Program Acceleration in a Heterogeneous Computing Environment Using OpenCL, FPGA, and CPU

Herman Noel Hoffman
University of Rhode Island, herman.hoffman1@gmail.com

PROGRAM ACCELERATION IN A HETEROGENEOUS COMPUTING ENVIRONMENT USING OPENCL, FPGA, AND CPU

BY

HERMAN NOEL HOFFMAN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

UNIVERSITY OF RHODE ISLAND

2017

MASTER OF SCIENCE IN COMPUTER ENGINEERING THESIS

OF

HERMAN NOEL HOFFMAN

APPROVED:

Thesis Committee:

Major Professor	Resit Sendag
	Jien-Chung Lo
	Lutz Hamel
	Nasser H. Zawia
	DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2017

ABSTRACT

Reaching the so-called “performance wall” in 2004 inspired innovative approaches to performance improvement. Parallel programming, distributive computing, and System on a Chip (SOC) design drove change. Hardware acceleration in mainstream computing systems brought significant improvement in the performance of applications targeted directly to a specific hardware platform. Targeting a single hardware platform, however, typically requires learning vendor and hardware-specific languages that can be very complex. Additionally, Heterogeneous Computing Environments (HCE) consist of multiple SOC hardware platforms, so why not use them all instead of just one? How do we communicate with all platforms while maximizing performance, decreasing memory latency, and conserving power consumption? Enter the Open Computing Language (OpenCL) which has been developed to harness the power and performance of multiple SOC devices in an HCE. OpenCL offers an alternative to learning vendor and hardware-specific languages while still being able to harness the power of each device. Thus far, OpenCL programming has been directly mostly at CPU and GPU hardware devices.

The genesis of this thesis is to examine the connections between parallel computing in a HCE using OpenCL with CPU and FPGA hardware devices. Underlining the industry trends to favor FPGAs in both computationally intensive and embedded systems, this research will also highlight the FPGA specifically demonstrating comparable performance ratings as CPU and GPU at a fraction of the power consumption. OpenCL benchmark suites run on a FPGA will show the importance of performance per watt and how it can be measured. Running traditional parallel programs will demonstrate the power and portability of the OpenCL language and how it can maximize performance of FPGA, CPU, and GPU. Results will show that OpenCL is a solid approach to exploiting all the computing power of a HCE and that performance per watt matters in mainstream computing systems, making a strong case for further research into using OpenCL with FPGAs in a HCE.

ACKNOWLEDGMENTS

First and foremost, I would like to give special thanks to my wife of 28 years who has been supportive of me and understanding of the long nights spent doing research after work. I also would like to give thanks to our beautiful children who inspire me every day to push on and set a good example for them to follow. Finally, I would like to acknowledge my major professor, Resit Sendag for all of his assistance during this endeavor. He is an impeccable teacher with a passion for learning, teaching, and inspiring his students to achieve. Without his ideas and countless discussions, I would not have been able to achieve the level of analysis in computing hardware and software that I have reached today.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGMENTS iv

TABLE OF CONTENTS v

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1 1

INTRODUCTION 1

Motivation for Research 1

Chapter Overview 2

CHAPTER 2 5

BACKGROUND 5

Parallel Programming 6

Speed and Performance 9

Heterogeneous Computing Environment 11

FPGA Programming and Design 16

CHAPTER 3 18

METHODOLOGY 18

Research Setup 19

Altera SDK and DE1-SoC Development Board 20

OpenCL Architecture 22

OpenCL Runtime and FPGA Programming 26

Maximizing Speed, Performance, and Portability 33

Converting from C to OpenCL 33

CHAPTER 4 36

FINDINGS 36

Vector Addition Program 36
Black Sholes Program 41
Rodinia Benchmarks 44
CHAPTER 5 49
CONCLUSION 49
APPENDIX A 52
APPENDIX B 57
APPENDIX C 63
APPENDIX D 66
APPENDIX E 68
BIBLIOGRAPHY 72

LIST OF TABLES

TABLE PAGE

Table 4.1 Relative performance adjustments. 39

LIST OF FIGURES

FIGURE PAGE

Figure 2.1 Processor performance growth 7

Figure 2.2 Amdahl’s law versus Gustafson’s law........ 8

Figure 2.3 Multiple cores and processors. 13

Figure 2.4 Heterogeneous concept. 15

Figure 3.1 OpenCL Heterogeneous platform. 20

Figure 3.2 DE1-SoC block diagram 22

Figure 3.3 OpenCL platform with devices 24

Figure 3.5 OpenCL memory model 26

Figure 3.6 Find platform and devices 28

Figure 3.7 Create the context 28

Figure 3.8 Create a command queue 28

Figure 3.9 Create memory objects 29

Figure 3.10 Transfer data to device 29

Figure 3.11 Create and compile from source 30

Figure 3.12 Kernel extraction 30

Figure 3.13 Kernel execution 31

Figure 3.14 Copy data back to host 31

Figure 3.15 Release resources 32

Figure 3.16 Gaussian Box-Muller function 34

Figure 3.17 Kernel target 35

Figure 4.1 Vector addition pipeline 37

Figure 4.2 Change in data bandwidth 39

Figure 4.3 Relative performance 41

Figure 4.4 Gaussian Box-Muller function 42

Figure 4.5 Kernel target 42

Figure 4.6 Host arrays. 43

Figure 4.7 Black Sholes results. 44

Figure 4.8 K-Means FPGA optimization. 46

Figure 4.9 Merge sort. 47

Figure 4.10 Bucket sort. 48

CHAPTER 1

INTRODUCTION

Making computer hardware and software perform faster has been the overarching goal of all development in the fields of computer engineering and computer programming. After reaching the “performance wall” in 2004, there was a paradigm shift from hardware to software development. New programming concepts emerged such as instruction, data, and task level parallelism aimed at faster program execution, or better performance. Hardware acceleration emerged thanks much in part to the gaming industry’s need to have near to real graphics in their video games.

Program acceleration is the idea of achieving the best program performance by utilizing all tools available, both hardware and software. Programmers create code that is clean, efficient, and built to run as parallel as possible in any computing environment. This idea is the motivation behind the Heterogenous Computing Environment (HCE) concept and the creation of a language to maximize its capabilities, OpenCL. This thesis will briefly discuss the connection between parallel computing and heterogeneity in mainstream computing as a precursor to understanding the need for OpenCL and why is somewhat revolutionary. This brief background in parallel programming and heterogeneity will be followed by program and data analysis using OpenCL in an HCE consisting of a CPU, GPU, and FPGA.

Motivation for Research

GPUs have cornered the lion’s share of developmental research due to their ability to process large amounts of data as well as their physical presence in nearly every mainstream computing system. Up until the release of OpenCL, developers shied away from FPGAs due to the complexity of programming needed just to perform simple functions. Within the past few years, some research has been conducted using OpenCL and FPGA acceleration such as information filtering[1], fractional video compression[2], finite impulse filters[3]. More research is warranted, however, due to the increased utilization of FPGAs and since they are the best choice in executing highly parallel programs while consuming the least amount of power possible. Measuring program execution speed is one measure of performance, but the amount of wattage consumed to achieve that speed is also critical and is referred to as performance per watt. This research will demonstrate how it has never been easier to program hardware thanks to OpenCL and demonstrate the performance enhancement and power savings of FPGA over CPU and GPU in an OpenCL controlled environment. OpenCL in a heterogeneous computing environment enables computer programmers and engineers alike to maximize acceleration and performance across all hardware platforms. This research will substantiate why OpenCL should be used for program and hardware acceleration in a HCE.

Chapter Overview

Chapter 2 provides the background information and research as to the significance of OpenCL as a computing language. It reviews the main literature used in this thesis and research, reviews parallel programming and how it ties into OpenCL,

[1] Doris Chen, *Using OpenCL to Evaluate the Efficiency of CPUs, GPUs, and FPGAs for Information Filtering*. (Invited Paper, Altera Toronto Technology Center, 2013).
[2] Doris Chen, *Fractional Video Compression in OpenCL: An Evaluation of CPUs, GPUs, and FPGAs as Acceleration Platforms*. (Invited Paper, Altera Toronto Technology Center, 2013).
[3] Desh Singh. *Higher Level Programming Abstractions for FPGAs using OpenCL*. (Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing, 2011).

analyzes speed and performance gains as they apply to hardware, and highlights FPGA as the de facto standard in power consumption savings in an HCE.

In Chapter 3, a detailed explanation of the methodology is brought forth. The OpenCL architecture is introduced in detail and how it bonds the components of the HCE. As supporting evidence for this research, existing OpenCL programs will be altered to achieve maximum performance results and demonstrate the portability of OpenCL programs and kernels. To illustrate the simplicity of hardware programming using OpenCL, some existing OpenCL programs will be compiled and run on FPGA. A feature program will demonstrate the speed and performance gains of using FPGA as a hardware platform. The Software Development Kit (SDK) for Altera will also be introduced since its understanding is crucial to visualize how the hardware and software communicate in the OpenCL controlled HCE.

Chapter 4 will validate the benefits of using OpenCL and FPGA in an HCE. First, comparisons will be made between the results of running a traditional C program on CPU versus FPGA after conversion to OpenCL. The performance will be measured in terms of performance per watt since FPGAs are the targeted hardware device in this research. Additionally, the results of running OpenCL specific benchmarks on the DE1-SoC board will be analyzed to show how FPGAs can thrive in an HCE.

Finally, chapter 5 consists of conclusions and recommendations for further research. Included in these discussions are the current developments of the FPGA and increase in its application. Also discussed is the explosion of OpenCL in the developmental community and progress up to the current version of 2.2. This research aims to justify the importance of OpenCL, HCE, and ultimately FPGAs.

CHAPTER 2

BACKGROUND

Literature Review

Having no prior experience with the OpenCL language inspired be to purchase the two books that I used for my research of the OpenCL language and how it works in the HCE. Heterogeneous Computing with OpenCL 2.0 by David Kaeli et al is a very well written and easy to follow approach to learning OpenCL. It explains the structure of the OpenCL architecture, runtime execution and memory models. The example code provided highlights some of the beneficial features of OpenCL and serves as a shell that can be used to build more complex programs. The other OpenCL textbook I purchased is OpenCL Programming by Example written by Banger and Bhattacharyya. It takes a similar approach to teaching OpenCL along with example code, diagrams, and explanations. Both books are highly recommended to gain a basic understanding of OpenCL.

Driving deeper into an understanding of OpenCL and how to use it also requires an extensive review of vendor specific APIs, user manuals, and white papers depending upon which type of hardware you are using. My research made use of an FPGA board from Altera which required not only the download of a Software Development Kit (SDK) from Altera but all of the reference materials that explains its use. Since Khronos Group manages the OpenCL API specifications it is necessary to download their reference material that aligns with the version of OpenCL being used. Their main website also maintains both deprecated and up to date versions of the

OpenCL languages along with header files, source code, and links to forums for development assistance.

The last major literature references used in this research were *Computer Organization and Design* and *Computer Architecture* both by Patterson and Hennessey. These books are the gold standard to understanding computer hardware architecture, as well as the interface between hardware and software. I utilized them mostly for the comprehension and application of parallel programming and how it relates to OpenCL. Much of the inspiration for this research spawned from the concepts and progressive thinking of these two authors.

Parallel Programming

The need to focus more on software versus hardware to accelerate program execution and efficiency came to fruition after processor manufacturers reached the "performance wall" around 2004. Figure 2.1 shows how the increase of performance of single processors and memory configuration flatlined. Sequential programming techniques such as the "divide and conquer" method, which divided a single program into smaller subsets or groups of code executed separately, were precursors to the many classes of parallelism and parallel architectures that exist today. Object oriented computing languages such as Java and C++ also were designed initially to speed up sequential program execution, but suffered from too many data dependencies between function calls to exploit the essence of true parallelism. Instruction level parallelism was used on superscalar and out of order execution uniprocessors, but it only increased the execution speed of traditional sequential programs.[4]

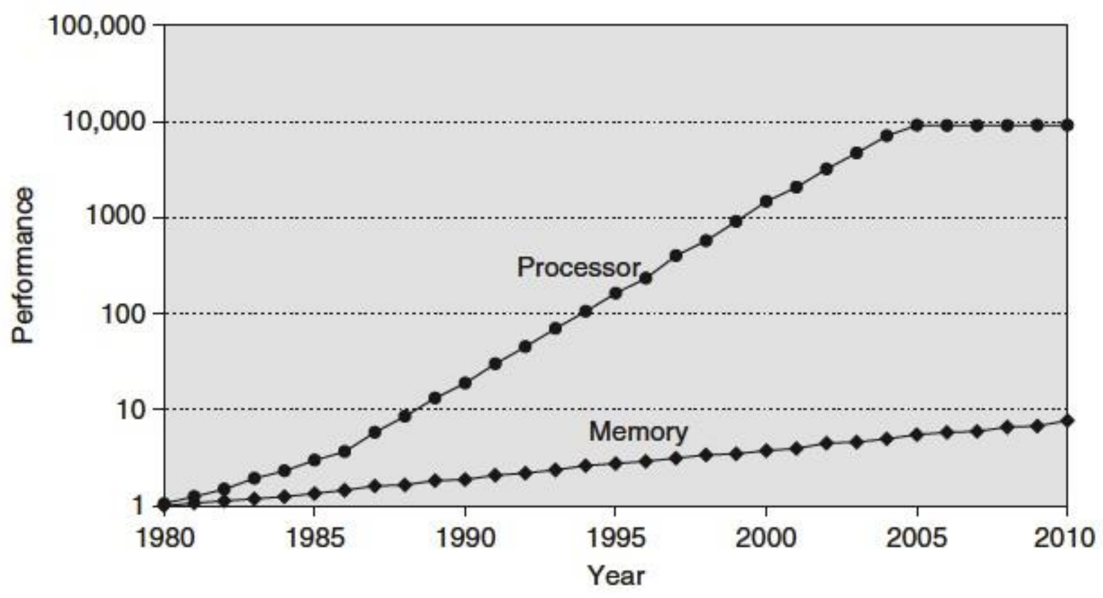


Figure 2.1 Processor performance growth.

Around 2006, chip developers began placing multiple processors or cores onto one chip. Though the idea of having multiple computers work together on one program was not a new concept, the era of multiple cores had begun. This opened the door for parallel processing programs (running one job on multiple processors) and job level parallelism (running multiple jobs on multiple processors).[5] Within the realm of the personal computer, however, these multi-core machines are typically confined to sharing the same memory or physical address space. Clusters and grid computing were also introduced which are commonplace among online databases,

[4] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface* (New York: Morgan Kaufmann, 2009), 633.
[5] Ibid, 632.

email servers, and search engines. These approaches, however, only offer solutions for large scale computing environments and not for your average consumer running programs on the home computer.

Parallel programming is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (i.e. in parallel). [6] Every job or task that needs to be accomplished brings its own level of complexity based on the algorithm, the programming language being used, and the hardware available. Different classes of parallelism have emerged that are aimed at exploiting specific hardware structures and software designs. Multithreading, for example, supports multiple threads executing in an interleaving fashion on a single-issue processor. [7] With the rise of multi-core platforms, task level parallelism was developed to leverage running sections of code or different tasks on separate processors. Yet another form referred to as request level parallelism may be exploited by a single application running on multiple processors, such as a database responding to queries, or multiple applications running independently, often called multiprogramming.[8]

The question becomes, if we have different classes of parallel programming to handle various types of applications, and multiple cores on which to run our programs, then why would we need to create additional programming languages such as OpenCL? To answer this question, we need to expand our aperture to view the entire realm of hardware and software. One problem that has emerged from using multiple cores draws a striking resemblance to the processor wall problem of 2004. There are

[6] (Kaeli, et al. 2015)
[7] (Patterson and Hennessy, Computer Architecture: A Quantatative Approach 2012) page 345
[8] (Patterson and Hennessy, Computer Architecture: A Quantatative Approach 2012) page 345

physical limitations to how many cores can be placed on a board, and even though cores designed today have their own data memory they still are competing with other cores for the same physical address space which is controlled by the CPU.

Additionally, Single Program Multiple Data (SPMD) models work well to handle running tasks in parallel but matters become much more complicated when data needs to be shared and synchronized between the tasks. With the release of OpenCL 2.0, shared virtual memory is used between program and device that also supports access to data across tasks being executed on separate devices.

Speed and Performance

Before we move on to discuss the significance of running OpenCL in an HCE, we need to review some basic concepts revolving around speed and performance. In the past decade, considerable strides have been made in increasing the overall speed and performance of programs being executed on hardware accelerators such as the GPU and FPGA boards. Where development based upon Amdahl's law of speedup in latency of task execution on a fixed workload stops, Gustafson's law continues and proves to be a more realistic formula for calculating parallel performance, especially as it relates to the basis of this research.

Gustafson's law is formulated as:

$$S_{latency}(s) = 1 - p + sp$$

Where:

- $S_{latency}$ is the theoretical speedup in latency of the execution of the whole task
- S is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system
- P is the percentage of the execution workload of the whole task that benefits from the improvement of the resources of the system before the improvement

Gustafson's law shows that even within a constant time interval, the complexity of programming can increase as long as the quantity or quality of resources being used to compute the program increases, such as the number or performance capability of processors for example. Figure 2.2 shows this along with a comparison between Amdahl's Law and Gustafson's Law.

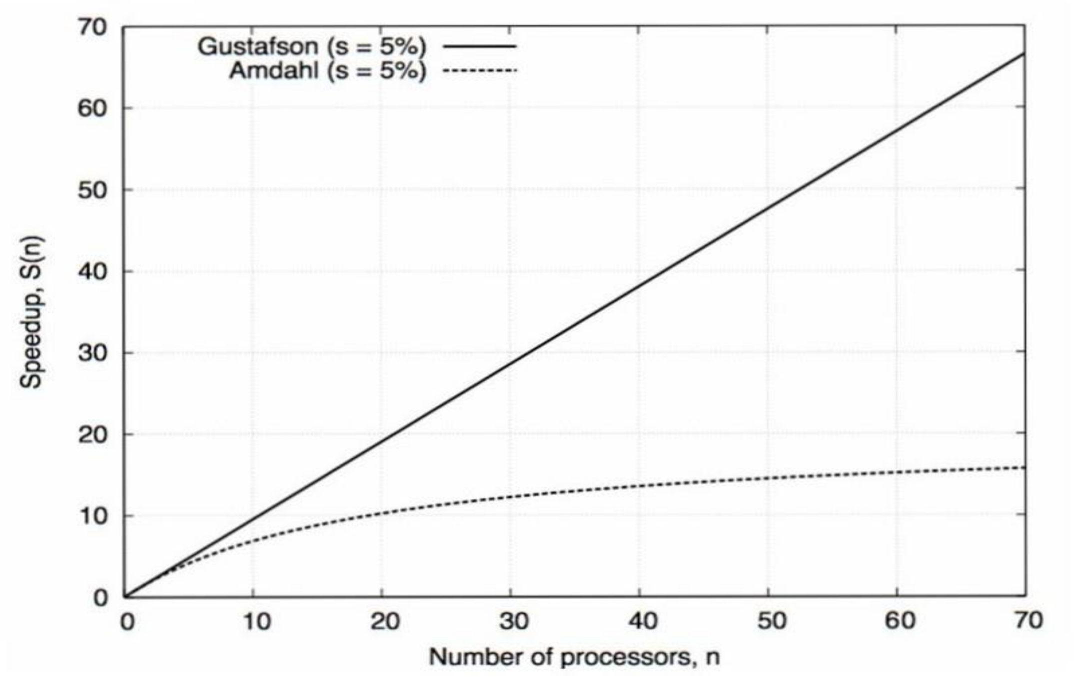


Figure 2.2 Amdahl's law versus Gustafson's law.

Viewing Gustafson's law from an HCE perspective, utilizing multiple devices to execute a program would give you access to more processing power, resulting is a somewhat linear speed-up curve.

Accomplishing more in a constant interval of time relates well to speed, but what about performance? As we can infer from previous discussion, the definition of performance in the world of computers has changed over the years as well. Prior to 2004, performance was determined by the clock rate of the processor measured in MHZ. The higher the clock rate, the faster the computer performed. This doesn't hold true today with the shift toward multi-core design, efficiency in software development, and the ever-increasing use of embedded computers. Embedded computing calls for the need of processing power in the most austere locations; automotive industry, MP3 players, digital watches, traffic lights, etc. Many of these systems run on batteries so the need to conserve power becomes top priority. Again, this is where we tie the importance of this research in to what is most important in the real world. Not only is maximizing processing power in a HCE important, but power consumption must also be kept at a minimum as the tech industry designs more and more applications of embedded computers that run off of battery power.

Heterogeneous Computing Environment

Now that we have reached the undeniable conclusion that more processing power is better, let's discuss how that processing power can or should be arrayed. Heterogeneous computing is the concept of using multiple "devices" with their own processor and memory capability to execute programs, tasks, or functions independently or concurrently with other devices. These devices include multi-core CPUs, GPUs, FPGAs, and digital signal processors.[9] As an example, GPUs have been used for decades as independent processing units to enhance computer generated graphics for gaming. Graphic generation and rendering require complicated floating point calculations that continually place a high demand on computing resources. Since GPUs are equipped with their own on-chip memory and processor, they opened the door for speed ups in graphics processing during hardcore gaming, especially 3D rendering. Modern GPUs have anywhere from 32 to 64 individual cores placed directly on the chip. NVIDIA Corporation developed their own API called CUDA to bind high-level computing languages such as C, C++, and FORTRAN with the massively parallel computing power of their GPUs.

The need for heterogeneous computing has increased exponentially over the past decade. Instead of relying on one processor to perform at the highest possible speed, multiple processors are utilized to achieve faster program execution. Parallelism in programming requires additional processors to gain a true advantage over traditional sequential programming that has dominated software development for decades. As shown in Figure 2.3, these additional processors can be found within the multiple cores of all mainstream computing systems.

[9] David Kaeli et al., *Heterogeneous Computing with OpenCL 2.0* (New York: Morgan Kaufmann, 2015), 1.

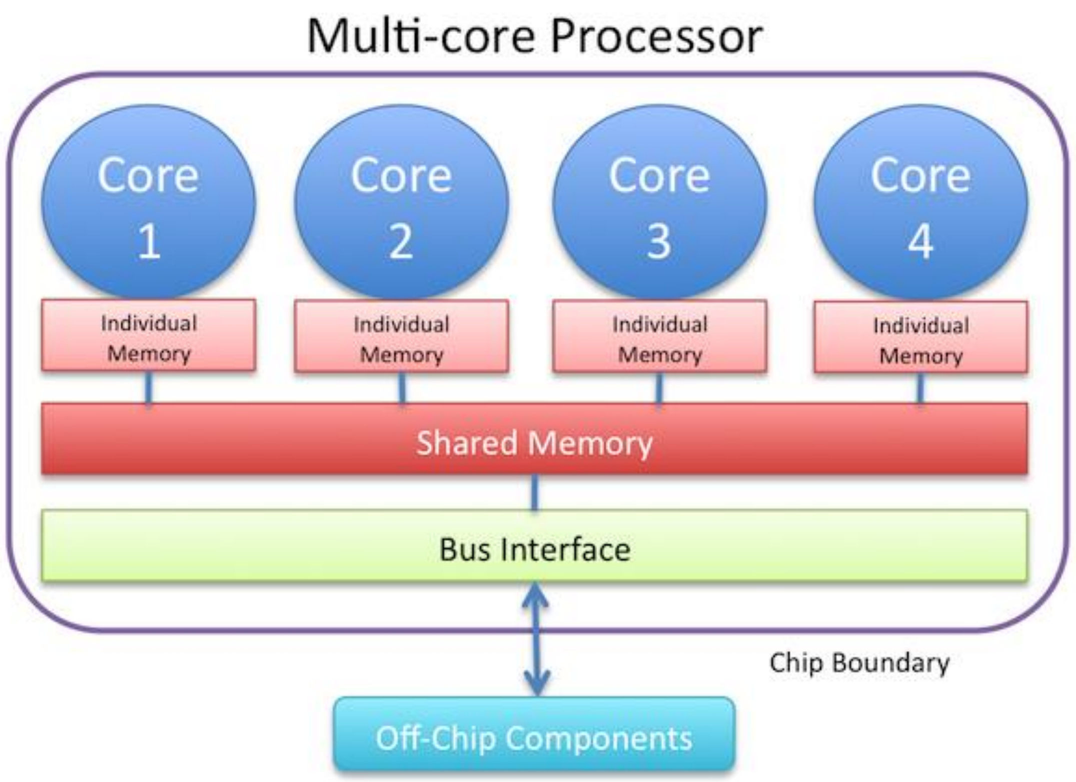


Figure 2.3 Multiple cores/processors.

Besides multiple cores, hardware components of computers today typically have their own memory and processor(s) located directly on the circuit board. SoC design takes advantage of spatial and temporal locality to reduce memory latency and achieve maximal speed-up. As mentioned earlier, manufacturers such as NVIDIA offer GPU cards for laptops and desktop computers that have their own CPU and memory on the same die which greatly reduces memory latency, data dependencies, and data conflicts. The DE1-SoC board used for this research has a dual-core ARM Cortex A9 MP Core processor coupled with a 1GB DDR3 SDRAM as part of the Hard Processor System (HPS). All of these processors or cores located in the various hardware components, along with any external boards that can be connect via PCIE or USB

UART connections, provide additional computational power that can be utilized to support data and thread parallelism. These “self-contained” constructs of processor and memory constitute the “devices” of a heterogeneous computing environment.

Each device in a heterogeneous environment handles the processing of complex applications differently. Complex applications can be vaguely categorized based upon the workload that they place on the device being used. Control intensive applications include searching, parsing, and sorting operations; data intensive applications focus more on image processing, simulation and modeling, and data mining; and compute intensive applications involve iterative methods, numerical methods, and financial modeling.[10] The complexity of the task at hand will determine which device is best suited for executing the task, as seen in Figure 2.4 which illustrates how a computer may handle data parallel versus serial and task parallel workloads. CPUs are typically best suited for control intensive applications while GPUs excel at processing imagery due to the massively parallel design that handle processing large amounts of data. FPGAs are also inherently parallel and handle complex parallelism with a lower consumption of power than both CPU and GPU platforms. An example of this will be examined exclusively in Chapter 4.

[10] David Kaeli et al., *Heterogeneous Computing with OpenCL 2.0* (New York: Morgan Kaufmann, 2015), 1.

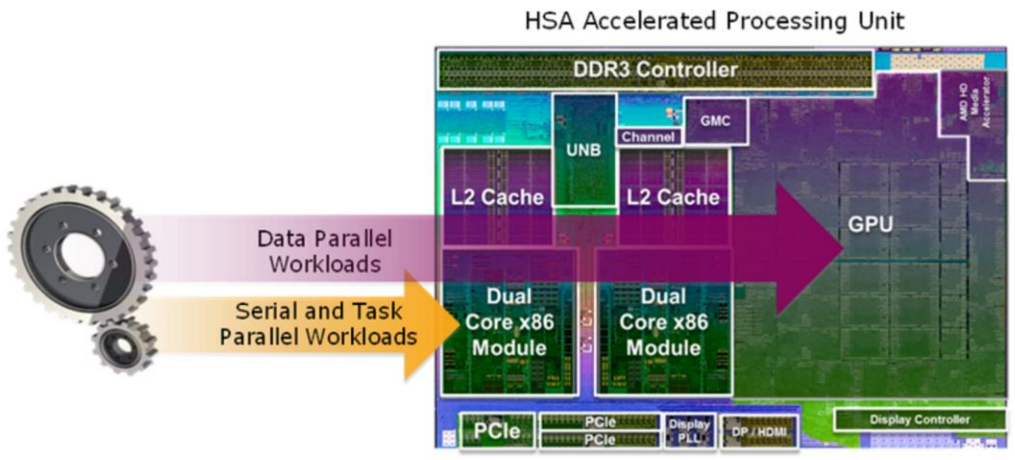


Figure 2.4 Heterogeneous concept.

Workload diversification has opened the door for improving performance and lowering overall power consumption for applications such as HD video conferencing and real-time language translation. Farming out the parallel processing chores to the GPU while keeping the operating system requirements and serial tasks with the CPU enables maximization of both devices, which is the essence of heterogeneous computing.

Now that the layout of an HCE is clear, let's look at an acceleration of the hardware devices. In order to tap into the acceleration power of hardware such as the multi-core CPUs, GPUs, or FPGA one must acquire an in-depth knowledge of high-level (C, C++, etc.), hardware (VHDL for FPGA), or even vendor specific (CUDA) languages which can be very complex and are only applicable to the one specific piece of hardware. If only there was a way to accelerate any hardware device by utilizing one programming language. This is precisely what OpenCL offers, which will be illustrated in Chapter 3.

The use of a heterogeneous computing environment coupled with the OpenCL language and FPGA, offers an alternative framework for maximizing performance, decreased power consumption, and workarounds to the aforementioned challenges.

FPGA Programming and Design

FPGA design has evolved significantly over the past few decades. In the past, using an FPGA in your development environment required extensive programming just to get your FPGA to perform some simple functions. As a result, FPGAs have been by in large avoided by program developers.[11] FPGAs were the forerunners of the gaming industry, back when games such as Space Invaders and Pac-Man ruled the arcade. Designed as simple two-dimensional arrays of logic gates connected in parallel that are field programmable, they can perform complex computations at a fraction of the performance cost of its hardware brethren. The large number of logic elements typically found on an FPGA provide the means for the multi-threaded parallelism. Modern FPGA design such as the DE1-Soc board from Altera used in this research contains over a million logic elements and thousands of memory blocks in a parallel design that enables multiple OpenCL workgroups to be processed concurrently.

Prior to OpenCL, programming an FPGA required learning complex hardware languages such as HDL or VHDL along with using Electronic Design Automation (EDA) tools in order to properly convert your design idea into a complex logic circuit that programmed the FPGA board. With the advent of the OpenCL language which

[11] (Moore 2014), Page 7

we will discuss in more detail in Chapter 4, FPGAs can now be programmed without having to learn HDL or VHDL. The SDK supplied by the vendor of each OpenCL compatible FPGA board handles the chore of creating the complex logic circuit needed to program the FPGA board. The SDK utilizes a software program in the background such as Quartus II for FPGA in order to create the hardware configuration file.

As mentioned earlier, the FPGA is becoming the hardware processor of choice in embedded applications where power is at a premium. The FPGA has a fine-grain parallelism architecture, and by using OpenCL you can generate only the logic you need to deliver one fifth of the power of the hardware alternatives.[12] It is clear that there is a solid connection between OpenCL and FPGA that may see a sharp increase in FPGA application in environments beyond the embedded market.

[12] (Moore 2014), page 39.

CHAPTER 3

METHODOLOGY

Program acceleration can be achieved in many ways. No one way will work all of the time. Sometimes data level parallelism is more critical than job level parallelism, or image processing enhancement is most crucial vice the speed at which linear algebra problems are calculated. But it is possible that there are alternate ways to accelerate hardware in the interest of achieving maximum results with minimal power consumption. The goal here is to shed light on a new paradigm that attempts to tackle complex programs in the most advantageous way possible instead of being constrained to one device. This research is rooted in five key reasons to substantiate the argument for using OpenCL to program hardware, especially FPGA, in an HCE:

1. Heterogeneity in program design provides access to more processing power
2. OpenCL makes programming hardware easier
3. OpenCL provides program portability along with speed and performance advantages
4. FPGAs can perform just as fast as GPUs and CPUs at a fraction of the power consumption
5. Parallel programming exploitation using OpenCL

Reason one was covered with a basic explanation of a heterogeneous computing environment and how it correlates to OpenCL. A "Hello World" example will be used to not only template the basic functions of the OpenCL API, but to also demonstrate just how simple it is to program hardware such as the FPGA. Portability, speed, performance, and power savings will be demonstrated using a vector addition program run in the HCE. Wrapping up the supporting arguments will be a black sholes financial options pricing program converted from C to OpenCL to illustrate how parallelization of existing software and hardware can be exploited.

Research Setup

The components of my heterogeneous computing environment and the interconnectivity of the hardware and software components are diagramed in Figure 3.1. The hardware and software used consisted of the following:

Hardware

- CPU: 4 cores (Intel® Core i3 4010U @ 1.70 GHz operating on a 64-bit Windows 8 OS with 4 GB RAM)
- GPU: Intel R HD Graphics 4400 @ .2 GHz
- FPGA: DE1-SoC Development and Education Board from Altera, obtained through the Altera University Program (Altera Cyclone V FPGA with ~85K Logic Elements, .8GHz, Dual-Core Arm Cortex A9 MP Core Processor, with 64MB SDRAM and 1GB of DDR3 Ram)

Software

- Altera SDK for OpenCL version 15.1
- Quartus II Prime 15.1
- NIOS II EDS 11.0
- Microsoft Visual Studio 2015

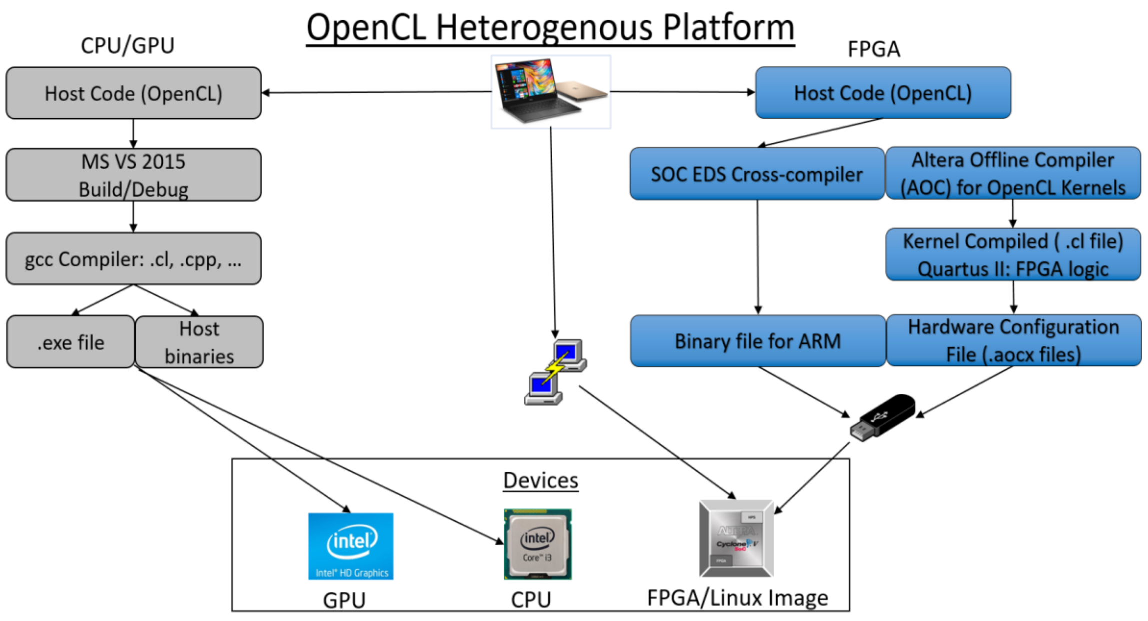

OpenCL Heterogenous Platform
CPU/GPU
FPGA
Host Code (OpenCL)
MS VS 2015 Build/Debug
gcc Compiler: .cl, .cpp, …
.exe file
Host binaries
Host Code (OpenCL)
SOC EDS Cross-compiler
Altera Offline Compiler (AOC) for OpenCL Kernels
Kernel Compiled (.cl file) Quartus II: FPGA logic
Binary file for ARM
Hardware Configuration File (.aocx files)
Devices
GPU
CPU
FPGA/Linux Image

Figure 3.1 OpenCL heterogeneous platform.

Altera SDK and DE1-SoC Development Board

In order to build and run programs within an OpenCL environment, you need to install a SDK that is vendor specific to the type of hardware in your HCE. Since the DE1-SoC board from Altera is used in this research, it was necessary to install the Altera SDK for OpenCL to build and run kernels on that board. The Altera SDK provides the logic components, drivers, and AOCL-specific libraries and files. The logic components include the Altera Offline Compiler (AOC), the AOCL utility, and the host runtime. The AOC creates the hardware configuration file that programs the FPGA, the AOCL utility contains the high-level commands that perform tasks such as diagnostic tests, and the host runtime consists of libraries that provide OpenCL APIs and helper libraries.[13]

As mentioned earlier, the DE1-SoC board has an Altera Cyclone V FPGA with a 800MHz Dual-Core Arm Cortex A9 Processor directly on the board. To run OpenCL kernels on the board, they are loaded directly to the FPGA as a hardware configuration file (.aocx) via the USB port along with a version of the host binary suitable for FPGA. The execution of the kernel can either be controlled by the source program (main) or by using a Putty terminal connection between the host computer and the FPGA. The Putty connection enables direct access to the memory and process embedded on the FPGA board via a bootable Linux image running off of a micro-SD card. Figure 3.2 shows the full block diagram of the DE1-SoC board.

[13] Altera Corporation. Altera SDK for OpenCL: Getting Started Guide. Instruction Manual, San Jose, 2015, page 1-3.

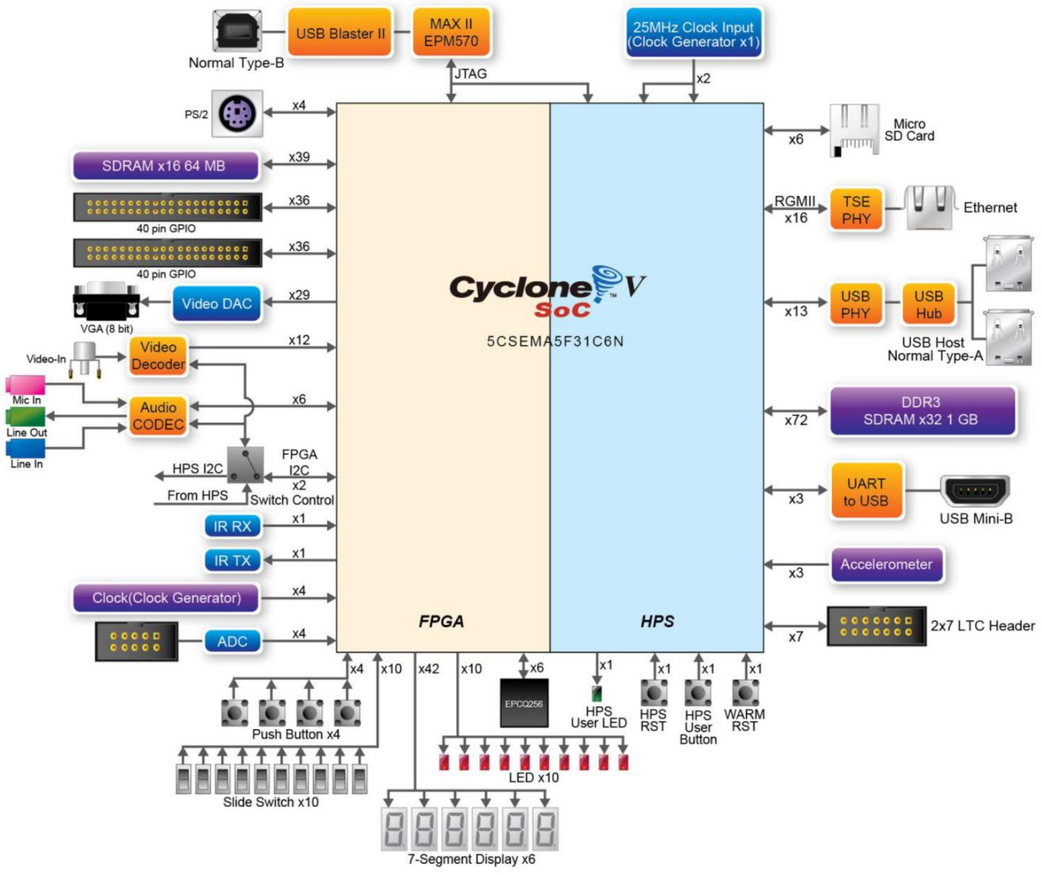


Figure 3.2 DE1-SoC block diagram.

OpenCL Architecture

Open Computing Language (OpenCL) is not only designed to communicate with all devices in a heterogeneous computing environment, but also direct the execution of programs by assigning programming tasks to the best device available.[14] It bridges the gap between the computing power of multiple devices and serving as one computing language that software programmers can learn fairly quickly to interact directly with hardware. Created by Apple, it is an open source language that was first released in

[14] David Kaeli et al., *Heterogeneous Computing with OpenCL 2.0* (New York: Morgan Kaufmann, 2015), 1.

2008 by the Khronos Group who still manages the libraries and releases. Khronos developed the OpenCL standard so that an application can offload parallel computation to accelerators in a common way, regardless of their underlying architecture or programming model.[15] Not only is it designed to support the heterogeneous computing environment, it also affords cross-platform portability. A program written in OpenCL can be run on virtually any machine that has an OpenCL SDK and applicable libraries installed.

The OpenCL C language is a restricted version of the C99 language, but it also has wrappings that support C++, Java, Python, and NET.[16] The specification of OpenCL is divided into four parts: the *platform* model, the *execution* model, the *kernel* programming model, and the *memory* model.

The OpenCL *platform* consists of the host computer and all *devices* that are connected to it. To recap, for OpenCL purposes a *device* is a piece of hardware that is capable of processing a kernel independently of or concurrently with other devices. Each device is divided into compute units that are further divided into processing elements as shown in Figure 3.3.[17]

[15] (Guillon 2015), page 2.
[16] (Kaeli, et al. 2015), page 42.
[17] Ibid.

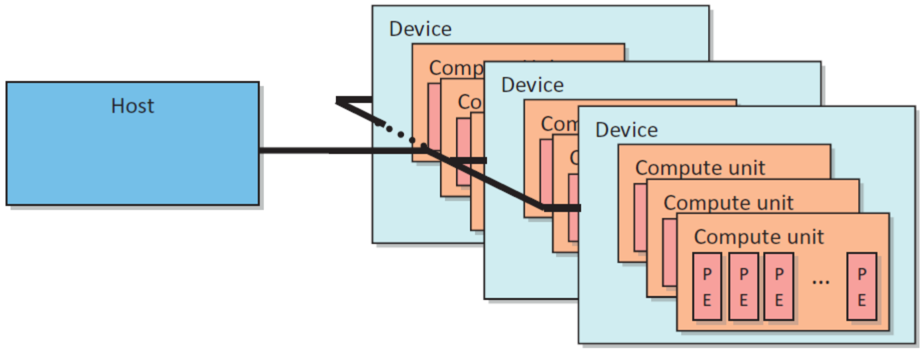


Figure 3.3 OpenCL platform with devices.

Within the *execution* model, the host configures the OpenCL environment and a *context* is established around the platform and the devices through which coordination and memory management are handled for kernel execution. It also establishes *command-queues* that handle communications between the host and the device as well as *events* that specify dependencies between commands. [18]

The *kernel* programming model construct builds kernels using items such as *work-items* and *work-groups* to obtain maximum parallelization based upon the targeted device. The host is responsible for compilation of the main source code using a standard C compiler and loading the host binaries into memory. The main source code contains the OpenCL commands and function calls, contains any constant data, and controls the execution of a kernel on a device. Runtime compilation prepares the kernel to be run on the best device based on computational workload which maximizes both concurrency and parallelism.[19] After the kernels are created, they are compiled by an offline compiler to create the hardware configuration files (.aoco and .aocx). The

[18] Ibid, page 48.
[19] Ibid, 1.

.aocx file is used to program the FPGA enabling it to run the kernel. It takes the place of the complicated VHDL programs of the past that were need to program an FPGA. Figure 3.4 shows a typical flow of both the host code and the kernels through the programming process assuming a PCIE mounted FPGA.

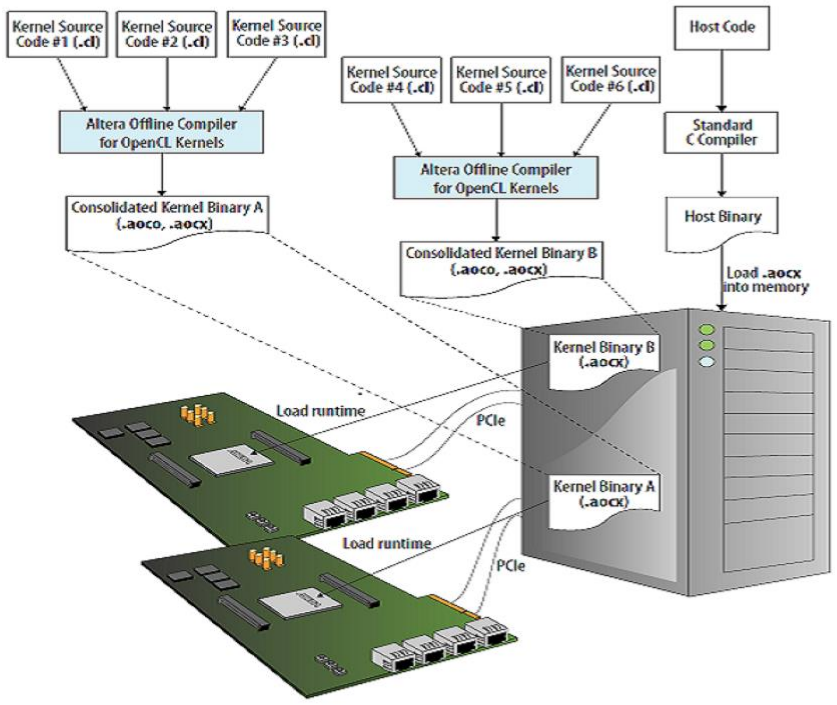

Kernel Source Code #1 (.cl)
Kernel Source Code #2 (.cl)
Kernel Source Code #3 (.cl)
Host Code
Kernel Source Code #4 (.cl)
Kernel Source Code #5 (.cl)
Kernel Source Code #6 (.cl)
Altera Offline Compiler for OpenCL Kernels
Standard C Compiler
Altera Offline Compiler for OpenCL Kernels
Consolidated Kernel Binary A (.aoco, .aocx)
Host Binary
Consolidated Kernel Binary B (.aoco, .aocx)
Load .aocx into memory
Kernel Binary B (.aocx)
Load runtime
PCIe
Kernel Binary A (.aocx)
Load runtime
PCIe

Figure 3.4 AOCL FPGA programming flow.

The *memory* model consists of *buffers*, *images*, and *pipes* that handle memory allocation, temporary storage of data, and prioritization of how data items are stored. OpenCL defines memory regions as either host or device specific. *Global* memory (DDR and QDR) is visible to all work items executing a kernel, *constant* memory stores data that remains constant, *local* memory (on-chip memory) shares data between *work-items*, and *private* memory (on-chip registers) is unique to an individual *work-item.*[20] Figure 3.5 shows the layout of host and device memory regions within OpenCL.

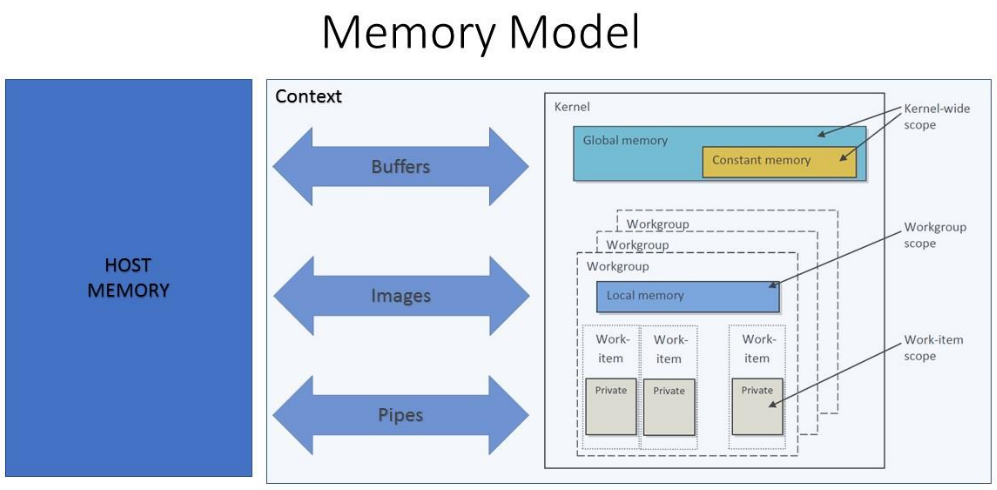


Figure 3.5 OpenCL memory model.

OpenCL Runtime and FPGA Programming

When learning a new computer language, most authors and instructors start off with a "Hello World" version of the code that shows a few basic commands, function calls, include files, and the like. OpenCL is no exception. In this case, the hello world program is used to demonstrate how the OpenCL architecture interacts with the associated hardware in your HCE. A full listing of the "Hello World" code is provided in Appendix A. We will examine this program by breaking it down into smaller sections of OpenCL code that align with the application steps that all OpenCL

[20] Ibid, page 60.

programs follow. The final step will be to convert it to run on the DE1-SoC FPGA, demonstrating just how easy it is to program hardware with OpenCL.

Since OpenCL is basically a derivative from the C language if follows the same principles of utilizing source and headers files. In the source file, there is one *main* function that controls the order of program execution along with the program-specific function definitions and declarations. It also provides space to declare host-side memory functions and operations, variables, and constants. The bulk of the standard OpenCL function definitions and declarations are located in a cluster of headers files that are continually updated as newer versions of OpenCL are released. All OpenCL header files are available for download from the Kronos Group website as well as popular developer websites such as GitHub.

As stated earlier, all OpenCL programs follow ten main steps that setup the OpenCL environment and run kernels on existing devices.[21] Below is a detailed breakdown of each step along with the associated code.

1. Discovering the platform and devices. In order for a kernel to run on a device, the host must first determine if an OpenCL platform is present and how many devices are associated with it. Figure 3.6 shows the standard function calls for finding the OpenCL platform and devices.

[21] (Kaeli, et al. 2015), page 63-66.

```
      // Get the OpenCL platform.
  platform = findPlatform("Altera");
  if(platform == NULL) {
    printf("ERROR: Unable to find Altera OpenCL platform.\n");
    return false;
  }

// Query the available OpenCL devices.
  scoped_array<cl_device_id> devices;
  cl_uint num_devices;

  devices.reset(getDevices(platform, CL_DEVICE_TYPE_ALL, &num_devices));
```

Figure 3.6 Find platform and devices.

2. Creating a context. After the platform and devices have been discovered, the host program creates a context that includes all devices. Figure 3.7 shows the standard function call for context creation along with error checking.

```
      // Create the context.
  context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
  checkError(status, "Failed to create context");
```

Figure 3.7 Create the context.

3. Creating a command-queue per device. Each device needs a command queue to handle most of the work for the device. The host submits commands to the command queue for execution on the device. Figure 3.8 shows the standard function call for command queue creation along with error checking.

```
// Create the command queue.
  queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE,
&status);
  checkError(status, "Failed to create command queue");
```

Figure 3.8 Create a command queue.

4. Creating memory objects (buffers) to hold data. Memory objects enable the transfer of data between the host and the device. The buffer is associated with the context on the host side, making it accessible to all devices in that context. Flags can also be used here to specify if data is read-only, write-only, or read-write. The hello world program has no need to hold data in memory, so Figure 3.9 is a code snippet from a vector addition program to show the creation of buffer objects.

```
// Input buffers.
    input_a_buf[i] = clCreateBuffer(context, CL_MEM_READ_ONLY,
        n_per_device[i] * sizeof(float), NULL, &status);
    checkError(status, "Failed to create buffer for input A");

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

    // Output buffer.
    output_buf[i] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
        n_per_device[i] * sizeof(float), NULL, &status);
    checkError(status, "Failed to create buffer for output");
```

Figure 3.9 Create host-side memory objects.

5. Copying the input data onto the device. Data is copied from a host pointer to a buffer which is ultimately transferred to the device when needed. Figure 3.10 shows the standard function call from the vector addition program.

```
status = clEnqueueWriteBuffer(queue[i], input_a_buf[i], CL_FALSE,
        0, n_per_device[i] * sizeof(float), input_a[i], 0, NULL,
&write_event[0]);
    checkError(status, "Failed to transfer input A");

    status = clEnqueueWriteBuffer(queue[i], input_b_buf[i], CL_FALSE,
        0, n_per_device[i] * sizeof(float), input_b[i], 0, NULL,
&write_event[1]);
    checkError(status, "Failed to transfer input B");
```

Figure 3.10 Transfer data to the device.

6. Creating and compiling a program from OpenCL C source code. The hello world kernel is stored in a character array named “helloWorld” which is used to create a program object that is compiled. During compilation, the information for each targeted device is provided as needed. Figure 3.11 show the create program and build program function calls.

```
// Create a program with source code
  cl_program program = clCreateProgramWithSource(context, 1, (const
char**)&helloWorld, NULL, &status);

  // Build (compile) the program for the device
  status = clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
```

Figure 3.11 Create and compile from source.

7. Extracting the kernel from the program. Since the hello world kernel is embedded in main.cpp as a character array “HelloWorld”, it must be extracted in order to be executed independently on a device. Figure 3.12 shows the standard function call for kernel extraction.

```
// Create the hello world kernel
  kernel = clCreateKernel(program, “helloWorld”, &status);
  checkError(status, "Failed to create kernel");
```

Figure 3.12 Kernel extraction.

8. Executing the kernel. Once the kernel has been created and data has been initialized, arguments can be set for the kernel. A command to execute the kernel can now be enqueued into the command-queue. Along with the kernel, the command requires specification of the ND-Range configuration, as well as the work-group sizes,

are set. Figure 3.13 depicts the standard function calls for setting kernel arguments and kernel execution.

```
// Set the kernel argument (argument 0)
  status = clSetKernelArg(kernel, 0, sizeof(cl_int),
(void*)&thread_id_to_output);
  checkError(status, "Failed to set kernel arg 0");

  printf("\nKernel initialization is complete.\n");
  printf("Launching the kernel...\n\n");

  // Configure work set over which the kernel will execute
  size_t wgSize[3] = {work_group_size, 1, 1};
  size_t gSize[3] = {work_group_size, 1, 1};

  // Launch the kernel
  status = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, gSize, wgSize, 0,
NULL, NULL);
         checkError(status, "Failed to launch kernel");unsigned argi = 0;
```

Figure 3.13 Kernel execution.

9. Copying output data back to the host. This step reads data back to a pointer on the host. Since the hello world program does not use buffer objects, Figure 3.14 is a code snippet from the vector addition program that copies output data back to the host.

```
// Read the result. This the final operation.
    status = clEnqueueReadBuffer(queue[i], output_buf[i], CL_FALSE,
        0, n_per_device[i] * sizeof(float), output[i], 1, &kernel_event[i],

&finish_event[i]);
```

Figure 3.14 Copy data back to host.

10. Releasing the OpenCL resources. The OpenCL resources that were allocated for kernel execution are released. This is similar to C or C++ programs where memory allocations (for example) are freed at the end of program execution. Figure 3.15 shows the standard function calls to release resources.

```
// Release local events.
    clReleaseEvent(write_event[0]);
    clReleaseEvent(write_event[1]);

// Release all events.
  for(unsigned i = 0; i < num_devices; ++i) {
    clReleaseEvent(kernel_event[i]);
    clReleaseEvent(finish_event[i]);
  }

clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseContext(context);
```

Figure 3.15 Release resources.

The hello world program is a good shell to start with when attempting to create OpenCL programs. The OpenCL reference guides from Kronos Group provide a comprehensive listing of all functions calls and their parameters for each version of OpenCL. At the writing of this research paper, the OpenCL API is on revision 2.2.

Converting the hello world program (or any program for that matter) to run on the DE1-SoC FPGA is as easy as creating a hardware configuration file (.aocx) and an executable file for the Linux-based environment used on the DE1-SoC board. As shown in Figure 3.1, the SOC EDS Cross-compiler is used to "make" a Linux executable file that runs the host code on the FPGA using the onboard processor. This executable file is built using the main.cpp file and includes linkages to any libraries and additional source code as needed. The hardware configuration file is built from the *kernel*.cl file using the Altera Offline Compiler (AOC) which communicates with the Quartus II software. This creates a VHDL-like version of the kernel that can be run directly on the FPGA. It is only necessary to have Quartus II installed for the AOC to use, no need to understand how to use this IDE or even understand VHDL programming. The SDK does all the heavy lifting of a hardware configuration for you.

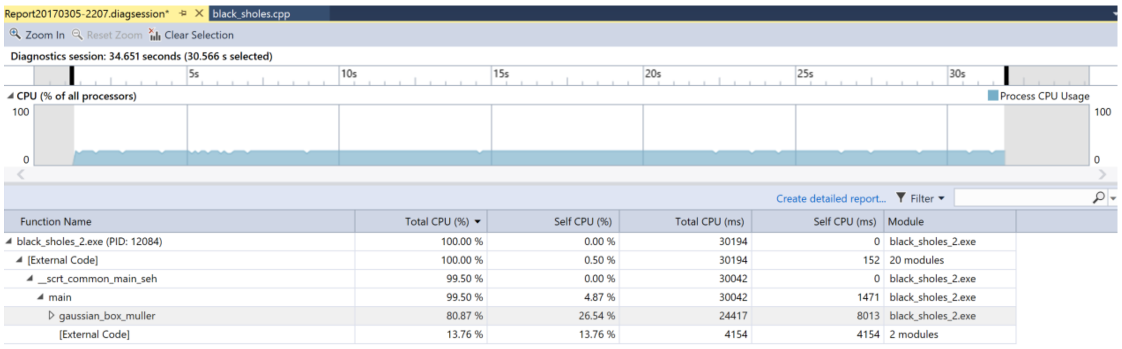
Maximizing Speed, Performance, and Portability

The advantages of using OpenCL can be best highlighted with example code that is straight forward and easy to manipulate in order to achieve easily recognizable results. It should be obvious to all software developers and hardware users that there is no one stop shop solution to solving computational problems. Some programs are designed to maximize the advantages offered by a specific hardware and vice versa. But OpenCL differs in that the program is created to be used across multiple hardware configurations. This opens up the possibility of the giving the OpenCL runtime the flexibility to choose the hardware device that is best suited for the task at hand. To emphasize the favorable attributes of OpenCL, I used a vector addition program that can be quickly altered to illustrate different results. The complete code listing for vector addition is listed in Appendix B.

Converting from C to OpenCL

To best demonstrate the effects of converting an existing program into OpenCL, I selected an existing code example that could be optimized to reap the benefits of running in an HCE using OpenCL. The financial market uses what is referred to as vanilla option pricing to determine call and put options for assets. The European vanilla option pricing method a uses random number generation to arrive at call and put options. A C-based example of this calculation provided the perfect test subject for analyzing existing code, determining how it could be optimized to run on existing hardware, and how the speed of program execution could be optimized. I utilized Michael Halls-Moore's version published on the QuantStart website which is a variation of the Black-Sholes Analytic Pricing Formula.[22] A complete listing of the original C-based program is provided in Appendix C, and I will refer to it as the black sholes program for reference. This C-based program uses the Box-Muller algorithm for determining random numbers used as input into a Monte Carlo function that calculates a call and put price options. Constant input values are entered for an option price, strike price, risk-free rate, volatility rate, and time.

The approach was to build and run the C version of this program, determine overall CPU execution time, and highlight which parts of the program are consuming the bulk of that time. Running performance profiler from Microsoft Visual Studio 2015 revealed that the gaussian_box_muller function was consuming 81% of the CPU's usage time as shown in Figure 3.16.



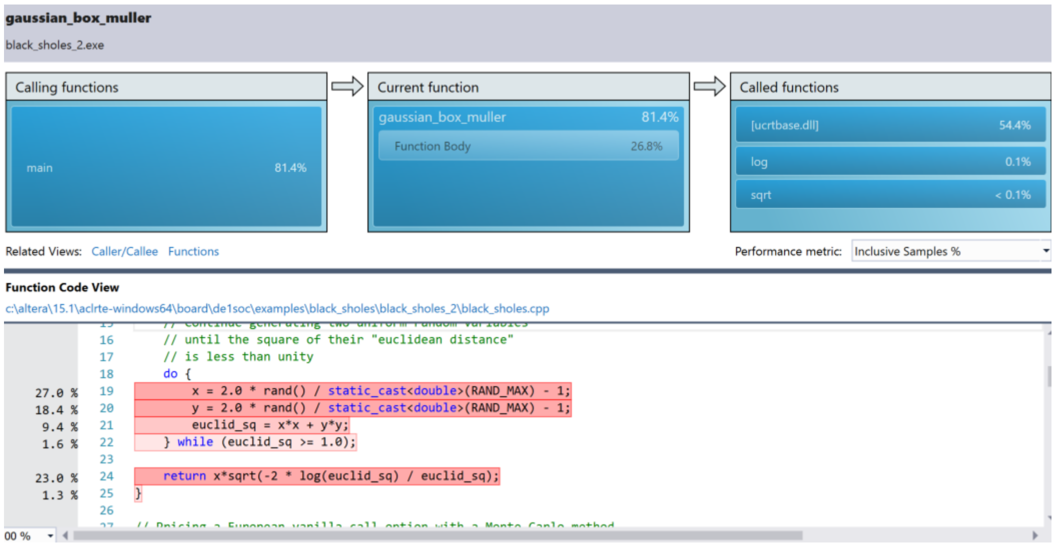
Diagnostics session: 34.651 seconds (30.566 s selected)

Function Name	Total CPU (%)	Self CPU (%)	Total CPU (ms)	Self CPU (ms)	Module
black_sholes_2.exe (PID: 12084)	100.00 %	0.00 %	30194	0	black_sholes_2.exe
[External Code]	100.00 %	0.50 %	30194	152	20 modules
__scrt_common_main_seh	99.50 %	0.00 %	30042	0	black_sholes_2.exe
main	99.50 %	4.87 %	30042	1471	black_sholes_2.exe
gaussian_box_muller	80.87 %	26.54 %	24417	8013	black_sholes_2.exe
[External Code]	13.76 %	13.76 %	4154	4154	2 modules

Figure 3.16 Gaussian Box-Muller Function.

[22] https://www.quantstart.com/articles/european-vanilla-option-pricing-with-c-via-monte-carlo-methods

As shown in Figure 3.17, the randomization of *x* and *y* within the *do…while* loop, which in itself only runs a few times to arrive at the final value for *x*, was consuming the bulk of effort from the CPU.


gaussian_box_muller
black_sholes_2.exe
Calling functions
main 81.4%
Current function
gaussian_box_muller 81.4%
Function Body 26.8%
Called functions
[ucrtbase.dll] 54.4%
log 0.1%
sqrt < 0.1%
Related Views: Caller/Callee Functions
Performance metric: Inclusive Samples %
Function Code View
c:\altera\15.1\aclrte-windows64\board\de1soc\examples\black_sholes\black_sholes_2\black_sholes.cpp

```
            // until the square of their "euclidean distance"
            // is less than unity
            do {
 27.0 %         x = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
 18.4 %         y = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
  9.4 %         euclid_sq = x*x + y*y;
  1.6 %     } while (euclid_sq >= 1.0);

 23.0 %     return x*sqrt(-2 * log(euclid_sq) / euclid_sq);
  1.3 % }
```

Figure 3.17 Kernel target

The challenge becomes how to convert this C-based program into OpenCL while optimizing program execution. The OpenCL approach is to look for functions or blocks of code as in Figure 3.17 that can be converted to a kernel for faster execution on a specific or multiple devices. This is akin to data and thread-level parallelization, but with the caveat of having more flexibility to tweak the number of work-items in a work-group and achieve even greater effects on the target hardware. The results and analysis gathered from this program conversion are discussed under the Black-Sholes sub-heading in chapter 4.

CHAPTER 4

FINDINGS

In order to quantify the results and analysis of using OpenCL in a HCE on multiple devices, this research focuses on three overarching use cases. The first use case focuses on a commonly used vector addition algorithm compare and contrast results across different hardware devices. The second use case involves converting an existing program from a tradition high level language such as C to OpenCL to demonstrate the benefits of such a conversion. The third use case focuses on running benchmarks that currently exist for GPU platform on the FPGA. The third use case also involves converting code, but demonstrates the portability of the OpenCL language. Common to all cases is the demonstration that programming hardware such as the FPGA has never been easier.

Vector Addition Program

The vector addition program takes two vectors and adds them together creating a third resultant vector. This particular OpenCL rendition of the program is a good example of how pipeline parallelism can be exploited on an FPGA to achieve results that are similar to a GPU. Figure 4.1 depicts a representation of the load and store operations that occur within each logic element of the FPGA. The DE1-SoC board has 84,000 logic elements that can be utilized simultaneously by segregating the input vectors into work groups. The FPGA handles each work group as if it was a thread during parallel execution. When loads of thread ID 0, for example, are passed to the

ALU for addition, the next two thread IDs are fetched from the host side memory buffer.

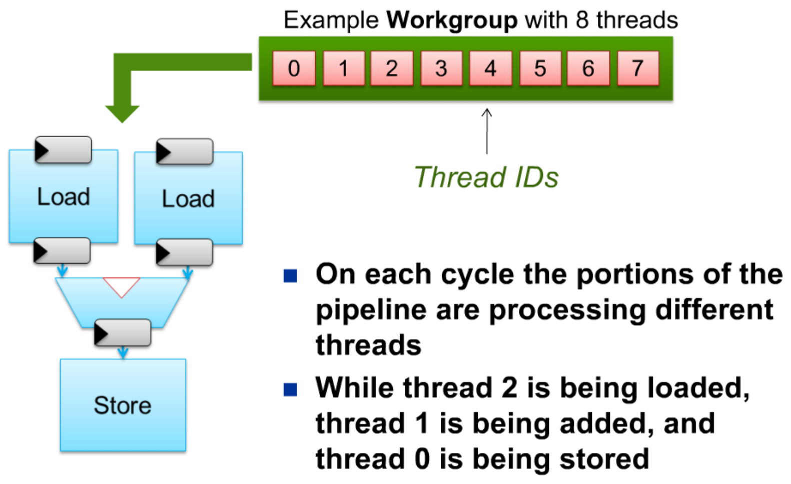

Example **Workgroup** with 8 threads
0
1
2
3
4
5
6
7
Thread IDs
Load
Load
Store
On each cycle the portions of the pipeline are processing different threads
While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Figure 4.1 Vector addition pipeline

The program is setup to run on all devices that are available so it is an excellent example of why multiple devices with an HCE are important and how OpenCL can best leverage those devices. It can be adjusted to run the entire kernel on one device only, mirrored vector addition on multiple devices simultaneously, or dividing the vector workload across the devices. Buffer objects are created for each device to facilitate the transfer of data between the host and device memory. For my setup, the DE1-SoC has its own memory and processor so running the main program and kernel directly on the board eliminates memory latency due to the spatial locality that would be present when running the source program from the computer.

Here is the method used to conduct a comparative analysis of the vector addition kernel run on 3 different devices:

<u>Devices:</u>

- CPU (4x compute units* at 1.7 GHZ) (Intel Core: 8 SP GFLOPS/cycle)
- GPU (20x compute units* at .2 GHZ)
- FPGA (1x compute unit* at .8 GHZ) (ARM Cortex A-9: 4 SP GFLOPS/cycle)

  * Depicts the number of compute units that the vector program detects per device.

<u>Program alteration</u>: altered the data bandwidth by changing the total elements to be processed:

- N = 500,000
- N = 1,000,000
- N = 10,000,000

Captured system clock time average over 5 runs for each value of N

<u>Results:</u>

- Results in Figure 4.2: straight system clock time for kernel execution
- Results 4-6: adjusted times to account for core advantage (FPGA baseline with 1 CU)
- Results 7-9: adjusted times for clock rate advantage (CPU baseline @ 1.7 GHZ)

Due to the construct of a one-dimensional vector array, there are limited options for manipulation of this program to achieve remarkable results in CPU type processing time. I chose to manipulate the data bandwidth of program execution by changing the number of elements in the array and produced results shown in Figure 4.2. The vector addition program captures performance internally by saving system clock time at the beginning and end of kernel execution. Changing the data bandwidth by increasing the number of elements in the vectors increased the overall execution time, which is to be expected and is shown in Figure 4.2.

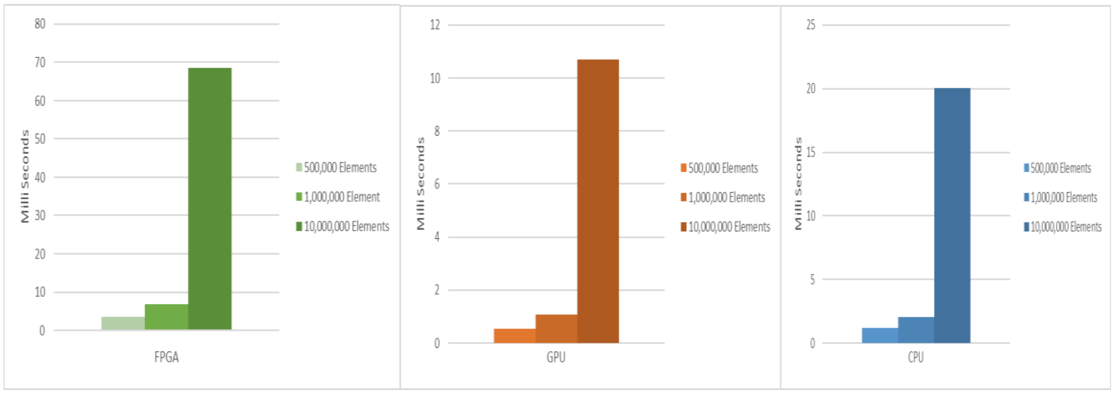


Figure 4.2 Change in data bandwidth

To realize the true measure of hardware performance between devices, however, adjustments need to be made to the results to compensate for advantages that individual devices have over other devices. As seen on the previous page in the device specifications, there is a considerable difference in the processor clock rate and a number of compute units for each device recognized by the program. The CPU has the advantage of clock rate at 1.7 GHz, followed by the FPGA at .8 GHz, and the GPU at .2 GHz. Also, OpenCL recognizes the GPU as having the most compute units at 20, followed by the CPU with 4, followed by the FPGA with 1.

To calculate CPU performance relative to each devices advantage, I used the standard calculation for determining relative performance. The formula for calculating relative performance is as simple as dividing device A by device B to arrive at a ratio:[23]

$$\frac{\text{CPU Performance}_A}{\text{CPU Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A}$$

Applying the same theory of relative performance to the vector addition results, I calculated the following adjustments to the results from Figure 4.2:

	# of Compute Units	Processor Speed Up
CPU @ 1.7 GHz on 4 CU	Execution time x 4	Execution time / 1
GPU @ .2 GHz on 20 CU	Execution time x 20	Execution time / 8.5
FPGA @ .8 GHz on 1 CU	Execution time x 1	Execution time / 2.125

Table 4.1 Relative Performance Adjustments

With a level playing field for each processor (excluding any data conflicts caused by the shared memory between the CPU and GPU), the adjustments listed in Table 4.1 yield the results in Figure 4.3. This is a prime example of how the advantage of an FPGA versus CPU or GPU can be overlooked. Given the same clock rate and equal program workload distribution, the FPGA outperforms the CPU, is very close to the GPU and consumes multitudes less power than both.

[23] Patterson, David, and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* New York: Morgan Kaufmann, 2009, 34.

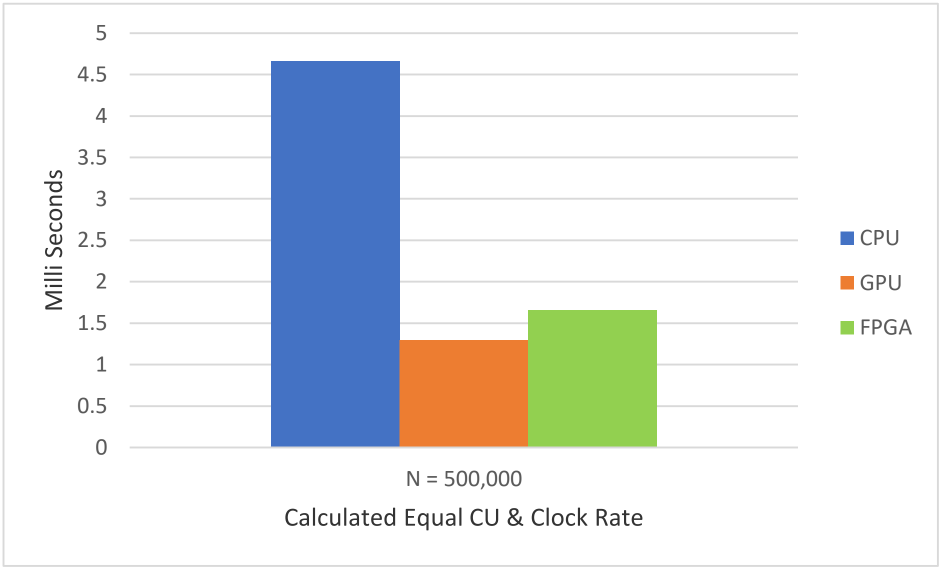


Figure 4.3 Relative performance.

Black Sholes Program

The black sholes program served as an entry level program that offered all the challenges of converting a standard C-based program into OpenCL along with all the benefits of doing so. Achieving program optimization without just creating redundant code is the goal of any developer. The best method is to work from the inside out, looking for functions and blocks of code that could benefit from parallelization. As mentioned earlier, the gaussian_box_muller function shown in Figure 4.4 consumes 81% of the CPU's resources during program execution.

```
double gaussian_box_muller() {
	double x = 0.0;
	double y = 0.0;
	double euclid_sq = 0.0;

	// Continue generating two uniform random variables
	// until the square of their "euclidean distance"
	// is less than unity
	do {
		x = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
		y = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
		euclid_sq = x*x + y*y;
	} while (euclid_sq >= 1.0);
	return x*sqrt(-2 * log(euclid_sq) / euclid_sq);
}
```

Figure 4.4 Gaussian Box Muller Function in C.

If the number of sims for the black sholes program is set at 100,000 for example, the function is called 100,000 times by both the put and call functions respectively (see Figure 4.5), with each call using multiple executions of a do-while loop to output a random number.

```
do {
		x = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
		y = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
		euclid_sq = x*x + y*y;
	} while (euclid_sq >= 1.0);
	return x*sqrt(-2 * log(euclid_sq) / euclid_sq);
}

// Pricing a European vanilla call option with a Monte Carlo method
double monte_carlo_call_price(const int& num_sims, const double& S, const
double& K, const double& r, const double& v, const double& T) {
	double S_adjust = S * exp(T*(r - 0.5*v*v));
	double S_cur = 0.0;
	double payoff_sum = 0.0;

	for (int i = 0; i<num_sims; i++) {
		double gauss_bm = gaussian_box_muller();
		S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
		payoff_sum += std::max(S_cur - K, 0.0);
	}

	return (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
	}
```

Figure 4.5 Original call function

Since OpenCL does not support the rand() function, I wasn't able to run the entire gaussian_box_muller function as a kernel. I used local host memory to create the random values of both *x* and *y*, copying them into arrays A and B as shown in Figure 4.6.

```
double *A = (double*)malloc(datasize);
double *B = (double*)malloc(datasize);
double *C = (double*)malloc(datasize);

//Loads the random x & y values into arrays A and B in host memory
int i;
for (i = 0; i < elements; i++) {
       A[i] = (2.0 * rand() / static_cast<double>(RAND_MAX) - 1);
       B[i] = (2.0 * rand() / static_cast<double>(RAND_MAX) - 1);
                                     }
```

Figure 4.6 Host arrays.

Having established that the performance bottleneck was related to the Gaussian_box_muller function and how the put and call functions interact with it, I focused on utilizing a kernel to speed up the process of generating random numbers and utilizing them in the put and call functions being handled by the host. Using input and output buffers, I transferred the random number arrays A and B into the kernel, continued the rest of the Gaussian_box_muller calculations within the kernel on the targeted device, loaded the results into the output buffer C, and transferred that data back into the put and call functions. The results shown in Figure 4.7 are indicative of the affects that vectorization of data transfers and processing can have on program performance. The original C version of the program was compared to the OpenCL version as run on the CPU, and both of those results were compared to FPGA. The performance of the program on CPU improved due to a decrease in program instructions and function calls. By selecting the appropriate portion of the code to be farmed out to the FPGA through a kernel, I achieved the best overall run time.

Effective resource and memory management enables the best balance between CPU and FPGA for overall program execution.

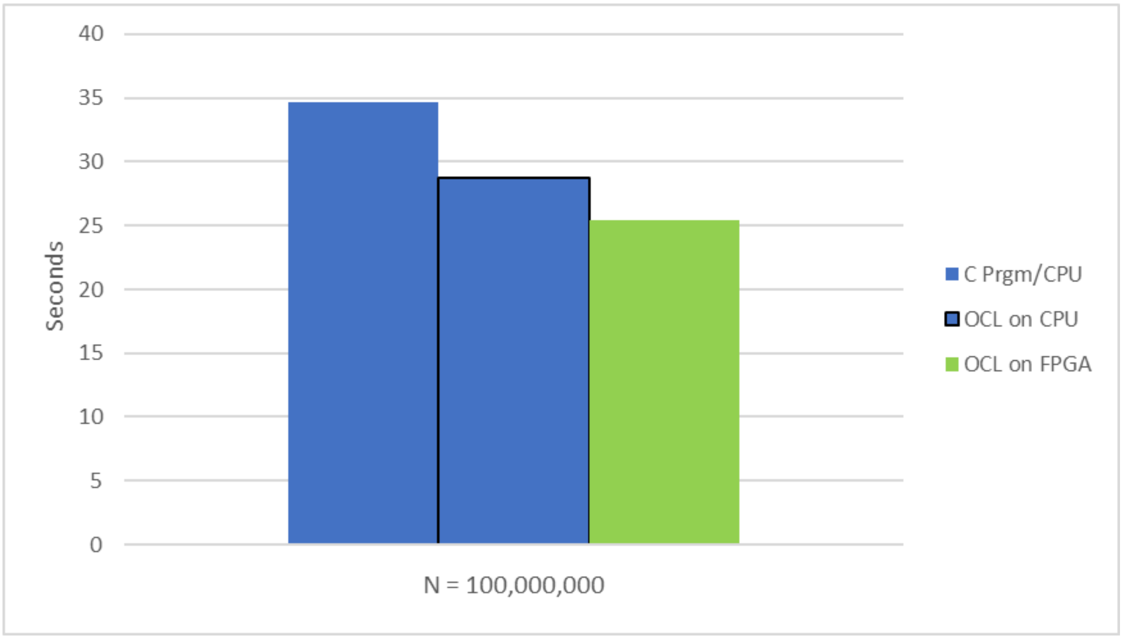


Figure 4.7 Black Sholes Results

Rodinia Benchmarks

The Rodinia Benchmark Suite, version 3.1 offers a wide range of benchmarks that are targeted for CUDA, OMP, and OCL applications on GPU hardware. Since no OpenCL benchmarks exist for FPGAs, the Rodinia benchmarks were chosen since they contained OpenCL coding and would serve as an opportunity to maximize on the benefits of running OpenCL in a HCE. Since the benchmarks were designed for optimal performance on GPU, not all of them are good candidates for conversion and execution on FPGA. The benchmarks chosen for research were algorithms that could benefit from vectorization, were heavy in data transfer between local, global, and host memory, and utilized multiple kernels.

The first benchmark chosen was the K-Means data-mining algorithm which is heavy in data parallelization. The algorithm is designed to take an initial set of input data and sort it into clusters based upon the data's unique features.   Within each of these clusters, one item is determined to be the centroid for that particular cluster. The program uses the Euclidean Distance metric to calculate distances between data elements. The Rodinia version of the K-Means program maintains all the data points, features, cluster centers, and data/cluster integrity stored in separate arrays. Pointers are used to reference all of these arrays while performing the I/O, clustering, centroid computation, and cluster reassigning based upon proximity of the data elements to the centroids. The program is divided into separate .c files with the main execution residing within the kmeans.cpp file.

As mentioned earlier, the Rodinia Benchmarks are optimized to run on GPU. The K-Means algorithm provides for multiple points of manipulation to achieve different results which benefit experiments with both software and hardware. Running the K-Means benchmark as downloaded showed results that favored the GPU over the FPGA. The OpenCL kernels with this benchmark are quite simple and don't provide an opportunity for FPGA optimization. By thorough examination of the code, I decided to manipulate the constant number of clusters while adjusting the work-group size to best match the FPGA's capabilities. The best combination of cluster groups to work-group size produced the worst performance for the GPU. Figure 4.8 shows the optimal results obtained when working on a 100,000-element data set with 10 clusters and a work-group size of 1024.

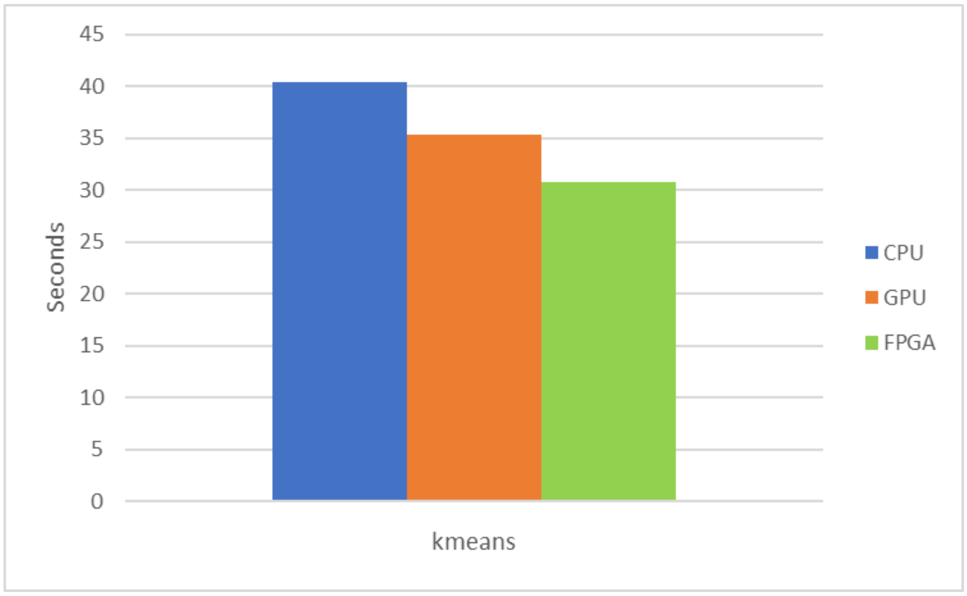


Figure 4.8 K-Means FPGA optimization.

The next Rodinia Benchmark selected for analysis was the hybrid sort program which is a combination of two popular sorting algorithms, bucket sort and merge sort. The program is designed to take list of floats in random order and run a bucket sort algorithm to separate the input data into groups, or, buckets. This step of the sorting process can be configured to run across all devices in the HCE, tracking the CPU execution time of each device. Next, the buckets are stored in a vector that is fed into the merge sort portion of the program. The merge sort can also be configured to run on all devices and the CPU execution time is tracked. As is a common theme with OpenCL optimization, examining the code execution and determining the proper work-group size to match the targeted hardware device can often be all that is required to achieved best results. Figure 4.9 shows the result of the merge sort and how the

DE1-SoC board achieved a slight advantage over GPU when the work-group size was set at 1024.

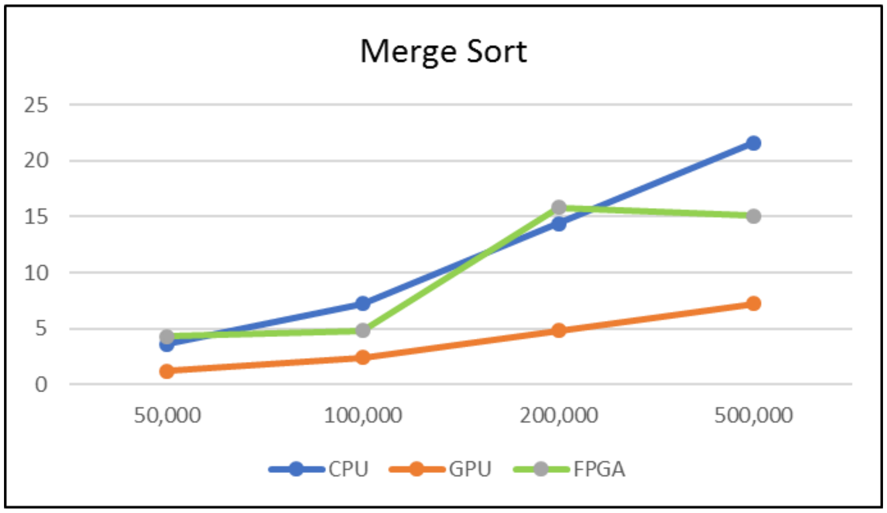


Figure 4.9 Merge sort.

This result demonstrates how vectors run well on FPGAs and can maximize the effect of their inherent pipeline parallelism. Figure 4.10 shows a more typical result for an FPGA board when running kernels that are not vectorized.

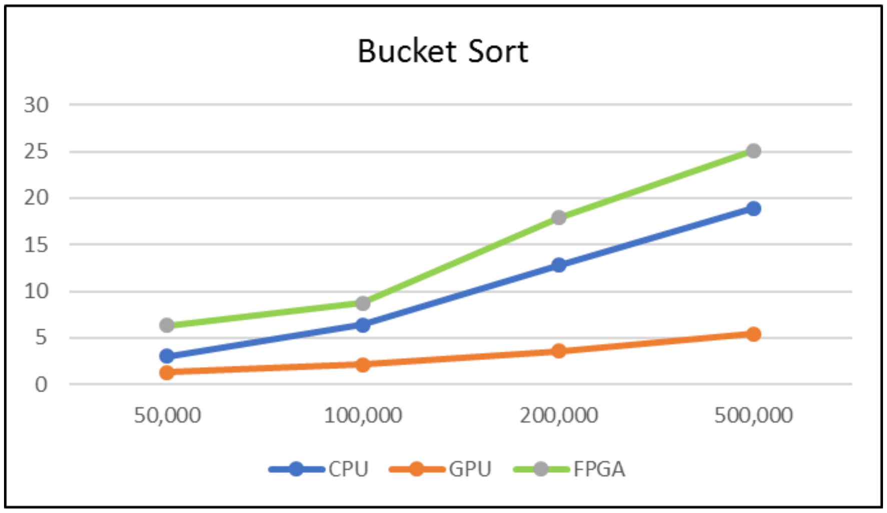


Figure 4.10 Bucket sort.

Also, the hierarchy of program execution between the ND-Range, work-group, and work-items underlines one of the major benefits of OpenCL in its portability and scalability.

CHAPTER 5

CONCLUSION

This research was dedicated to determining if program acceleration with OpenCL in a Heterogeneous Computing Environment consisting of FPGA, CPU, and GPU is a credible approach to increasing speed, performance, and ultimately minimizing power consumption. Today’s computing industry has a higher demand than ever before on maximal computation performance with minimal power consumption. We have moved well beyond the era of processor overclocking and moved on to multi-tentacle approaches to enhancing CPU performance such as multi-cores, distributive computing, and now OpenCL. Being able to maximize the performance of all hardware platforms simultaneously, independently, or sequentially is what OpenCL is all about.

Learning a new computer language can always be a challenge. Time and repetition are paramount as well as tapping into the best resources available. Compared to other high-level languages such as Java, OpenCL may be somewhat intimidating for developers without C or C++ experience. I found the language to be straightforward to learn the basics, and there are multiple online training sessions offered at no cost from Intel which will get even the novice programmer up and running in relatively short order. The more detailed and in-depth programming lessons will cost money, however. Since the Kronos Group is the manager of the official OpenCL API, their website is naturally an excellent repository of source code and libraries. GitHub can also be used but be cautious not to mix up header files for different versions of OpenCL.

The software development kit used was provided by Altera Corporation which required me to obtain a student license. Altera has since been acquired by Intel, so all software downloads can be obtained through their website as well. Intel will direct you to use Microsoft Visual Studio 2010 for compiling your host code for use with their SDK. Learning the API of the SDK also takes some time, but the instructions are very well written and come with simple code examples to help understand the interoperability between all devices in your environment.

The results gathered from the vector addition and black sholes programs definitely underline how quickly one can begin to code in OpenCL and obtain immediate performance improvements that are manageable and scalable. Experimenting with work item quantities and work-group sizes gives you the power to parallelize your applications ever further than normal thread and data level parallelization techniques. Utilizing the multiple NDRanges (3 in total) offered by the OpenCL platform gives you additional scalability options that increase the need for synchronization between work-items, groups, multiple kernels (if used), and host and device-side memory. All of which is controllable by either host or device.

Programming hardware has never been easier. OpenCL eliminates the need to learn complex hardware programming languages such as Verilog or VHDL to program FPGA. Kernels can be built to run on multiple devices or be tailored to maximize the efficiency of a specific hardware platform.

Having device-side and host-side memory models that are scalable gives additional programming flexibility for larger and more data-intensive applications. Shared memory was introduced with the release of OpenCL version 2.0 which provides and advantage similar to having multiple devices on the same motherboard sharing the same address space. Partitioning and programming specific device to host memory transfers generated significant performance improvements with the black sholes program.

Future research projects could include using OpenGL which focuses primarily on GPU or, WebCL which is used to optimize the performance of web applications such as web browsers and cloud servers. There are additional vendors that offer SDKs for OpenCL as well, and true OpenCL benchmarks for FPGA should be plentiful within the next few years. The possibilities for use of OpenCL are endless. Wherever there is code to be optimized, OpenCL should be considered. It may not provide the perfect solution for every application, but can reap immediate results and is most definitely a vehicle for future progress.

APPENDIX A

Hello World OpenCL Code

```
/*Copyright (C) 2013-2014 Altera Corporation, San Jose, California, USA. All
rights reserved. Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including without
limitation the rights to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell copies of the Software, and to permit persons to whom
the Software is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cstring>
#include "CL/opencl.h"
#include "AOCL_Utils.h"

using namespace aocl_utils;

#define STRING_BUFFER_LEN 1024

// Runtime constants
// Used to define the work set over which this kernel will execute.
static const size_t work_group_size = 8;  // 8 threads in the demo workgroup
// Defines kernel argument value, which is the workitem ID that will
// execute a printf call
static const int thread_id_to_output = 2;

// OpenCL runtime configuration
static cl_platform_id platform = NULL;
static cl_device_id device = NULL;
static cl_context context = NULL;
static cl_command_queue queue = NULL;
static cl_kernel kernel = NULL;
static cl_program program = NULL;

// Function prototypes
bool init();
void cleanup();
static void device_info_ulong( cl_device_id device, cl_device_info param, const
char* name);
static void device_info_uint( cl_device_id device, cl_device_info param, const
char* name);
static void device_info_bool( cl_device_id device, cl_device_info param, const
char* name);
static void device_info_string( cl_device_id device, cl_device_info param,
const char* name);
static void display_device_info( cl_device_id device );

// Entry point.
int main() {
  cl_int status;
  if(!init()) {
    return -1;
  }

  // Set the kernel argument (argument 0)
  status = clSetKernelArg(kernel, 0, sizeof(cl_int),
(void*)&thread_id_to_output);
  checkError(status, "Failed to set kernel arg 0");

  printf("\nKernel initialization is complete.\n");
  printf("Launching the kernel...\n\n");

  // Configure work set over which the kernel will execute
  size_t wgSize[3] = {work_group_size, 1, 1};
  size_t gSize[3] = {work_group_size, 1, 1};

  // Launch the kernel
  status = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, gSize, wgSize, 0,
NULL, NULL);
  checkError(status, "Failed to launch kernel");

  // Wait for command queue to complete pending events
  status = clFinish(queue);
  checkError(status, "Failed to finish");

  printf("\nKernel execution is complete.\n");

  // Free the resources allocated
  cleanup();

  return 0;
}

/////// HELPER FUNCTIONS ///////

bool init() {
  cl_int status;

  if(!setCwdToExeDir()) {
    return false;
  }

  // Get the OpenCL platform.
  platform = findPlatform("Altera");
  if(platform == NULL) {
    printf("ERROR: Unable to find Altera OpenCL platform.\n");
    return false;
  }

  // User-visible output - Platform information
  {
    char char_buffer[STRING_BUFFER_LEN];
    printf("Querying platform for info:\n");
    printf("==========================\n");
    clGetPlatformInfo(platform, CL_PLATFORM_NAME, STRING_BUFFER_LEN,
char_buffer, NULL);
    printf("%-40s = %s\n", "CL_PLATFORM_NAME", char_buffer);
    clGetPlatformInfo(platform, CL_PLATFORM_VENDOR, STRING_BUFFER_LEN,
char_buffer, NULL);
    printf("%-40s = %s\n", "CL_PLATFORM_VENDOR ", char_buffer);
    clGetPlatformInfo(platform, CL_PLATFORM_VERSION, STRING_BUFFER_LEN,
char_buffer, NULL);
    printf("%-40s = %s\n\n", "CL_PLATFORM_VERSION ", char_buffer);
  }

  // Query the available OpenCL devices.
  scoped_array<cl_device_id> devices;
  cl_uint num_devices;

  devices.reset(getDevices(platform, CL_DEVICE_TYPE_ALL, &num_devices));

  // We'll just use the first device.
  device = devices[0];

  // Display some device information.
  display_device_info(device);

  // Create the context.
  context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
  checkError(status, "Failed to create context");

  // Create the command queue.
  queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE,
&status);
  checkError(status, "Failed to create command queue");

  // Create the program.
  std::string binary_file = getBoardBinaryFile("hello_world", device);
  printf("Using AOCX: %s\n", binary_file.c_str());
  program = createProgramFromBinary(context, binary_file.c_str(), &device, 1);

  // Build the program that was just created.
  status = clBuildProgram(program, 0, NULL, "", NULL, NULL);
  checkError(status, "Failed to build program");

  // Create the kernel - name passed in here must match kernel name in the
  // original CL file, that was compiled into an AOCX file using the AOC tool
  const char *kernel_name = "hello_world";  // Kernel name, as defined in the
CL file
  kernel = clCreateKernel(program, kernel_name, &status);
  checkError(status, "Failed to create kernel");

  return true;
}

// Free the resources allocated during initialization
void cleanup() {
  if(kernel) {
    clReleaseKernel(kernel);
  }
  if(program) {
    clReleaseProgram(program);
  }
  if(queue) {
    clReleaseCommandQueue(queue);
  }
  if(context) {
    clReleaseContext(context);
  }
}

// Helper functions to display parameters returned by OpenCL queries
static void device_info_ulong( cl_device_id device, cl_device_info param, const
char* name) {
   cl_ulong a;
   clGetDeviceInfo(device, param, sizeof(cl_ulong), &a, NULL);
   printf("%-40s = %lu\n", name, a);
}
static void device_info_uint( cl_device_id device, cl_device_info param, const
char* name) {
   cl_uint a;
   clGetDeviceInfo(device, param, sizeof(cl_uint), &a, NULL);
   printf("%-40s = %u\n", name, a);
}
static void device_info_bool( cl_device_id device, cl_device_info param, const
char* name) {
   cl_bool a;
   clGetDeviceInfo(device, param, sizeof(cl_bool), &a, NULL);
   printf("%-40s = %s\n", name, (a?"true":"false"));
}
static void device_info_string( cl_device_id device, cl_device_info param,
const char* name) {
   char a[STRING_BUFFER_LEN];
   clGetDeviceInfo(device, param, STRING_BUFFER_LEN, &a, NULL);
   printf("%-40s = %s\n", name, a);
}

// Query and display OpenCL information on device and runtime environment
static void display_device_info( cl_device_id device ) {

   printf("Querying device for info:\n");
   printf("========================\n");
   device_info_string(device, CL_DEVICE_NAME, "CL_DEVICE_NAME");
   device_info_string(device, CL_DEVICE_VENDOR, "CL_DEVICE_VENDOR");
   device_info_uint(device, CL_DEVICE_VENDOR_ID, "CL_DEVICE_VENDOR_ID");
   device_info_string(device, CL_DEVICE_VERSION, "CL_DEVICE_VERSION");
   device_info_string(device, CL_DRIVER_VERSION, "CL_DRIVER_VERSION");
   device_info_uint(device, CL_DEVICE_ADDRESS_BITS, "CL_DEVICE_ADDRESS_BITS");
   device_info_bool(device, CL_DEVICE_AVAILABLE, "CL_DEVICE_AVAILABLE");
   device_info_bool(device, CL_DEVICE_ENDIAN_LITTLE,
"CL_DEVICE_ENDIAN_LITTLE");
   device_info_ulong(device, CL_DEVICE_GLOBAL_MEM_CACHE_SIZE,
"CL_DEVICE_GLOBAL_MEM_CACHE_SIZE");
   device_info_ulong(device, CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE,
"CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE");
   device_info_ulong(device, CL_DEVICE_GLOBAL_MEM_SIZE,
"CL_DEVICE_GLOBAL_MEM_SIZE");
   device_info_bool(device, CL_DEVICE_IMAGE_SUPPORT,
"CL_DEVICE_IMAGE_SUPPORT");
   device_info_ulong(device, CL_DEVICE_LOCAL_MEM_SIZE,
"CL_DEVICE_LOCAL_MEM_SIZE");
   device_info_ulong(device, CL_DEVICE_MAX_CLOCK_FREQUENCY,
"CL_DEVICE_MAX_CLOCK_FREQUENCY");
```

```
   device_info_ulong(device, CL_DEVICE_MAX_COMPUTE_UNITS,
"CL_DEVICE_MAX_COMPUTE_UNITS");
   device_info_ulong(device, CL_DEVICE_MAX_CONSTANT_ARGS,
"CL_DEVICE_MAX_CONSTANT_ARGS");
   device_info_ulong(device, CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE,
"CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE");
   device_info_uint(device, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
"CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS");
   device_info_uint(device, CL_DEVICE_MEM_BASE_ADDR_ALIGN,
"CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS");
   device_info_uint(device, CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE,
"CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE");
   device_info_uint(device, CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR,
"CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR");
   device_info_uint(device, CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT,
"CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT");
   device_info_uint(device, CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT,
"CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT");
   device_info_uint(device, CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG,
"CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG");
   device_info_uint(device, CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT,
"CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT");
   device_info_uint(device, CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE,
"CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE");

   {
      cl_command_queue_properties ccp;
      clGetDeviceInfo(device, CL_DEVICE_QUEUE_PROPERTIES,
sizeof(cl_command_queue_properties), &ccp, NULL);
      printf("%-40s = %s\n", "Command queue out of order? ", ((ccp &
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE)?"true":"false"));
      printf("%-40s = %s\n", "Command queue profiling enabled? ", ((ccp &
CL_QUEUE_PROFILING_ENABLE)?"true":"false"));
   }
}
```

APPENDIX B

Vector Addition OpenCL Code

```
/* Copyright (C) 2013-2014 Altera Corporation, San Jose, California, USA. All
rights reserved. Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including without
limitation the rights to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell copies of the Software, and to permit persons to whom
the Software is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software. */

/* This host program executes a vector addition kernel to perform:  C = A + B
where A, B and C are vectors with N elements.  This host program supports
partitioning the problem across multiple OpenCL devices if available. If there
are M available devices, the problem is divided so that each device operates on
N/M points. The host program assumes that all devices are of the same type
(that is, the same binary can be used), but the code can be generalized to
support different device types easily. Verification is performed against the
same computation on the host CPU. */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "opencl.h"
//#include "c:\hld\host\include\CL\opencl.h"
#include
"c:\altera\15v1\hld\board\terasic\de1soc\examples\common\inc\AOCL_Utils.h"

using namespace aocl_utils;
//using namespace AOCL_Utils;

// OpenCL runtime configuration
cl_platform_id platform = NULL;
unsigned num_devices = 0;
scoped_array<cl_device_id> device; // num_devices elements
cl_context context = NULL;
scoped_array<cl_command_queue> queue; // num_devices elements
cl_program program = NULL;
scoped_array<cl_kernel> kernel; // num_devices elements
scoped_array<cl_mem> input_a_buf; // num_devices elements
scoped_array<cl_mem> input_b_buf; // num_devices elements
scoped_array<cl_mem> output_buf; // num_devices elements

// Problem data.
const unsigned N = 1000000; // problem size
scoped_array<scoped_aligned_ptr<float> > input_a, input_b; // num_devices
elements
scoped_array<scoped_aligned_ptr<float> > output; // num_devices elements
scoped_array<scoped_array<float> > ref_output; // num_devices elements
scoped_array<unsigned> n_per_device; // num_devices elements

// Function prototypes
float rand_float();
bool init_opencl();
```

```
void init_problem();
void run();
void cleanup();

// Entry point.
int main() {
  // Initialize OpenCL.
  if(!init_opencl()) {
    return -1;
  }

  // Initialize the problem data.
  // Requires the number of devices to be known.
  init_problem();

  // Run the kernel.
  run();

  // Free the resources allocated
  cleanup();

  return 0;
}

/////// HELPER FUNCTIONS ///////

// Randomly generate a floating-point number between -10 and 10.
float rand_float() {
  return float(rand()) / float(RAND_MAX) * 20.0f - 10.0f;
}

// Initializes the OpenCL objects.
bool init_opencl() {
  cl_int status;

  printf("Initializing OpenCL\n");

  if(!setCwdToExeDir()) {
    return false;
  }

  // Get the OpenCL platform.
  platform = findPlatform("Intel(R) OpenCL");
  if(platform == NULL) {
    printf("ERROR: Unable to find Altera OpenCL platform.\n");
    return false;
  }

  // Query the available OpenCL device.
  device.reset(getDevices(platform, CL_DEVICE_TYPE_ALL, &num_devices));
  printf("Platform: %s\n", getPlatformName(platform).c_str());
  printf("Using %d device(s)\n", num_devices);
  for(unsigned i = 0; i < num_devices; ++i) {
    printf("  %s\n", getDeviceName(device[i]).c_str());
  }

  // Create the context.
  context = clCreateContext(NULL, num_devices, device, NULL, NULL, &status);
  checkError(status, "Failed to create context");

  // Create the program for all device. Use the first device as the
  // representative device (assuming all device are of the same type).

  std::string binary_file = getBoardBinaryFile("vectorAdd", device[0]);
  printf("Using AOCX: %s\n", binary_file.c_str());
  program = createProgramFromBinary(context, binary_file.c_str(), device,
num_devices);

  // Build the program that was just created.
  status = clBuildProgram(program, 0, NULL, "", NULL, NULL);
  checkError(status, "Failed to build program");

  // Create per-device objects.
  queue.reset(num_devices);
  kernel.reset(num_devices);
  n_per_device.reset(num_devices);
  input_a_buf.reset(num_devices);
  input_b_buf.reset(num_devices);
  output_buf.reset(num_devices);

  for(unsigned i = 0; i < num_devices; ++i) {
    // Command queue.
    queue[i] = clCreateCommandQueue(context, device[i],
CL_QUEUE_PROFILING_ENABLE, &status);
    checkError(status, "Failed to create command queue");

    // Kernel.
    const char *kernel_name = "vectorAdd";
    kernel[i] = clCreateKernel(program, kernel_name, &status);
    checkError(status, "Failed to create kernel");

    // Determine the number of elements processed by this device.
    n_per_device[i] = N / num_devices; // number of elements handled by this
device

    // Spread out the remainder of the elements over the first
    // N % num_devices.
    if(i < (N % num_devices)) {
      n_per_device[i]++;
    }

    // Input buffers.
    input_a_buf[i] = clCreateBuffer(context, CL_MEM_READ_ONLY,
        n_per_device[i] * sizeof(float), NULL, &status);
    checkError(status, "Failed to create buffer for input A");

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

    // Output buffer.
    output_buf[i] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
        n_per_device[i] * sizeof(float), NULL, &status);
    checkError(status, "Failed to create buffer for output");
  }
  return true;
}

// Initialize the data for the problem. Requires num_devices to be known.
void init_problem() {
  if(num_devices == 0) {
    checkError(-1, "No devices");
  }

  input_a.reset(num_devices);
  input_b.reset(num_devices);
  output.reset(num_devices);
  ref_output.reset(num_devices);

  // Generate input vectors A and B and the reference output consisting
  // of a total of N elements.
  // We create separate arrays for each device so that each device has an
  // aligned buffer.
  for(unsigned i = 0; i < num_devices; ++i) {
    input_a[i].reset(n_per_device[i]);
    input_b[i].reset(n_per_device[i]);
    output[i].reset(n_per_device[i]);
    ref_output[i].reset(n_per_device[i]);

    for(unsigned j = 0; j < n_per_device[i]; ++j) {
      input_a[i][j] = rand_float();
      input_b[i][j] = rand_float();
      ref_output[i][j] = input_a[i][j] + input_b[i][j];
    }
  }
}

void run() {
  cl_int status;

  const double start_time = getCurrentTimestamp();

  // Launch the problem for each device.
  scoped_array<cl_event> kernel_event(num_devices);
  scoped_array<cl_event> finish_event(num_devices);

  for(unsigned i = 0; i < num_devices; ++i) {

    // Transfer inputs to each device. Each of the host buffers supplied to
    // clEnqueueWriteBuffer here is already aligned to ensure that DMA is used
    // for the host-to-device transfer.
    cl_event write_event[2];
    status = clEnqueueWriteBuffer(queue[i], input_a_buf[i], CL_FALSE,
        0, n_per_device[i] * sizeof(float), input_a[i], 0, NULL,
&write_event[0]);
    checkError(status, "Failed to transfer input A");

    status = clEnqueueWriteBuffer(queue[i], input_b_buf[i], CL_FALSE,
        0, n_per_device[i] * sizeof(float), input_b[i], 0, NULL,
&write_event[1]);
    checkError(status, "Failed to transfer input B");

    // Set kernel arguments.
    unsigned argi = 0;

    status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem),
&input_a_buf[i]);
    checkError(status, "Failed to set argument %d", argi - 1);

    status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem),
&input_b_buf[i]);
    checkError(status, "Failed to set argument %d", argi - 1);

    status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &output_buf[i]);
    checkError(status, "Failed to set argument %d", argi - 1);

    // Enqueue kernel.
    // Use a global work size corresponding to the number of elements to add
    // for this device.
    //
    // We don't specify a local work size and let the runtime choose
    // (it'll choose to use one work-group with the same size as the global
    // work-size).
    //
    // Events are used to ensure that the kernel is not launched until
    // the writes to the input buffers have completed.
    const size_t global_work_size = n_per_device[i];
    printf("Launching for device %d (%zd elements)\n", i, global_work_size);

    status = clEnqueueNDRangeKernel(queue[i], kernel[i], 1, NULL,
        &global_work_size, NULL, 2, write_event, &kernel_event[i]);
    checkError(status, "Failed to launch kernel");

    // Read the result. This the final operation.
    status = clEnqueueReadBuffer(queue[i], output_buf[i], CL_FALSE,
        0, n_per_device[i] * sizeof(float), output[i], 1, &kernel_event[i],
&finish_event[i]);

    // Release local events.
    clReleaseEvent(write_event[0]);
    clReleaseEvent(write_event[1]);
  }

  // Wait for all devices to finish.
  clWaitForEvents(num_devices, finish_event);

  const double end_time = getCurrentTimestamp();

  // Wall-clock time taken.
  printf("\nTime: %0.3f ms\n", (end_time - start_time) * 1e3);

  // Get kernel times using the OpenCL event profiling API.
  for(unsigned i = 0; i < num_devices; ++i) {
    cl_ulong time_ns = getStartEndTime(kernel_event[i]);
    printf("Kernel time (device %d): %0.3f ms\n", i, double(time_ns) * 1e-6);
  }

  // Release all events.
  for(unsigned i = 0; i < num_devices; ++i) {
    clReleaseEvent(kernel_event[i]);
    clReleaseEvent(finish_event[i]);
  }

  // Verify results.
  bool pass = true;
  for(unsigned i = 0; i < num_devices && pass; ++i) {
    for(unsigned j = 0; j < n_per_device[i] && pass; ++j) {
      if(fabsf(output[i][j] - ref_output[i][j]) > 1.0e-5f) {
        printf("Failed verification @ device %d, index %d\nOutput: 
%f\nReference: %f\n",
            i, j, output[i][j], ref_output[i][j]);
        pass = false;
      }
    }
  }

  printf("\nVerification: %s\n", pass ? "PASS" : "FAIL");
}

// Free the resources allocated during initialization
void cleanup() {
  for(unsigned i = 0; i < num_devices; ++i) {
    if(kernel && kernel[i]) {
      clReleaseKernel(kernel[i]);
    }
    if(queue && queue[i]) {
      clReleaseCommandQueue(queue[i]);
    }
    if(input_a_buf && input_a_buf[i]) {
      clReleaseMemObject(input_a_buf[i]);
    }
    if(input_b_buf && input_b_buf[i]) {
      clReleaseMemObject(input_b_buf[i]);
    }
    if(output_buf && output_buf[i]) {
      clReleaseMemObject(output_buf[i]);
    }
  }

  if(program) {
    clReleaseProgram(program);
  }
  if(context) {
    clReleaseContext(context);
  }
}
```

APPENDIX C

OpenCL Histogram Program

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* OpenCL includes */
#include <CL/cl.h>

/* Utility functions */
#include "utils.h"
#include "bmp-utils.h"
#include "gold.h"

static const int HIST_BINS = 256;

int main(int argc, char **argv)
{
   /* Host data */
   int *hInputImage = NULL;
   int *hOutputHistogram = NULL;

   /* Allocate space for the input image and read the
    * data from disk */
   int imageRows;
   int imageCols;
   hInputImage = readBmp("../../Images/cat.bmp", &imageRows, &imageCols);
   const int imageElements = imageRows*imageCols;
   const size_t imageSize = imageElements*sizeof(int);

   /* Allocate space for the histogram on the host */
   const int histogramSize = HIST_BINS*sizeof(int);
   hOutputHistogram = (int*)malloc(histogramSize);
   if (!hOutputHistogram) { exit(-1); }

   /* Use this to check the output of each API call */
   cl_int status;

   /* Get the first platform */
   cl_platform_id platform;
   status = clGetPlatformIDs(1, &platform, NULL);
   check(status);

   /* Get the first device */
   cl_device_id device;
   status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
   check(status);

   /* Create a context and associate it with the device */
   cl_context context;
   context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
   check(status);

   /* Create a command queue and associate it with the device */
   cl_command_queue cmdQueue;
   cmdQueue = clCreateCommandQueueWithProperties(context, device, 0, &status);
   check(status);

   /* Create a buffer object for the input image */
   cl_mem bufInputImage;
   bufInputImage = clCreateBuffer(context, CL_MEM_READ_ONLY, imageSize, NULL,
         &status);
   check(status);

   /* Create a buffer object for the output histogram */
   cl_mem bufOutputHistogram;
   bufOutputHistogram = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
      histogramSize, NULL, &status);
   check(status);

   /* Write the input image to the device */
   status = clEnqueueWriteBuffer(cmdQueue, bufInputImage, CL_TRUE, 0,
imageSize,
         hInputImage, 0, NULL, NULL);
   check(status);

   /* Initialize the output histogram with zeros */
   int zero = 0;
   status = clEnqueueFillBuffer(cmdQueue, bufOutputHistogram, &zero,
         sizeof(int), 0, histogramSize, 0, NULL, NULL);
   check(status);

   /* Create a program with source code */
   char *programSource = readFile("histogram.cl");
   size_t programSourceLen = strlen(programSource);
   cl_program program = clCreateProgramWithSource(context, 1,
      (const char**)&programSource, &programSourceLen, &status);
   check(status);

   /* Build (compile) the program for the device */
   status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
   if (status != CL_SUCCESS) {
      printCompilerError(program, device);
      exit(-1);
   }

   /* Create the kernel */
   cl_kernel kernel;
   kernel = clCreateKernel(program, "histogram", &status);
   check(status);

   /* Set the kernel arguments */
   status  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufInputImage);
   status |= clSetKernelArg(kernel, 1, sizeof(int), &imageElements);
   status |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufOutputHistogram);
   check(status);

   /* Define the index space and work-group size */
   size_t globalWorkSize[1];
   globalWorkSize[0] = 1024;

   size_t localWorkSize[1];
   localWorkSize[0] = 64;
   /* Enqueue the kernel for execution */
   status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL,
      globalWorkSize, localWorkSize, 0, NULL, NULL);
   check(status);

   /* Read the output histogram buffer to the host */
   status = clEnqueueReadBuffer(cmdQueue, bufOutputHistogram, CL_TRUE, 0,
         histogramSize, hOutputHistogram, 0, NULL, NULL);
   check(status);

   /* Verify the output */
   int *refHistogram;
   refHistogram = histogramGold(hInputImage, imageRows*imageCols, HIST_BINS);
   int passed = 1;
   int i;
   for (i = 0; i < HIST_BINS; i++) {
      if (hOutputHistogram[i] != refHistogram[i]) {
         passed = 0;
      }
   }
   if (passed) {
      printf("Passed!\n");
   }
   else {
      printf("Failed.\n");
   }
   free(refHistogram);

   /* Free OpenCL resources */
   clReleaseKernel(kernel);
   clReleaseProgram(program);
   clReleaseCommandQueue(cmdQueue);
   clReleaseMemObject(bufInputImage);
   clReleaseMemObject(bufOutputHistogram);
   clReleaseContext(context);

   /* Free host resources */
   free(hInputImage);
   free(hOutputHistogram);
   free(programSource);

   return 0;
}
```

APPENDIX D

Black Sholes Original C Code

```
#include <algorithm>    // Needed for the "max" function
#include <cmath>
#include <iostream>


// A simple implementation of the Box-Muller algorithm, used to generate
// gaussian random numbers - necessary for the Monte Carlo method below
// Note that C++11 actually provides std::normal_distribution<> in
// the <random> library, which can be used instead of this function
double gaussian_box_muller() {
	double x = 0.0;
	double y = 0.0;
	double euclid_sq = 0.0;

	// Continue generating two uniform random variables
	// until the square of their "euclidean distance"
	// is less than unity
	do {
		x = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
		y = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
		euclid_sq = x*x + y*y;
	} while (euclid_sq >= 1.0);
	return x*sqrt(-2 * log(euclid_sq) / euclid_sq);
}

// Pricing a European vanilla call option with a Monte Carlo method
double monte_carlo_call_price(const int& num_sims, const double& S, const
double& K, const double& r, const double& v, const double& T) {
	double S_adjust = S * exp(T*(r - 0.5*v*v));
	double S_cur = 0.0;
	double payoff_sum = 0.0;

	for (int i = 0; i<num_sims; i++) {
		double gauss_bm = gaussian_box_muller();
		S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
		payoff_sum += std::max(S_cur - K, 0.0);
	}

	return (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
}

// Pricing a European vanilla put option with a Monte Carlo method
double monte_carlo_put_price(const int& num_sims, const double& S, const
double& K, const double& r, const double& v, const double& T) {
	double S_adjust = S * exp(T*(r - 0.5*v*v));
	double S_cur = 0.0;
	double payoff_sum = 0.0;

	for (int i = 0; i<num_sims; i++) {
		double gauss_bm = gaussian_box_muller();
		S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
		payoff_sum += std::max(K - S_cur, 0.0);
	}
```

```
        return (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
}

int main(int argc, char **argv) {


        // First we create the parameter list
        int num_sims = 100000;   // Number of simulated asset paths
        double S = 100.0;  // Option price
        double K = 100.0;  // Strike price
        double r = 0.05;   // Risk-free rate (5%)
        double v = 0.2;    // Volatility of the underlying (20%)
        double T = 1.0;    // One year until expiry

                                      // Then we calculate the call/put values
via Monte Carlo
        double call = monte_carlo_call_price(num_sims, S, K, r, v, T);
        double put = monte_carlo_put_price(num_sims, S, K, r, v, T);

        // Finally we output the parameters and prices
        std::cout << "Number of Paths: " << num_sims << std::endl;
        std::cout << "Underlying:      " << S << std::endl;
        std::cout << "Strike:          " << K << std::endl;
        std::cout << "Risk-Free Rate:  " << r << std::endl;
        std::cout << "Volatility:      " << v << std::endl;
        std::cout << "Maturity:        " << T << std::endl;

        std::cout << "Call Price:      " << call << std::endl;
        std::cout << "Put Price:       " << put << std::endl;

        return 0;
                                        }
```

APPENDIX E

Black Sholes OpenCL Code

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
#include <CL_2_0\cl.h>
#include <c:/altera/15.1/aclrte-windows64/host/include/CL/opencl.h>
#include <iostream>
#include <algorithm>    // Needed for the "max" function
#include <cmath>
#include "c:/altera/15.1/aclrte-
windows64/board/de1soc/examples/common/inc/aocl_utils.h"



//Kernel
const char* programSource =
"__kernel                                                          \n"
"void vecadd(__int num_elements,                                   \n"
"                           __global double *A,                    \n"
"               __global double *B,                                \n"
"               __global double *C)                                \n"
"{                                                                 \n"
"                                                                  \n"
"int idx = get_global_id(0);                                       \n"
"double euclidSq = 0.0;                                            \n"
"double x_select = 0.0;                                            \n"
"for (int i = 0; i<num_elements; ++i){                             \n"
"do {                                                              \n"
"       euclidSq = A[i]*A[i] + B[i]*B[i];                          \n"
"       x_select = A[i];                                           \n"
"    } while (euclidSq >= 1.0);                                    \n"
"    C[i] = x_select*sqrt(-2 * log(euclidSq) / euclidSq);          \n"
"}                                                                 \n"
"}                                                                 \n"
;
using namespace aocl_utils;

int main() {

       //**********Problem Data***********************
       const int elements = 100000;
       double S = 100.0;  // Option price
       double K = 100.0;  // Strike price
       double r = 0.05;   // Risk-free rate (5%)
       double v = 0.2;    // Volatility of the underlying (20%)
       double T = 1.0;    // One year until expiry
       double call, put;


       size_t datasize = sizeof(double)*elements;

       double *A = (double*)malloc(datasize);
       double *B = (double*)malloc(datasize);
       double *C = (double*)malloc(datasize);
```

```
//Loads the random x & y values into arrays A and B in host memory
int i;
for (i = 0; i < elements; i++) {
       A[i] = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
       B[i] = 2.0 * rand() / static_cast<double>(RAND_MAX) - 1;
}

cl_int status;

//Find the OpenCL Platform
cl_platform_id platform;
status = clGetPlatformIDs(1, &platform, NULL);


//Get the Devices
cl_device_id device;
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, NULL);

//Create the Context
cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL,
&status);

//Create the command queue for the device chosen
cl_command_queue cmdQueue = clCreateCommandQueue(context, device, 0,
&status);

//Create the buffers for memory transfers between host and device
cl_mem bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize,
       NULL, &status);
cl_mem bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize,
       NULL, &status);
cl_mem bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize,
       NULL, &status);

//Write the data from host memory to the kernel buffers
status = clEnqueueWriteBuffer(cmdQueue, bufA, CL_FALSE, 0,
       datasize, A, 0, NULL, NULL);
status = clEnqueueWriteBuffer(cmdQueue, bufB, CL_FALSE, 0,
       datasize, B, 0, NULL, NULL);

//Create the program by extracting the kernel
cl_program program = clCreateProgramWithSource(context, 1,
       (const char**)&programSource, NULL, &status);

//Build the program for the device
status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);

//Create the kernel object
cl_kernel kernel = clCreateKernel(program, "vecadd", &status);

//Set the arguments for the kernel
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
status = clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);

//Define an index space of wor-items for execution
size_t indexSpaceSize[1], workGroupSize[1];
```

```
//These are 'elements' wor-items
indexSpaceSize[0] = elements;
workGroupSize[0] = elements;

//Execute the kernel
status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL,
indexSpaceSize,
       workGroupSize, 0, NULL, NULL);

//Read the device output buffer to the host output array
status = clEnqueueReadBuffer(cmdQueue, bufC, CL_TRUE, 0,
       datasize, C, 0, NULL, NULL);

//*****************call function**********************************
double S_adjust = S * exp(T*(r - 0.5*v*v));
double S_cur = 0.0;
double payoff_sum = 0.0;

for (int j = 0; j < elements; j++) {
       double gauss_bm = C[j]; //pulls the values returned by the kernel
       //printf("%d", gauss_bm);
       S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
       payoff_sum += std::max(S_cur - K, 0.0);
}

call = (payoff_sum / static_cast<double>(elements)) * exp(-r*T);
//*****************************************************************

//****************put function***********************************
S_adjust = S * exp(T*(r - 0.5*v*v));
S_cur = 0.0;
payoff_sum = 0.0;

for (int j = 0; j < elements; j++) {
       double gauss_bm = C[j]; //pulls the values returned by the kernel
       //printf("%d", gauss_bm);
       S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
       payoff_sum += std::max(K - S_cur, 0.0);
}

put = (payoff_sum / static_cast<double>(elements)) * exp(-r*T);
//*****************************************************************


// Finally we output the parameters and prices**********
std::cout << "Number of Paths: " << elements << std::endl;
std::cout << "Underlying:      " << S << std::endl;
std::cout << "Strike:          " << K << std::endl;
std::cout << "Risk-Free Rate:  " << r << std::endl;
std::cout << "Volatility:      " << v << std::endl;
std::cout << "Maturity:        " << T << std::endl;

std::cout << "Call Price:      " << call << std::endl;
std::cout << "Put Price:       " << put << std::endl;
//*******************************************************
```

```
        //Free OpenCL resources
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(cmdQueue);
        clReleaseMemObject(bufA);
        clReleaseMemObject(bufB);
        clReleaseMemObject(bufC);
        clReleaseContext(context);

        //Free host resources
        free(A);
        free(B);
        free(C);

        return 0;
}
```

BIBLIOGRAPHY

Altera Corporation. *"Running OpenCL on Altera FPGAs."* Altera online training. https://www.altera.com/customertraining/webex/OpenCLRun/launcher.html (accessed December 25, 2016).

AMD & ATI Stream Technology, 2010 Training Guide: *Introduction to OpenCL Programming.*

AMD. *"What is Heterogeneous System Architecture (HAS)?.* AMD Developer Central. http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/ (accessed December 29, 2016).

Banger Ravishekhar, and Koushik Bhattacharyya. *OpenCL Programming by Example*: Birmingham, UK: Packt Publishing, 2013.

Chen Doris, Deshanand Singh. *Using OpenCL to Evaluate the Efficiency of CPUs, GPUs, and FPGAs for Information Filtering*. Invited Paper, Altera Toronto Technology Center, 2013.

Chen Doris. *Fractional Video Compression in OpenCL: An Evaluation of CPUs, GPUs, and FPGAs as Acceleration Platforms.* Invited Paper, Altera Toronto Technology Center, 2013.

Guillon, AJ. *An Introduction to OpenCL C++*. The Khronos Group Inc., 2015.

Kaeli David, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous Computing with OpenCL 2.0*: New York: Morgan Kaufmann, 2015.

Moore, Andrew. *FPGAs for Dummies: Altera Special Addition.* Hoboken: John Wiley & Sons Inc., 2014.

Ndu Geoffrey, Mikel Lujan, and Javier Navaridas. *CHO: A Benchmark Suite for OpenCL-based FPGA Accelerators*. University of Manchester, 2014.

Patterson David, and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*: New York: Morgan Kaufmann, 2009.

Patterson David, and John L. Hennessy. *Computer Architecture: A Quantitative Approach*: New York: Morgan Kaufmann, 2012.

Singh Desh. *Higher Level Programming Abstractions for FPGAs using OpenCL:* Toronto Technology Center, Altera Corporation, 2011.

Singh, Desh, and Supervising Principal Engineer. *Higher Level Programming Abstractions for FPGAs using OpenCL.* Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing. 2011.