on the device. A subfolder is generated along with the two files contains a number of intermediary files used to create the final hardware binary. A log file is also created containing the estimated resource usage within the FPGA.

The multi-step compilation is used in more complex kernels, which can greatly benefit from optimizations. The first step assumes the form of an intermediate compilation, which checks for syntax errors, then creates the .aoco file without generating the hardware binary for it. This step also generates the log file showing estimated resource usage.

Second, the functionality of the OpenCL kernel can be emulated on one or multiple emulation devices to locate any existing functional errors. Third, the resource usage of the OpenCL kernel on the FPGA can be reviewed to uncover possible optimizations to hardware resource usage.

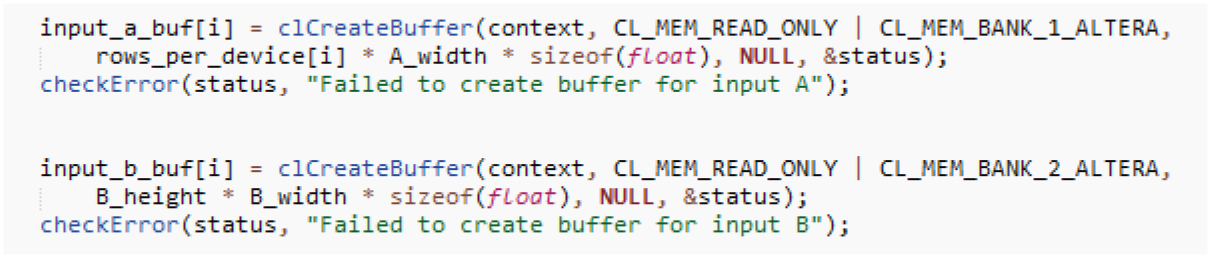
Profiling, allows the introduction of performance counters into the .aocx file. These functions measure performance during runtime and can be interpreted using the Altera Profiler to further optimize the application.

Once all desired steps are achieved, the final application can be compiled from the .aoco file to generate the desire .aocx.

6.2.2 Application porting

In order to function with the AOCL, any OpenCL design targeting Altera FPGAs benefits from using the libraries released by Altera for this purpose reason; which add number of features to the base OpenCL library.

For example, the opencl.cpp file shipped by Altera together with an OpenCL example design contains both functions that are relevant to application design, such as error checking, profiling, or wait-timers, but also a number of Altera specific functions for memory allocations, .aocx interpretation, and memory clean-up. In Figure 37, memory buffers are created for two input matrixes, these are allocated into the FPGA memory on two separate banks, which increases memory bandwidth during data transfers.



```
input_a_buf[i] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_BANK_1_ALTERA,
    rows_per_device[i] * A_width * sizeof(float), NULL, &status);
checkError(status, "Failed to create buffer for input A");


input_b_buf[i] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_BANK_2_ALTERA,
    B_height * B_width * sizeof(float), NULL, &status);
checkError(status, "Failed to create buffer for input B");
```

Figure 37: Custom Memory Region Targeting

These libraries ensure faster development of FPGA based OpenCL applications at the cost of encumbering cross-platform design. It was shown in a previous chapter that with minimal changes an OpenCL application could be changed between targeting CPUs and GPUs; however the FPGA implementation requires a much more complex redesign. Despite this, the FPGA segment could be implemented alongside the former two, in order to create a unified, heterogeneous application.

6.3 DE1 System-on-Chip

The FPGA platform chosen for the development and benchmarking of OpenCL based applications was a development kit aimed at university use known as the DE1 System-on-Chip. This small development board features an ARM CPU paired with an ALTERA FPGA designed for embedded applications. (Terasic, 2016)

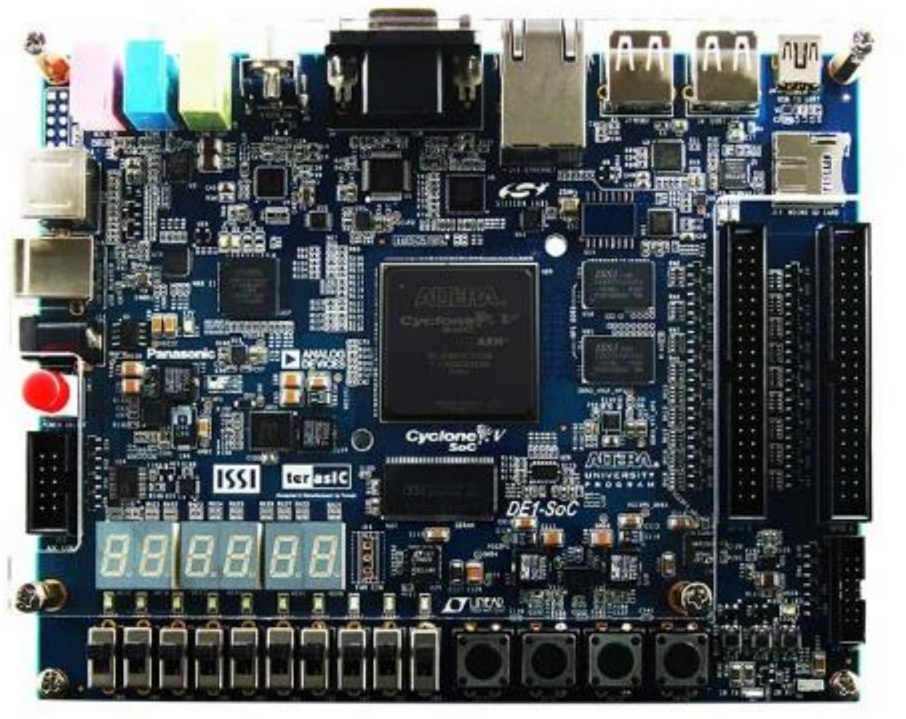


Figure 38: DE1 SoC Development Board

The device includes multiple features intended for user input during run-time, embedded system operation as a standalone computer, and expansion slots, however these features are beyond the scope of this work and as such are not explored further.

6.3.1 Setup

Before software development began, the SoC was set-up for OpenCL execution, connected to a PC for command-passing, and tested for functionality.

Given that the FPGA is controlled by the embedded ARM CPU, an operating system must is required to control all actions on the board. Prior to initial start-up of the device, the FPGA configuration mode must be defined using the on-board via the MSEL pins.

There are three modes in which the FPGA can be configured; first, the FPGA is programmed using the on-board flash memory. This is used when programming the FPGA from a host computer using the Altera Programmer. Second, the FPGA is programmed using the built-in processor, referred to as a Hard Processor System, or HPS running a Linux OS with a command line interface, Figure 39. The third and final configuration mode is similar to the second, in the sense that the FPGA is once again programmed via the HPS; however this mode is set when using much larger Linux OS images that feature desktop environments as opposed to the CLI. (Terasic, 2016)

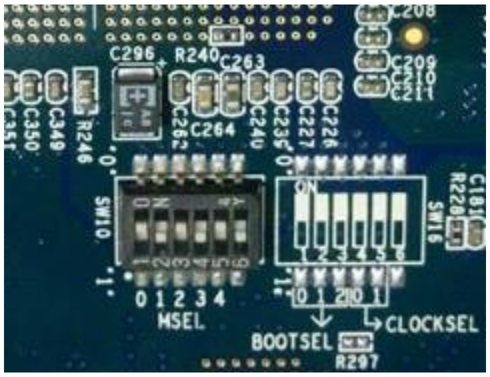
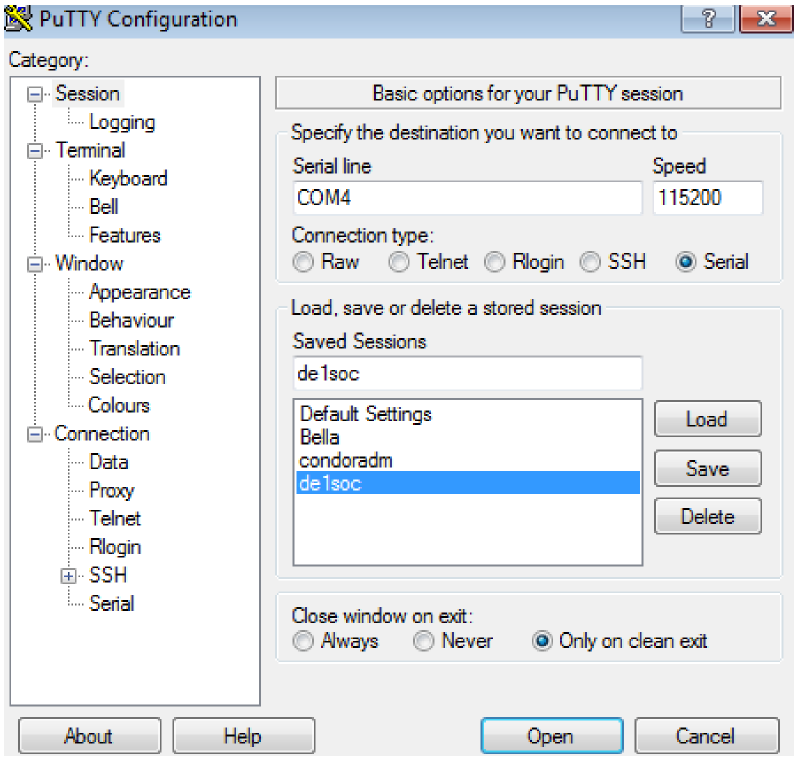


Figure 39: MSEL Position for Linux with CLI

For this experiment, the second configuration mode was chosen, using the DE1-SoC Linux Console image that was burned onto an external flash memory card.

Connection to the board was established using a USB port through a serial connection managed by the PuTTY software, Figure 40. Once powered on, the system boots the Linux OS and can be controlled via the serial connection.



Specify the destination you want to connect to
Serial line: COM4
Speed: 115200
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Load, save or delete a stored session
Saved Sessions: de1soc
Default Settings
Bella
condoradm
de1soc
Close window on exit: Always Never [✓ Only on clean exit]

Figure 40: Serial Connection Through PuTTY

The Linux image of the DE1-SoC comes with two demo applications preinstalled. While not in any way compute intensive, these two applications serve as a straightforward method to test the functionality of the OpenCL environment.

Prior to execution, the user must initiate the OpenCL environment, loading the OpenCL driver and the environment variables pointing to the OpenCL run-time library on the system. This is done via a pre-installed script called “init_opencl.sh”, Figure 41.

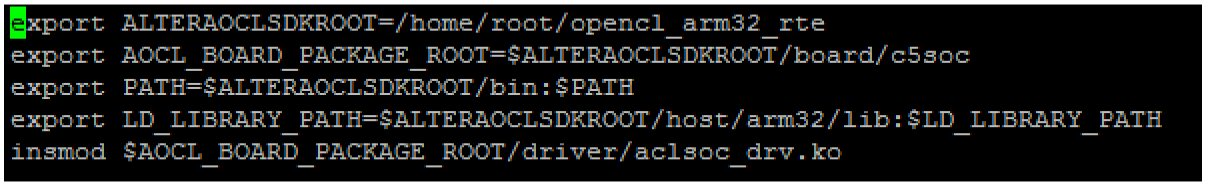
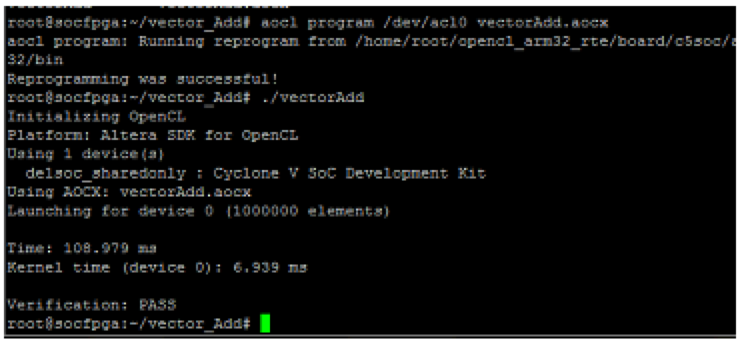

```
export ALTERAOCLSDKROOT=/home/root/opencl_arm32_rte
export AOCL_BOARD_PACKAGE_ROOT=$ALTERAOCLSDKROOT/board/c5soc
export PATH=$ALTERAOCLSDKROOT/bin:$PATH
export LD_LIBRARY_PATH=$ALTERAOCLSDKROOT/host/arm32/lib:$LD_LIBRARY_PATH
insmod $AOCL_BOARD_PACKAGE_ROOT/driver/aclsoc_drv.ko
```

Figure 41: Contents of init_opencl.sh Script

Before executing the OpenCL application of choice, the FPGA must be programmed with the binary file generated by the AOC for said application. The AOCL is invoked for this purpose using the command "aocl program /dev/acl0 application.aocx" where acl0 is the targeted FPGA and the .aocx file is the used binary. Following this step, the targeted host executable can be run for OpenCL execution, Figure 42. (Terasic, 2016)



```
root@socfpga:~/vector_Add# aocl program /dev/acl0 vectorAdd.aocx
aocl program: Running reprogram from /home/root/opencl_arm32_rte/board/c5soc/
32/bin
Reprogramming was successful!
root@socfpga:~/vector_Add# ./vectorAdd
Initializing OpenCL
Platform: Altera SDK for OpenCL
Using 1 device(s)
  de1soc_sharedonly : Cyclone V SoC Development Kit
Using AOCX: vectorAdd.aocx
Launching for device 0 (1000000 elements)

Time: 108.979 ms
Kernel time (device 0): 6.939 ms

Verification: PASS
root@socfpga:~/vector_Add#
```

Figure 42: FPGA Programming and Vector Addition Demo

Having determined that the device functions as expected, the next step is installing the relevant development hardware on the PC.

The AOCL is bundled together with the Quartus software, in this case version 14.1, which is recommended for the DE1-SoC. Installation of the software is straightforward and will not be detailed in this work. Once the software has been installed, the user has to set the environmental variables for the AOCL. These variables point to the AOCL installation and the board support package for the targeted FPGA.

Applications are designed in the same manner as CPU and GPU ones, using, in this case, the Visual Studio environment, however compilation of the two required files, host and kernel, is significantly different.

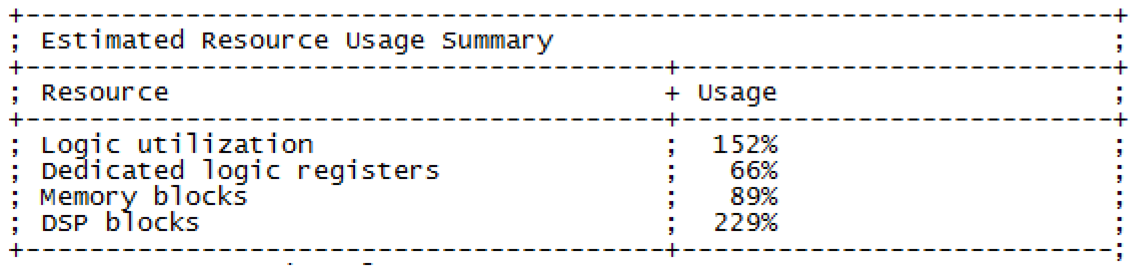
The AOC is used to turn the kernel code into a hardware configuration file, through the command-line interface. The practice is straightforward however the compilation process is resource demanding and time consuming. The compilation process will be detailed in the following section.

The host-code needs to be compiled for the ARM processor, as a Linux executable. Fortunately, the Altera software comes with an embedded Linux environment through which a Linux executable can be cross-compiled on Windows by using the MAKE software.

6.4 Benchmark Application

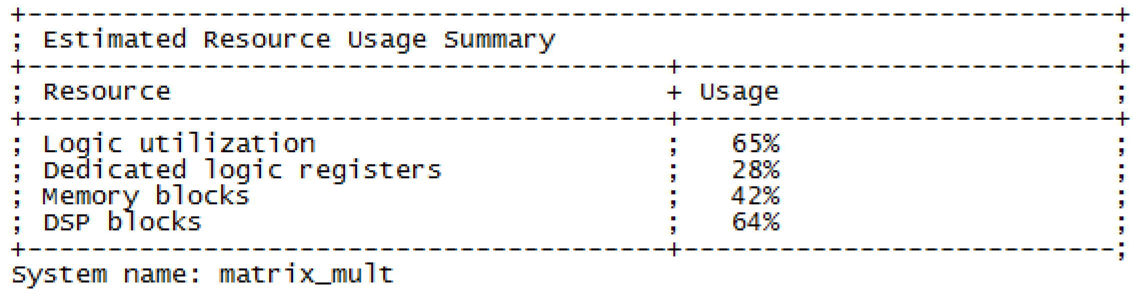
The chosen benchmarking application for the SoC board was once again based on Matrix Multiplication. However, rather than modifying the existing CPU/GPU application, this implementation was based on a similar design offered by Altera used in benchmarking much larger FPGA devices. (Altera, 2015)

Because of this, the design had to be reduced in size and complexity in order to fit onto the FPGA. This design change was similar to the batch method used in the FFT implementation; however, since it affects the kernel code as well as the host, it cannot be altered at run-time as it is used in generating the hardware configuration binary. As such, two variations of the application were created, one based on block sizes of 16 and 8. The initial version of the application, with a block size of 64, reported an estimated resource usage that exceeded device capabilities, seen in Figure 43.



```
+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ;  152%                     ;
; Dedicated logic registers              ;   66%                     ;
; Memory blocks                          ;   89%                     ;
; DSP blocks                             ;  229%                     ;
+----------------------------------------+---------------------------;
```

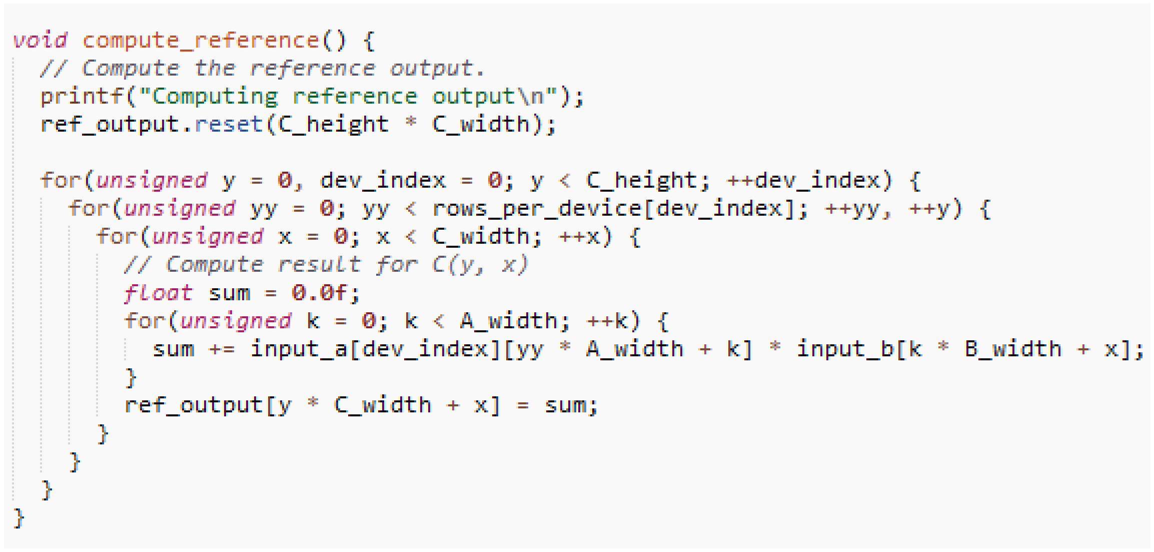
Figure 43: Usage Estimation Report for block 64



```
+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ;   65%                     ;
; Dedicated logic registers              ;   28%                     ;
; Memory blocks                          ;   42%                     ;
; DSP blocks                             ;   64%                     ;
+----------------------------------------+---------------------------;
System name: matrix_mult
```

Figure 44: Usage Estimation Report for block 16

The application included accuracy testing, where the application was executed on the CPU after FPGA execution and results were compared, Figure 45. This feature was used in initial execution of the application, on small matrix numbers however it was removed from the benchmark version because the embedded CPU was struggling to complete the larger matrix size executions.



```
void compute_reference() {
  // Compute the reference output.
  printf("Computing reference output\n");
  ref_output.reset(C_height * C_width);

  for(unsigned y = 0, dev_index = 0; y < C_height; ++dev_index) {
    for(unsigned yy = 0; yy < rows_per_device[dev_index]; ++yy, ++y) {
      for(unsigned x = 0; x < C_width; ++x) {
        // Compute result for C(y, x)
        float sum = 0.0f;
        for(unsigned k = 0; k < A_width; ++k) {
          sum += input_a[dev_index][yy * A_width + k] * input_b[k * B_width + x];
        }
        ref_output[y * C_width + x] = sum;
      }
    }
  }
}
```

Figure 45: Reference Computation Executed on the ARM CPU

Application complexity was further reduced by removing the built-in options feature and replacing it with a single external parameter which determined matrix size, in tone with the previous Matrix Multiplication application.

The application was initially created to target multiple FPGA devices at the same time, through the introduction of "for" loops in the host code segment, this functionality was maintained, to be used in future work.

6.5 Execution

The fully compiled application, composed of two files, the Linux executable and hardware configuration binary were transferred onto the SoC board via flash storage. Programming the FPGA with the Matrix Multiplication hardware binary was done first, as it would remain unchanged until the device was powered down, or a different binary was programmed in its place. The Linux executable was ran through a script calling it with different external parameters for matrix sizes multiple time, with the resulting output being written to a separate file for interpreting. Execution results are shown in Table 10.

Table 10: FPGA Execution Time

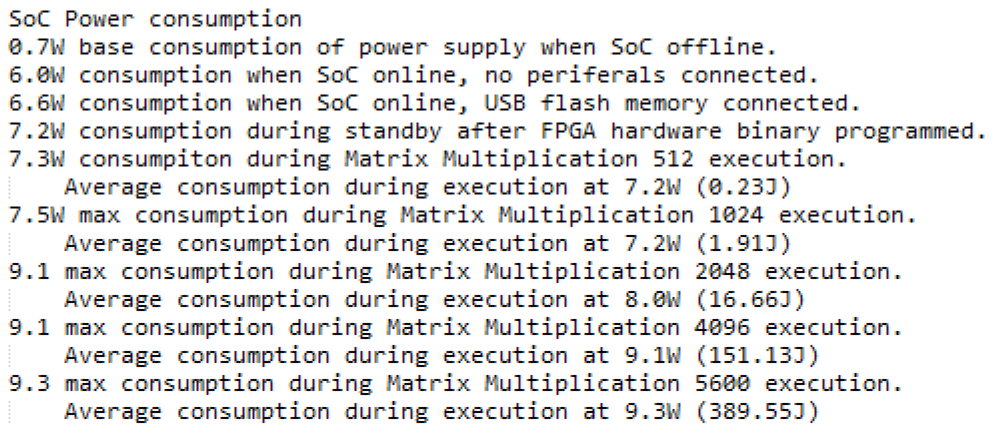
Execution Time ms	Workgroup Partition		
Matrix Size	Block Size 16 Workgroup 8	Block Size 8 Workgroup 4	Block Size 8 Workgroup 2
512	32.38	30.73	58.51
1024	265.91	245.671	466.84
2048	2,083.45	3,830.81	4,041.5
4096	16,608.05	57,556.30	45,898.44
5600	41,888.51	49,716.25	76,312.36

It can be observed that overall, higher block sizes lead to increased performance. However for very small values the much smaller block size proves to be faster due to less overhead communication. For the above implementations, block size 16 exploited the FPGA resources to the fullest of their potential. The other two implementations used around 50%, and 30% of available logic. The performance doesn't however seem tied to the amount of resources used, leading to the assumption that there is still optimization work to be done to the compiler.

The execution time noted in red appears to represent a bug that only affects the 4096 value on block size 8, workgroup size 4. For unknown reasons, performance on that matrix size is heavily impacted. The values immediately above and below it (4088/4104) both execute in ½ of the time shown above. Debugging of the implementation in order to determine the actual cause of the impacted performance was not executed due to time constraints.

Unfortunately there is not enough memory on the FPGA device to execute matrix sizes of 8192x8192. The execution duration of the FPGA execution is up to 20 times faster than the CPU execution, however it does not match GPU performance. However, this device was not intended for computational performance, but rather functional testing and experimentation. It is also worth mentioning that as a SoC solution, the power consumption is greatly reduced, most likely leading to equivalent or better performance than the two other architectures when compared in terms of computational power per watt.

A preliminary measurement of the SoC power consumption was executed during the block 16 executions, in order to provide a stepping-stone for further work in this area. These measurements are showcased below.



```
SoC Power consumption
0.7W base consumption of power supply when SoC offline.
6.0W consumption when SoC online, no periferals connected.
6.6W consumption when SoC online, USB flash memory connected.
7.2W consumption during standby after FPGA hardware binary programmed.
7.3W consumpiton during Matrix Multiplication 512 execution.
    Average consumption during execution at 7.2W (0.23J)
7.5W max consumption during Matrix Multiplication 1024 execution.
    Average consumption during execution at 7.2W (1.91J)
9.1 max consumption during Matrix Multiplication 2048 execution.
    Average consumption during execution at 8.0W (16.66J)
9.1 max consumption during Matrix Multiplication 4096 execution.
    Average consumption during execution at 9.1W (151.13J)
9.3 max consumption during Matrix Multiplication 5600 execution.
    Average consumption during execution at 9.3W (389.55J)
```

Figure 46: SoC Power Consumption

6.6 Conclusion

It is shown in this chapter that FPGA based development of OpenCL applications is not only possible, but also a viable solution for achieving computational speed-up for applications that can benefit from heavy parallelisation. The application development when using the OpenCL framework was shown to be both straightforward, and highly similar to applications targeting other architectures in OpenCL. Based on the author’s previous experience with HDLs, using the OpenCL framework for developing FPGA applications is much less time-consuming, while also enabling for rapid and straightforward improvements to the developed application at any point after development. The OpenCL framework also opens up the FPGA architecture to a much larger pool of developers since it does not involve a tedious learning process for HDLs.

Chapter 7: Further Research

A continuation of this work could seek to include the newest introduced architecture in high-performance computing, that of the many-core systems, like the Intel Xeon-Phi. Systems such as this one, that include a high number of compute units could greatly benefit from the OpenCL framework.

During this work, a number of applications were created for system benchmark across multiple architectures, and although these applications were based on similar algorithms, they existed as separate entities. While this allowed for a better understanding of how each architecture functions, creating a single application, containing functionality for each entity type, either individually or together, should be the next goal within this topic. Work toward this goal could also determine the best approach to heterogeneous programming, creating a single kernel code to be compiled at run-time or multiple binary files, one for each architecture type or sub-type.

All applications used in this work were based on the OpenCL 1.2 version; however the latest version is 2.2. It is expected that utilizing these newer standards would increase performance while decreasing complexity and development time. However, due to the slow adoption of these standards by manufacturers, OpenCL is fragmented across devices, similar to the fragmentation of the Android OS platform. As such developing applications based on standards that are not yet supported by all targeted architectures seems counterintuitive. Future research could determine whether the benefits of using newer OpenCL standards outweigh the cons.

The University of Huddersfield, like many other institutions like it, makes use of a number of Digital Signal Processor devices as teaching tools. Since DSPs are advertised as specialized accelerators that OpenCL can target, attempts could be made at determining the computational benefits of integrating them as part of more robust heterogeneous systems.

The power measurements presented within this work are very limited in both scale and complexity; however more in-depth research is needed to determine the actual speed-up of using FPGA devices as computational accelerators alongside CPUs and GPUs. To this end, a much more comprehensive test environment needs to be used, one that can accurately measure the power usage of both the accelerator device and the complete system required to operate it.

This work has shown that FPGA devices are able to match and surpass CPU devices, while requiring a fraction of the power and storage space. Based on the findings of this work, and the initial power measurements, a future study was planned, to determine the feasibility of implementing FPGA devices as part of the High-Performance Computing environment. This could be either as extensions to current hardware, by integrating PCI-E devices, or as separate stand-alone resources in the form of a SoC FPGA resource pool.

Chapter 8: Summary and Conclusions

This work focused on investigating the efficiency of the OpenCL development framework and environment in conjunction with heterogeneous systems, mainly, exploiting the massively parallel architectures of GPUs, FPGAs or High-Throughput Systems. The work covered the development of OpenCL applications aimed at benchmarking the performance of accelerators within heterogeneous systems, portability across different devices and platforms, and design methodologies. It was shown through several different implementations and test-benches that through the OpenCL framework a large number of specialised resources can be exploited to increase computational performance without significant development time trade-offs.

Another outcome of the project has been the integration of OpenCL based functionality within the High-Throughput Computing environment at the University of Huddersfield, exploiting already existing hardware for General Purpose GPU computing. Exposing the dormant resources available in the HTCondor pool offered not only increased system performance but also facilitated the expansion of the user and application base by allowing for the introduction of much more complex applications within the HTCondor pool. Resulting evidence from this work has shown that the OpenCL platform offers a reliable solution for targeting large, heterogeneous systems, such as those in HTCondor. Platform portability was demonstrated through the seamless execution of the applications across the varied architectures present in the resource pool. Platform specific optimisations are not omitted; their implementation however is left to the judgement of the user/developer. The results of this investigation lead to a peer-reviewed publication at the Emerging Technologies conference in Barcelona. One of the main discussion points during the conference was software sustainability, where interest was shown in both the OpenCL framework, for its platform portability, and the HTCondor for its use as a pool manager for heterogeneous dedicated resources.

The investigation also focused on the FPGA based OpenCL SDK in order to determine the effectiveness of OpenCL programming in reconfigurable computing, the portability of OpenCL applications from CPU/GPU to FPGA, and the development effort involved in setting-up FPGA based OpenCL applications. Although limited in size and scope, results have nonetheless shown that the OpenCL framework has reached the level of maturity needed to allow for the implementation of applications targeting FPGAs. Resulting output shows the potential of FPGAs; however more in-depth research is required to determine the performance gain of the system, not just from a speed-up perspective but also by investigating the power consumption, acquirement cost, and sustainability.

Three research questions were posed in the introduction chapter; the aim of this work has been to offer answers, both through literature research and experimental findings.

The first of these questions was related to the maturity of heterogeneous systems for use in computational tasks. It is concluded, based on reviewed literature of current work done in the field and demonstrated performance increase in specialised accelerators, that heterogeneous computing has reached a sufficient maturity as to offer a promising environment for the computational environment, and warrant a change traditional CPU based computing.

The second questions regarded the feasibility of integrating heterogeneous systems in HPC resources through the OpenCL framework. Based on the successful implementation of the

HTCondor upgrade, the obtained benchmarking data, and the resulting publication, it is inferred that the OpenCL framework is a viable solution for the integration of heterogeneous computing resources in HPC clusters.

The third research question, and topic, sought to determine the usability of the OpenCL framework in conjunction with FPGA architectures for the development of computing applications. It is evidenced, through both research into the functionality of the Altera OpenCL SDK, and the benchmarking implementation on the DE1-SoC device that the OpenCL heterogeneous platform can be successfully used to implement FPGA specific computational applications. The FPGA device used in this work is shown to be faster than the investigated CPU units, however not on par with the GPU devices. It is assumed that once power efficiency is introduced as a component of the system benchmark, the performance gain associated with FPGA usage will become more pertinent.

Based on the research, results, findings, and outcomes of the above work, it is concluded that all original aims and objectives have been achieved, and that a solid groundwork for future research and development, especially on the topic of FPGA based computational acceleration, has been established.

Chapter 9: References

Aldec. (n.d.). *High-Level Synthesis and Verification.* Retrieved 06 2016, from https://www.aldec.com/images/content/products/cyberworkbench_hires_1660.jpg

Altera. (2007). *Accelerating High-Performance Computing With FPGAs.* Altera.

Altera. (2011). *Implementing FPGA design with the OpenCL standard.* Altera.

Altera. (2013). *Radar Processing: FPGAs or GPUs?* Altera.

Altera. (2015). *Matrix Multiplication Design Example.* Retrieved 05 2016, from https://www.altera.com/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html

Altera. (2016). *Altera SDK for OpenCL programming guide.* Retrieved 2016, from https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

AMD. (2014). *OpenCL 2.0 Device Enqueue*. Retrieved 06 2016, from http://developer.amd.com/community/blog/2014/11/17/opencl-2-0-device-enqueue/

AMD. (2014). *OpenCL 2.0 Shared Virtual Memory*. Retrieved 06 2016, from http://developer.amd.com/community/blog/2014/10/24/opencl-2-shared-virtual-memory/

AMD. (2016). *clFFT.* Retrieved 02 2016, from github.com: https://github.com/clMathLibraries/clFFT

Asano, S., Maruyama, T., & Yamaguchi, Y. (2009). Performance comparison of FPGA, GPU and CPU in image processing. *international conference on field programmable logic and applications* , (pp. 126-131).

Bodily, J., Nelson, B., Wei, Z., Lee, D., & Chase, J. (2010). A comparison study on implementing optical flow and digital communications on FPGAs and GPUs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1-22.

Bodily, J., Nelson, B., Wei, Z., Lee, D., & Chase, J. (2010). A Comparison Study on Implementing Optical Flow and Digital Communications on FPGAs and GPUs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1-22.

Brigham, E. O. (1974). *The fast Fourier transform (Vol. 7).* Englewood Cliffs: Prentice-Hall.

CMSoft. (n.d.). Retrieved 06 05, 2016, from http://www.cmsoft.com.br/tutorialOpenCL/schemehostdevices.png

Dafinoiu, A. (2015). *FPGA Based Implementation of Offset PPM.* University of Huddersfield.

Dafinoiu, A., Higgins, J., & Holmes, V. (2016). Accelerating High-Throughput Computing through OpenCL. *Emerging Technologies Conference*, (pp. 46-49). Barcelona.

Embedded.com. (n.d.). *Retargeting embedded software stacks for many-core systems.* Retrieved 06 2016, from http://m.eet.com/media/1176760/rti%20man%20ycore%20fig%203%20500.jpg

Fang, J., Varbanescu, A. L., & Sips, H. (2011). A comprehensive performance comparison of CUDA and OpenCL. *2011 International Conference on Parallel Processing* , (pp. 216-225).

Fykse, E. (2013). *Performance Comparison of GPU, DSP and FPGA implementations of image processing and computer vision algorithms in embedded systems.* Trotenheim: NTNU.

Garland, M., & al., e. (2008). Parallel Computing Experiences with CUDA. *IEEE Micro*, 13-27.

Gubb, D. (2013). *Implementation of a condor pool at the university of huddersfielod.* University of Huddersfield.

Intel. (n.d.). Retrieved 06 2016, from http://cdn.wccftech.com/wp-content/uploads/2015/02/intel_10nm_panel2-Copy.png

Kalinov, A., Lastovetsky, A., & Robert, Y. (2005). Heterogeneous computing. . *Parallel Computing*, 649-652.

Khronos Group. (2016). *The first open standard intermediate language for parallel compute and graphics*. Retrieved 06 2016, from https://www.khronos.org/spir

Livny, M., Basney, J., Raman, R., & Tannenbaum, T. (1997). Mechanisms for high throughput computing. *SPEEDUP journal*, 36-40.

Mazsola. (n.d.). *Field Programmable Gate Arrays (FPGA).* Retrieved 06 2016, from http://mazsola.iit.uni-miskolc.hu/cae/gifs/fig1_6.gif

Microsoft. (n.d.). *Timeout Detection and Recovery (TDR)*. Retrieved 06 2016, from https://msdn.microsoft.com/en-us/library/windows/hardware/ff570087(v=vs.85).aspx

Owens, J. D. (2008). GPU Computing. *Proceeding of the IEEE*, 879-899.

Pellerin, D. T. (1997). *VHDL made easy!* Upper Saddle River: N.J: Prentice Hall.

Perez, B., Bosque, J., Stafford, E., & Beivide, R. (2016). Energy Efficiency Evaluation in Heterogeneous. *Emerging Technologies Conference*, (pp. 50-53).

Schaller, R. R. (1997). Moore's law: past, present and future. *IEEE spectrum*, 52-59.

Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 66-73.

Terasic. (2016, 08). *DE1-SoC User Manual*. Retrieved 08 2016, from http://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=836&FID=3a3708b0790bb9c721f94909c5ac96d6

Tessier, R., Pocek, K., & DeHon, A. (2015). Reconfigurable computing architectures. *Proceedings of the IEEE*, 332-354.

Top500. (2016, 06). *June 2016*. Retrieved 06 2016, from Top500.org: https://www.top500.org/lists/2016/06/

Chapter 10: Appendix

List of files included on portable medium

1. OpenCL Device Detection C Code/Executable(improved version);
2. Matrix Multiplication Host Code/Executable (using external parameters);
3. Matrix Multiplication Host Code/Executable (using user prompted commands);
4. Matrix Multiplication Kernel Code;
5. FFT Client Source, as retrieved from (AMD, 2016);
6. FFT Client Compiled(with batch script);
7. FPGA Matrix Multiplication Host Code(with modifications);
8. FPGA Matrix Multiplication Kernel Code (Altera, 2015);
9. FPGA Matrix Multiplication Compiled Files.

N.B: The Matrix Multiplication kernel code (4) is also present with the executable in (2 & 3).