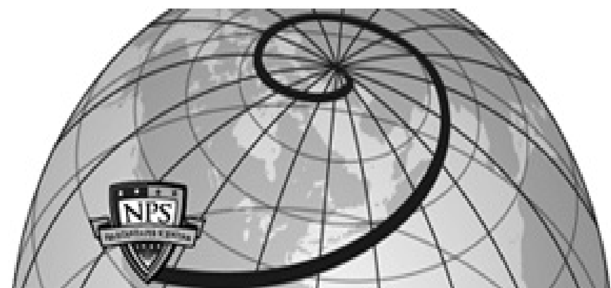
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations 1. Thesis and Dissertation Collection, all items

2014-06

HIGH PERFORMANCE COMPUTING FOR RECONNAISSANCE APPLICATIONS

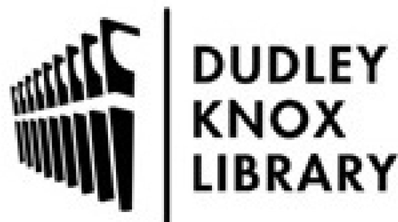
Stevens, Christopher J.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/70447

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

HIGH PERFORMANCE COMPUTING FOR RECONNAISSANCE APPLICATIONS

by

Christopher J. Stevens

June 2014

Thesis Co-Advisors: Douglas Fouts
Weilian Su

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE					*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2014	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE HIGH PERFORMANCE COMPUTING FOR RECONNAISSANCE APPLICATIONS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Christopher J. Stevens	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release;distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Parallel programming is vital to fully utilize the multicore architectures that dominate the processor market. The market, however, is constantly evolving, with new processors and new architectures getting released annually. Using an open parallel processing language, such as OpenCL (Open Computing Language), enables the use of a single program across multiple architectures. It also enables a method of evaluation between multiple devices so the best choice can be made for a given application. In this research, OpenCL is used to evaluate the performance of two signal processing algorithms across two graphics processing units and one central processing unit. Experimental results show that for each algorithm, a specific device can clearly be shown to outperform the others.

14. SUBJECT TERMS Fast Fourier transformation (FFT), binary phase-shift keying (BPSK), OpenCL, parallel processing, graphic processing unit (GPU), central processing unit (CPU), field-programmable gate array (FPGA)	**15. NUMBER OF PAGES** 75
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release;distribution is unlimited

HIGH PERFORMANCE COMPUTING FOR RECONNAISSANCE APPLICATIONS

Christopher J. Stevens
Ensign, United States Navy
B.S., United States Naval Academy, 2013

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2014

Author: Christopher J. Stevens

Approved by: Douglas Fouts
Thesis Co-Advisor

Weilian Su
Thesis Co-Advisor

Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Parallel programming is vital to fully utilize the multicore architectures that dominate the processor market. The market, however, is constantly evolving, with new processors and new architectures getting released annually. Using an open parallel processing language, such as OpenCL (Open Computing Language), enables the use of a single program across multiple architectures. It also enables a method of evaluation between multiple devices so the best choice can be made for a given application. In this research, OpenCL is used to evaluate the performance of two signal processing algorithms across two graphics processing units and one central processing unit. Experimental results show that for each algorithm, a specific device can clearly be shown to outperform the others.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION ... 1
A. PROBLEM STATEMENT ... 1
B. PROJECT GOAL ... 1
C. MOTIVATION ... 1
D. RELATED WORK ... 2
E. ORGANIZATION OF THESIS ... 3

II. PROBLEM DESCRIPTION ... 5
A. DIFFERENCES IN COMPUTING DEVICES ... 5
1. Central Processing Units ... 5
2. Graphics Processing Units ... 6
3. Field Programmable Gate Arrays ... 8
B. PARALLEL PROCESSING ... 8
C. SIGNAL MODULATION ... 10
D. FOURIER TRANSFORM ... 11

III. RESEARCH METHODS ... 15
A. OPENCL ... 15
B. DEVICES USED ... 18
1. NVIDIA GPUs ... 19
a. Tesla K20c ... 19
b. GeForce GTX 650 ... 20
2. Intel Xeon E5-2643 ... 20
3. Device Comparison ... 20
4. FPGA ... 21
C. TESTING ALGORITHMS ... 21
1. Binary Phase-Shift Keying ... 21
a. Motivation ... 21
b. OpenCL Algorithm ... 23
c. Demodulation ... 24
2. Fast Fourier Transform ... 24
a. Motivation ... 24
b. Algorithm ... 25

IV. EXPERIMENTAL RESULTS AND ANALYSIS ... 27
A. FFT RESULTS ... 27
1. Performance: Float Data Type ... 28
2. Performance: Double Data Type ... 31
3. Algorithm Output ... 32
B. BPSK PERFORMANCE ... 34
1. Initial Simulation ... 34
2. Additional Simulations ... 38
3. Demodulation ... 40
C. COMBINED ALGORITHM PERFORMANCE ... 43

V. CONCLUSION AND RECOMMENDATIONS 47
A. FINDINGS AND CONCLUSIONS 47
B. RECOMMENDATIONS FOR FUTURE WORK......................................48
1. FPGA..48
2. Additional Algorithms 48
3. Utilizing Multiple Devices 48
4. Recovery of the FFT Output 48

LIST OF REFERENCES 51

INITIAL DISTRIBUTION LIST 55

LIST OF FIGURES

Figure 1. Single-thread integer performance of CPUs relative to time based on data from the SPEC, after [5]. 5
Figure 2. Single-threaded floating-point performance of CPUs relative to time based on data from the SPEC, after [5]. 6
Figure 3. NVIDIA GPU and Intel CPU raw computing power in gigaFLOPS relative to time, from [7]. 7
Figure 4. Three binary digital modulation schemes: phase-shift keying, frequency-shift keying, and-amplitude shift keying, as well as the original binary bitstream. 10
Figure 5. A sinusoidal signal with its two component sinusoids and its Fourier transform. 12
Figure 6. A pictorial representation of a game of cards, from [9, p. 9] 16
Figure 7. A pictorial representation of kernel distribution among OpenCL-compliant devices, from [9, p. 8] 16
Figure 8. A visual representation of the deployment of work groups to compute units, from [9, p. 66]. 17
Figure 9. The OpenCL memory model, from [19]. 18
Figure 10. A block diagram of a BPSK modulator. 22
Figure 11. An example of a BPSK modulated signal 22
Figure 12. Runtime results of the FFT algorithm on all three devices using the float data type. 28
Figure 13. Runtime results of the FFT algorithm on the NVIDIA GPUs using the float data type. 29
Figure 14. Runtime results of the FFT algorithm on all three devices using the double data type. 32
Figure 15. Comparison of the FFT output for a signal $rect[n]$. The output of the MATLAB FFT is shown by $X_M(\omega)$ and the output of the OpenCL algorithm is shown by $X_O(\omega)$ 33
Figure 16. Performance results for the BPSK algorithm for all three devices. The number of samples per bit is held constant at 100. 35
Figure 17. Performance results for the BPSK algorithm for the three devices. The number of symbols is held constant at 1000. 36
Figure 18. Performance results of the BPSK algorithm for the NVIDIA devices with a fixed number of symbols (1000). 37
Figure 19. Performance of the BPSK algorithm on the Xeon to show similarities with the NVIDIA results. A fixed number of symbols (1000) is used. 38
Figure 20. Performance of the BPSK algorithm on all three devices with 16 samples per bit and a variable number of symbols. The plotted performance is the average of 10 runs. 39

Figure 21. Performance of the demodulation kernel on all three devices. The horizontal axis is the number of integers resultant from the demodulated signal. 40
Figure 22. Performance of the demodulation kernel on all three devices. The horizontal axis is the number of integers resultant from the demodulated signal. The plotted performance is the average of 15 runs. 41
Figure 23. Performance of the demodulation kernel on all three devices. The horizontal axis is the number of integers resultant from the demodulated signal. The plotted performance is the average of 10 runs. 43
Figure 24. Flow of the combined FFT and BPSK algorithms....44
Figure 25. Performance of the combined algorithm on all three devices using the average of 10 runs. 44
Figure 26. Performance of the IFFT and demodulation blocks of the combined algorithm on all three devices. 45

LIST OF TABLES

Table 1. Device specifications relevant to the conducted research. Note that the listed memory for the Xeon E5-2643 is actually the system RAM. 21

Table 2. Output of the FFT algorithm for the Tesla showing the possible recoverability of the incorrect GPU output...30

Table 3. A comparison of the signal lengths at which the FFT algorithm breaks down on the three devices with the float data type...31

Table 4. A comparison of the signal lengths at which the FFT algorithm breaks down on the three devices with the double data type. 32

Table 5. RMSE values for the FFT algorithm using a signal length of 16384. The algorithm does not operate on the GeForce for a length 16384 signal using the double data type. 34

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ASW	antisubmarine warfare
BPSK	binary phase-shift keying
CONOPS	concept of operations
COTS	commercial off-the-shelf
CPU	central processing unit
CUDA	Compute Unified Device Architecture
DFT	discrete Fourier transform
DSP	digital signal processing
FFT	fast Fourier transform
FLOPS	floating-point operations per second
FPGA	field-programmable gate array
GPGPU	general-purpose computing on graphics processing units
GPU	graphics processing unit
GUI	graphical user interface
IFFT	inverse FFT
MIC	Many Integrated Core
OpenCL	Open Computing Language
RAM	random access memory
RMSE	root mean square error
SDK	software development kit
SIGINT	signals intelligence
SM	Streaming Multiprocessor
SPEC	Standard Performance Evaluation Corporation

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Utilizing the most powerful computing equipment available is vital to processing the increasing amount of data collected by increasingly powerful sensors. The fast processing of collected data and the information required by the systems is vital to enabling the Navy to maintain its lead in control of the seas. Alternatively, utilizing more powerful but less expensive processors allows for a larger number to be utilized, potentially allowing for a wider area of coverage. The multitude of different processors available, and the multitude of architectures used by those processors, makes it difficult to determine what the most powerful computing equipment is for a given application without evaluation.

Current processor architectures are almost entirely multicore. This allows for greater performance but also requires programming methods that are much different from standard, sequential programming. Open Computing Language (OpenCL) is a parallel programming language that contains a library of functions and data structures that serve as an open-source method for using the C language for parallel programming. The OpenCL developers leave the creation of the compiler for the language up to the processor manufacturers. Although this causes some fragmentation in the market, it enables each developer to focus on optimizing the library for their specific device architecture.

Central processing units (CPUs) and graphics processing units (GPUs) are two of the devices present in virtually all modern computing systems. CPUs are the backbone of the majority of computing systems, while GPUs are the devices that enable the visual display of all the information provided by the system. As a result of their prevalence, there is strong competition between manufacturers to create the most powerful and least expensive device. They are, therefore, an important area for consideration as a component in a processing system. In this research, two GPUs and one CPU were evaluated. The CPU tested was the Intel Xeon E5-2643, while the two GPUs were both provided by NVIDIA. The GeForce GTX 650 is a “typical” GPU, while the Tesla K20c is a member of a family of GPUs designed solely for data processing.

For this research, two signal processing algorithms were used to evaluate the three processors. The Fourier transform is extremely important to signal processing, allowing for the determination of the frequencies present in the incoming signal. In the processing of acoustic data this is particularly important, as it is one of the few ways to discriminate between underwater targets. The fast Fourier transform (FFT) algorithm is one of the most efficient methods for calculating the Fourier transform on a finite sequence. Modulation is vital to communications as before a signal can be transmitted through any medium it must be modulated. Shift keying functions change a parameter of a carrier sinusoid, depending on the binary symbol to be transmitted. Binary phase-shift keying (BPSK) modulates the phase of the carrier wave using single binary bits as the modulating symbol. An FFT and BPSK algorithm were, therefore, both used to evaluate the three devices. This includes the inverse FFT and a BPSK demodulation algorithm.

The two test algorithms delivered different results for the performance of the processors. In addition to requiring that an input signal be a power of two, the FFT algorithm did not operate on each of the devices at the test signal lengths. Due to device specifications, the CPU could not perform the algorithm for small signal lengths, while the two GPUs were unable to operate on large signal lengths. For the operable signal lengths, however, the two GPUs vastly outperformed the CPU as expected. The GeForce, for the smaller signal lengths, outperformed the Tesla. This was contrary to the original hypothesis. The IFFT (inverse FFT) algorithm, as it only added a single function to scale the results of the FFT algorithm, performed similarly.

When simulating the BPSK algorithm for a large amount of data, the GeForce performed the slowest of the three devices. The Tesla performed the fastest and required time on the order of milliseconds. With a limited but more realistic set of parameters for the modulation scheme, the GeForce and CPU outperform the Tesla for a brief period. The CPU performance, however, is hurt by its necessity to act as the host of the application, performing computations that the GPUs do not need to perform. The demodulation algorithm performed similarly to the forward modulation.

An additional application was used to evaluate the performance of the devices, a combination of the two algorithms and their inverse. A number of test sequences were

sent through BPSK modulator, the FFT, the IFFT, and the BPSK demodulator. Here the performance closely aligned with the same trend of the performance of the FFT algorithm, with the CPU performing worst and the GeForce outperforming the Tesla when a smaller number of symbols was used.

It is clear from the results that the performance of the devices is strongly affected by the algorithm used. In this case, although the Tesla performance was not the best in every situation, it was consistent. In certain situations, this is more important than occasional higher performance. In addition, cost must sometimes be considered. The Tesla costs $3,000 while the GeForce costs only $300. Either way, it is necessary for the specific algorithm that is to be employed get tested.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

In order to maintain its lead in sea control, it is necessary for the U.S. Navy to utilize the most powerful computing equipment available. Architectural differences in the various processors available, however, make it difficult to determine what the best computing equipment would be for a given application. Also, the ever-evolving technology market can make certain hardware obsolete within 18 months. It is clearly necessary to evaluate the available options prior to hardware selection. Using an open computer language for the testing platform allows for less expensive software development. Utilizing commercial off-the-shelf (COTS) processors as the hardware devices can provide a suitable, relatively inexpensive solution to the problem of the constantly changing market.

B. PROJECT GOAL

The goal of this research is to determine which of the three provided processors is best suited to perform two specific signal processing algorithms. Ideally, a single device will clearly outperform the others, making for a clear decision. Realistically, of course, this may not be the case.

C. MOTIVATION

In warfare, one of the key areas where processing data is important is signals intelligence (SIGINT). Maintaining an advantage in the area of SIGINT is vital to preserving spectral dominance. In the area of SIGINT, it is necessary to process the large amount of data that is received. Although ideally data processing and interpretation are always done as quickly as possible, speed is especially important in warfare. Decrypting a received or intercepted signal could be a matter of life or death.

Acoustics is another area that supplies a large amount of data that is required to be processed. This is increasingly true as sensors continue to become more powerful. The ability of sensors to collect more and more data directly correlates to a need to process

more and more data. However, it is unmistakably undesirable for the increase in data to also lead to an increase in the time it takes to process the data. It is essential, therefore, to update the hardware that processes the incoming data at the same time that the sensors collecting the data are upgraded.

In December of 2004, Chief of Naval Operations Admiral Vernon Clark approved the release of a document outlying the general vision of the U.S. Navy's newest concept of operations (CONOPS) for antisubmarine warfare (ASW). Inside the CONOPS, the use of a distributed network of miniaturized sensors is described several times [1]. Utilizing such a network of sensors would improve the understanding that warfighters have of the theater in which they are operating [2]. In order to create this type of network, it would likely be necessary to utilize small, high-performance processors that can do a large amount of the associated data processing necessary prior to that data getting sent to a collector for further analysis.

Finally, it is not just necessary to be able to process more data from external collectors. As technology continues to play an increasingly large role in all areas, it is necessary to improve the level of performance at which that technology operates. Submarines, for example, are now utilizing digital photonics masts in place of more traditional periscopes. This requires a large amount of processing capability.

D. RELATED WORK

Many studies have been conducted that have looked at the processing abilities of different computing devices. Due to its cross-platform operability, Open Computing Language (OpenCL) is frequently used for the evaluations. The majority of the studies have found that even though an OpenCL program can be run on any device for which the developer has created a compiler, performance across multiple devices is not necessarily portable. Of course, it could sometimes be unwise to compare the analysis of results from different devices due to differences in architectures, but in certain cases it can be useful. For example, it might be desired to explore how a specific algorithm runs on specific devices.

In certain cases, due to differences in how the different types of processors handle certain computations, performance portability can be abysmal. A study by researchers at the University of Chicago gathered initial results that found the "portable performance of three OpenCL programs is poor, generally achieving a low percentage of peak performance (7.5%–40% of peak GFLOPS and 1.4%–40.8% of peak bandwidth)" [3]. They identified several areas where portable performance could be improved, both in the test algorithms themselves and in the OpenCL compilers. After modifying the test algorithms portable performance was increased "from the current 15% to a potential 67% of the state-of-the-art performance" [3]. Although this wording is slightly vague, it does show that it is possible to improve performance portability. In addition, it must be noted that manually making changes to the test algorithms greatly increases development time—where the ability to compile OpenCL programs on multiple platforms without making changes is what makes the desired, decreased development time possible.

Researchers at China's National University of Defense Technology also identified areas to convert OpenCL programs written specifically for GPUs into programs that would run more efficiently on central processing units (CPUs). However, they found that in doing so the changes they made to the programs were "good for CPU [*sic*], while bad for GPU [*sic*]" [4]. In this case, the researchers manually transformed the OpenCL programs. They are also purportedly working on "developing an automatic transforming tool to implement [their] present work" [4]. Even automatic program transformations, however, make trade-offs for increased performance on one type of device over another and increase development time.

E. ORGANIZATION OF THESIS

The rest of this thesis is organized as follows. The problem at hand is discussed in Chapter II; namely, the two algorithms covered as well as the differences that exist between computing devices. A background to parallel processing is also provided in Chapter II. The methods by which the research was carried out are described in Chapter III, covering the OpenCL programming language, the three test devices, and the two

algorithms. The findings of the research are discussed in Chapter IV, and the overall conclusions as well as recommendations for future work are provided in Chapter V.

II. PROBLEM DESCRIPTION

A. DIFFERENCES IN COMPUTING DEVICES

As mentioned previously, different processing devices use different architectures. These differences are explored in this section.

1. Central Processing Units

Central processing units have been the core component in many computing systems since their inception. For the computing systems that have them, CPUs are the brain of the system, carrying out the instructions stored in memory. As semiconductor technology has improved and feature size decreased, CPU performance has vastly increased. In order to evaluate the performance of CPUs and other computing devices, the Standard Performance Evaluation Corporation (SPEC) collects and publishes various results of their benchmark programs. Shown in Figure 1 is a graph of the single-thread integer performance of various CPUs covering a period of 18 years, produced using a Python script written by Canadian programmer Jeff Preshing.

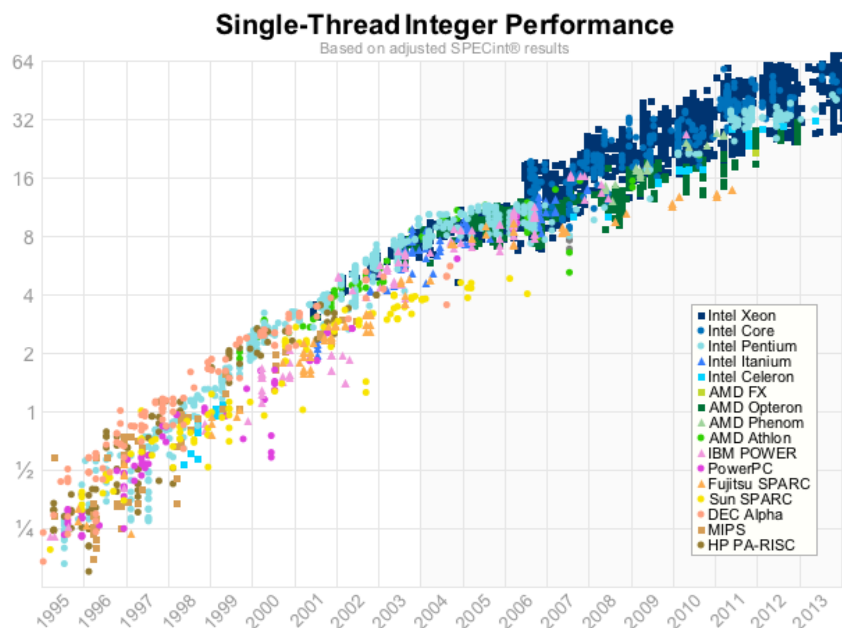


Figure 1. Single-thread integer performance of CPUs relative to time based on data from the SPEC, after [5].

Single-thread performance is used to remove any impact that parallel processing might have, allowing for a better comparison between older and newer CPUs. The results shown in Figure 1 are normalized to the performance of the Sun Ultra Enterprise 2 [6], with the performance of that machine having a value of one. It should be noted that the vertical axis scale is logarithmic. As can be seen, CPU performance increased dramatically for the first 10 years, and although the performance increase appears to have slowed slightly, it has not stopped improving. Single-thread floating-point performance can be seen in Figure 2 and shows much the same results as Figure 1.

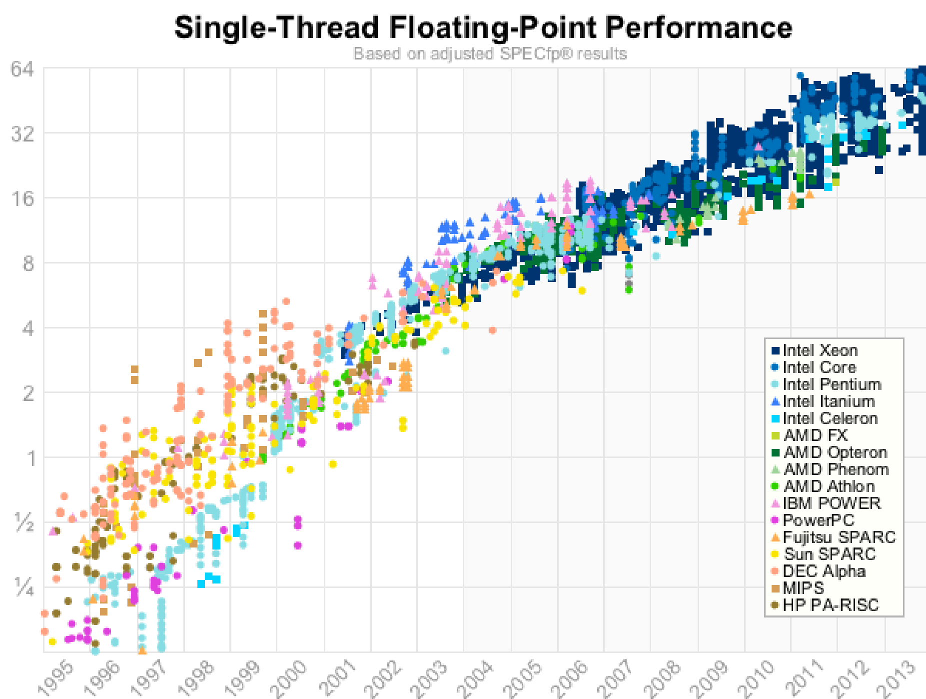


Figure 2. Single-threaded floating-point performance of CPUs relative to time based on data from the SPEC, after [5].

2. Graphics Processing Units

As personal computers became more and more commonplace, their operating systems shifted from providing text-based output, such as the Microsoft Disk Operating System, towards graphical user interfaces (GUI). As GUI displays became more complex and required more computationally intensive calculations, GPUs were developed to share the computational load with CPUs. Initially, graphics processors were not separate

devices as is currently the case. Rather, early GPUs were integrated with the computer's motherboard and shared the system random access memory (RAM). Some computer systems today—typically only laptop computers and netbooks that place an importance on size and weight—still use integrated graphics processors. Dedicated GPUs, however, have become increasingly powerful, following much the same trends that early CPUs did. In fact, even as CPU performance increases have slowed, GPU performance still swiftly improves each year [7]. A graph of the performance over time of NVIDIA's GeForce line of processors is shown in Figure 3, measured in gigaFLOPS (where one gigaFLOP is 10^9 floating point operations per second [FLOPS]). The GPU performance is compared to Intel's CPU performance over the same period.

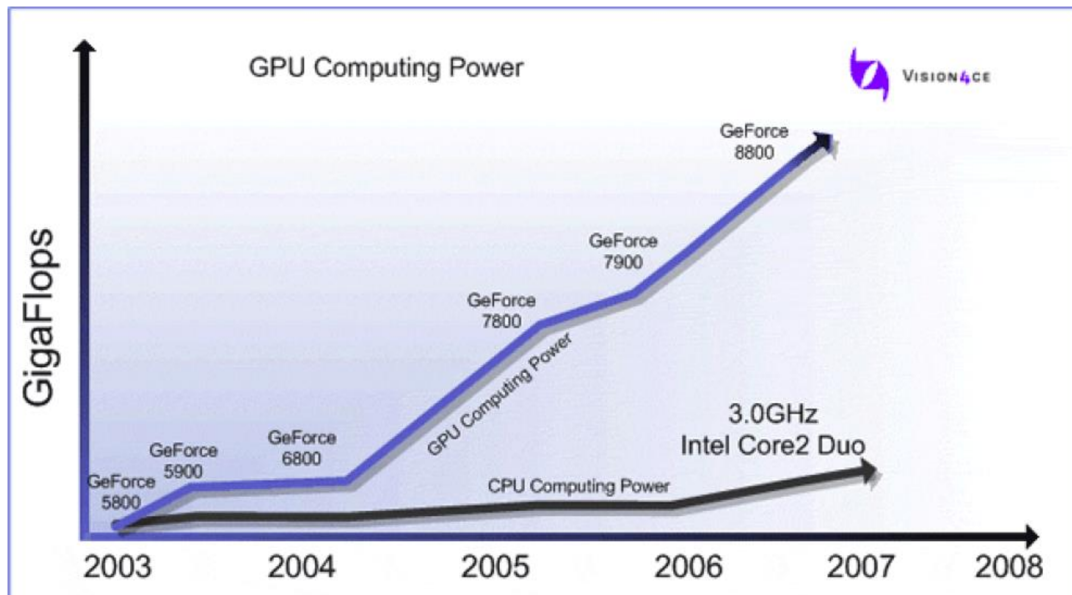


Figure 3. NVIDIA GPU and Intel CPU raw computing power in gigaFLOPS relative to time, from [7].

As GPUs became more powerful, it was realized that their high computational performance could be put to use in areas other than displaying GUIs. Modern GPUs are exclusively multicore devices. As such, they are inherently parallel. This is necessary to be able to handle the processing of thousands of pixels with multiple bits per pixel. Researchers recognized that they are well suited to handle the processing of large amounts of other data. GPUs dedicated to data processing, such as the NVIDIA Tesla

family of processors, have been developed that are in fact only able to operate as data processors—that is, they are unable to display graphics data.

3. Field Programmable Gate Arrays

Since their invention in the 1980's, field-programmable gate arrays (FPGA) have been used in many areas of computing. As their name suggests, FPGAs are reprogrammable devices that can be used and reused in a vast multitude of situations. FPGAs consist of large numbers of logic blocks that can be reconfigured using a hardware description language to suit the programmer's needs. FPGAs also contain memory blocks of varying sizes. In processing data, one of the key advantages FPGAs maintain over other devices is their low power requirement. Some FPGA power consumption can be on the order of milliwatts [8], whereas certain processors can require over 100 W. This, combined with the ability to optimize FPGA architectures to the specific problem at hand enable them to remain viable even with clock speeds that are typically much lower than other processors.

Programming FPGAs, however, is usually more difficult than writing regular programs. It typically requires an intimate knowledge of a hardware description language such as Verilog. For large applications, this can be extremely tedious and time consuming, greatly adding to the development cost. To aid in FPGA development, however, an FPGA developer, Altera, released a software development kit (SDK) that enables compilation of OpenCL applications on certain devices. Another manufacturer, Nallatech, has developed FPGA-based processors that utilize Altera FPGAs and the Altera OpenCL SDK. On the other hand, Xilinx is working on developing its own OpenCL SDK.

B. PARALLEL PROCESSING

It has already been shown that as the technology behind the production of computer devices has improved, the raw processing power of computing devices has increased. In order to prevent a plateau in performance, manufacturers began to create processors with multiple cores. The first multicore processor was released in 2001 by IBM [9, p. 5] and was used for multiple applications. Today, virtually all processors

produced have multiple cores. Even certain (relatively) low performance processors used in mobile computers—such as Intel's Atom family of processors—can be purchased that contain two or more cores [10]. On the other end of the spectrum, Intel's Many Integrated Core Architecture (MIC), used by Intel's Xeon Phi coprocessors, can provide power-hungry users with up to 61 cores [11], and each of the processors in the NVIDIA Tesla family of GPUs contains over 2,000 Compute Unified Device Architecture (CUDA) cores per processor [12].

In order to make full use of multicore devices, new programming styles and languages were developed; otherwise, running the programs that were created for single-core devices could potentially waste the processing power provided by the extra cores. Initial methods of parallel programming were convoluted, particularly when attempting to use GPUs for processing non-graphical data—the programmer had to "trick" the GPU into thinking it was performing graphics rendering tasks. In order to improve upon the programming process, a few months after the November 2006 release of its GeForce 8800 GTX GPU, NVIDIA released a public compiler for its CUDA C language [13]. By taking the C programming language and adding certain keywords, NVIDIA created the "first language specifically designed by a GPU company to facilitate general-purpose computing on GPUs" [13]. Because CUDA C is proprietary, however, it can only be compiled to run on NVIDIA processors. This limits its usefulness.

OpenCL is an open source alternative to NVIDIA's CUDA C. Originally released in 2008, OpenCL was initially developed by Apple and is currently maintained by one of the many working groups of the Khronos Group, a consortium of technology companies that define open standards for a variety of software APIs [14]. The Khronos Group leaves the development of the OpenCL compilers to each individual company, enabling improved performance on individual devices but also creating the potential for fragmentation. NVIDIA was the first company to release its compiler for OpenCL in early 2009, and AMD released its compiler several months later [9, p. 5]. OpenCL version 2.0 was released in 2013 [15]. As each company creates its own compiler, however, not all vendors have updated their compilers, supporting only earlier versions of the language.

C. SIGNAL MODULATION

In order to transmit data over long distances, some type of modulation scheme must be used to manipulate some parameter of the analog sinusoidal carrier signal and convey the data. The modulation can itself be either analog or digital, both of which manipulate the carrier sinusoid amplitude, frequency, or phase. Digital modulation schemes use the concept that when conveying digital data there are a limited, discrete number of symbols that can be transmitted. The digital schemes all follow the general equation $N = 2^n$ where N represents the number of possible symbols that can modulate the carrier wave and n represents the number of bits per symbol. As a result of the limited number of modulation symbols, the variable parameter of the carrier sinusoid also has a limited number of possible values. Signals modulated by an analog method, on the other hand, can theoretically have an infinite number of values. Examples of three binary digital modulation schemes (where the modulation symbols are single bits) are shown in Figure 4.

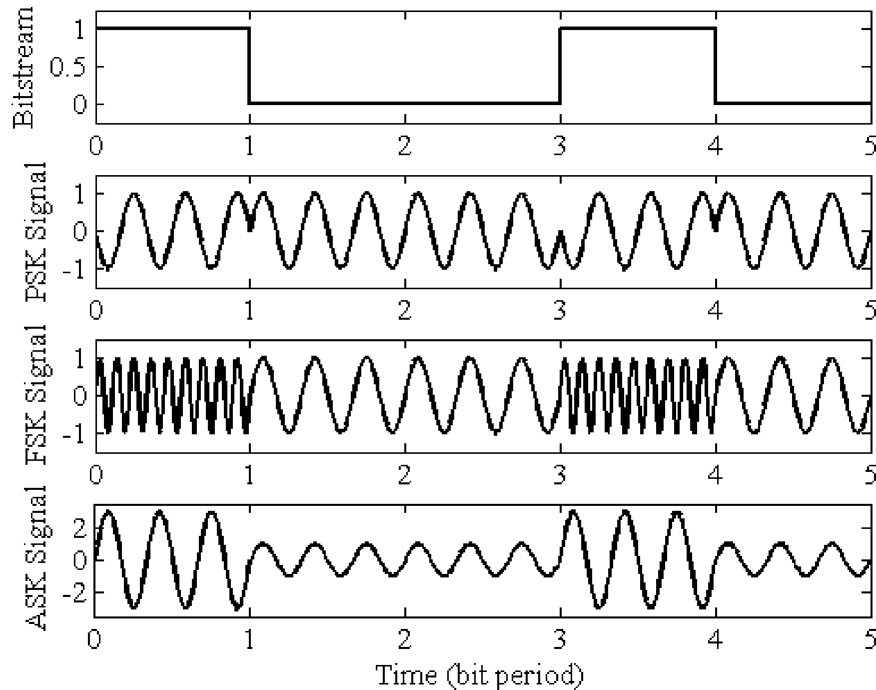


Figure 4. Three binary digital modulation schemes: phase-shift keying, frequency-shift keying, and-amplitude shift keying, as well as the original binary bitstream.

Once a signal has been modulated, transmitted, and received, it must be demodulated. This is done by matching the value of the carrier sinusoid modulation parameter to its potential values. Whether using analog or digital modulation, this is a fairly straightforward process, particularly when a scheme that modulates either amplitude or frequency is used. If a phase modulation scheme is used, the demodulator must be synchronized with the signal before the demodulation can take place. Otherwise, the interpretation of the transmitted symbols will be incorrect.

D. FOURIER TRANSFORM

One of the first steps typically done in signal analysis is the determination of the frequencies present in the incoming signal. This is done by taking the Fourier transform of the signal. The Fourier transform is calculated as

$$G(f) = \mathcal{F}\left[g(t)\right] = \int_{-\infty}^{\infty} g(t)e^{-j2\pi ft}dt, \tag{1}$$

where $g(t)$ is the incoming signal, integrated over time t, and $G(f)$ is the value of the Fourier transform over frequency f. Conversely, the inverse Fourier transform is determined by

$$g(t) = \mathcal{F}^{-1}\left[G(f)\right] = \int_{-\infty}^{\infty} G(f)e^{j2\pi ft}df. \tag{2}$$

Both of these equations can also be represented in terms of angular frequency ω, where $\omega = 2\pi f$. Substituting this into Equation (1) gives

$$\mathcal{F}\left[g(t)\right] = \int_{-\infty}^{\infty} g(t)e^{-j\omega t}dt \tag{3}$$

When calculating the inverse Fourier transform in terms of ω, it is necessary to divide the result by 2π due to the relation between ω and f.

The importance of the Fourier transform lies in the physical information it gives about the transformed signal. The spectrum given by the Fourier transform indicates the properties of the sinusoids necessary to recreate the signal (the phases and relative amplitudes of the sinusoids) [16, p. 96]. Periodic signals have Fourier spectrums that have discrete frequencies with finite amplitudes. Consider a signal

$$y(t) = 0.7\sin\left(2\pi 50t\right) + \sin\left(2\pi 120t\right) \quad (4)$$

where $y(t)$ is comprised of two sinusoids. This signal, along with its single-sided Fourier transform (which has been truncated to more easily show the relevant portion of the spectrum) is shown in Figure 5. The signals here are sampled with a sample rate of 10.0 kHz. Although it is easy to see that the signal $y(t)$ is periodic, it is clear that it is not a single sinusoid. It is also difficult to visually determine what the component sinusoids may be. Taking the Fourier transform easily enables the determination of any and all sinusoids present in the signal. This is, therefore, a very important algorithm for signal processing.

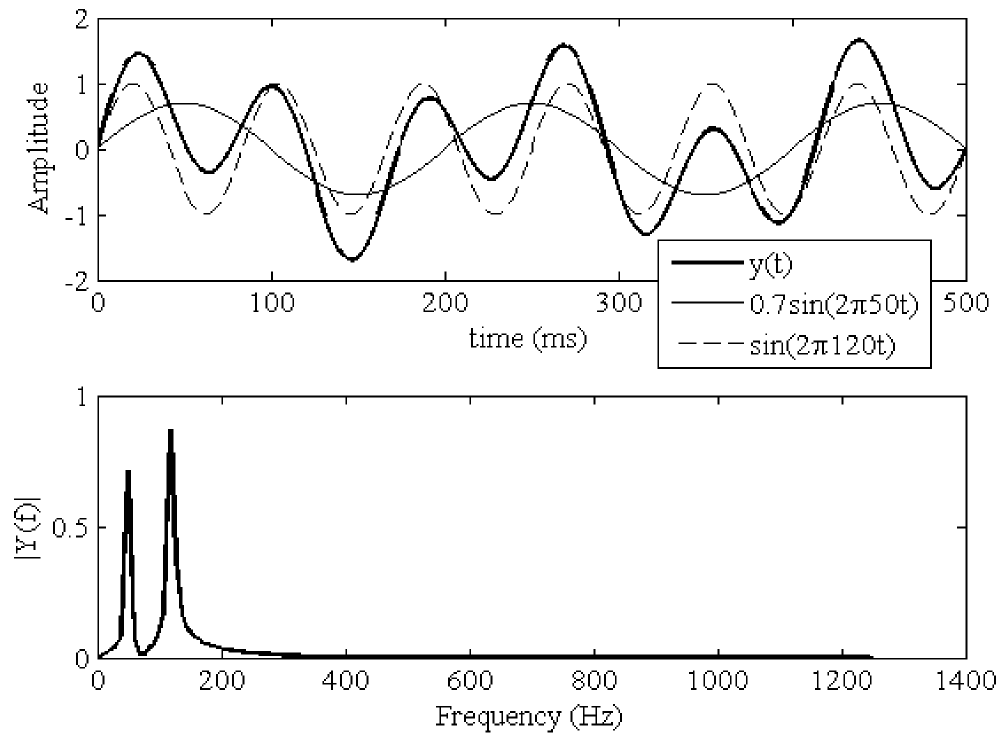


Figure 5. A sinusoidal signal with its two component sinusoids and its Fourier transform.

In order to calculate the Fourier transform of a signal using computers, the discrete Fourier transform (DFT) is used. For a sequence

$$x = x_0, x_1, \cdots, x_{N-1} \tag{5}$$

having length N, the DFT is

$$X[k] = \sum_{n=0}^{N-1} \omega_N^{kn} x[n] \tag{6}$$

where

$$\omega_N = e^{-j\frac{2\pi}{N}}. \tag{7}$$

Calculating the DFT directly, however, is computationally expensive, having complexity of order $O(n^2)$. The fast Fourier transform (FFT) algorithm was developed in 1965 as a method to more efficiently calculate the Fourier transform of a discrete sequence and has complexity of only order $O(n \log n)$ [17]. Although not a new algorithm by any means, the efficiency of the FFT, when combined with the importance of the Fourier transform, has made it one of the most important algorithms developed in the last century [18].

THIS PAGE INTENTIONALLY LEFT BLANK

III. RESEARCH METHODS

A. OPENCL

As previously discussed, OpenCL is an open, "royalty-free native, cross-platform, cross-vendor standard" for use in the area of parallel programming on heterogeneous systems [15]. OpenCL is not a completely separate language. Instead, it is primarily a library of functions and data structures that enable programmers to write code using C that runs on any device for which there is a compiler. Utilizing an open computer language and library has several benefits over a proprietary one. First and foremost among these benefits is the ability to run on a vast multitude of systems, preventing a system from remaining in use simply because it is the only workable solution. In addition, because there is generally a large population of users, it is easier to both find bugs in the language and fix those bugs.

For one knowledgeable in the CUDA C language, the OpenCL programming model appears very similar—many of the differences between the two languages are only differences in terminology. For one without much experience in parallel programming, on the other hand, the OpenCL library is daunting. OpenCL calls any function that is intended to be run on one or more target devices a *kernel*. The list of possible kernels that can be called is known as the *program* [9, p. 7]. When kernels are called they are placed into a *queue*, which is itself a member of the *context*, which keeps track of the queue of the device and any associated data [9, p. 7]. All of these structures are managed by the application *host*. One of the best analogies for the organization of an OpenCL application is one comparing it to a game of cards, which can be seen in Figures 6 and 7.

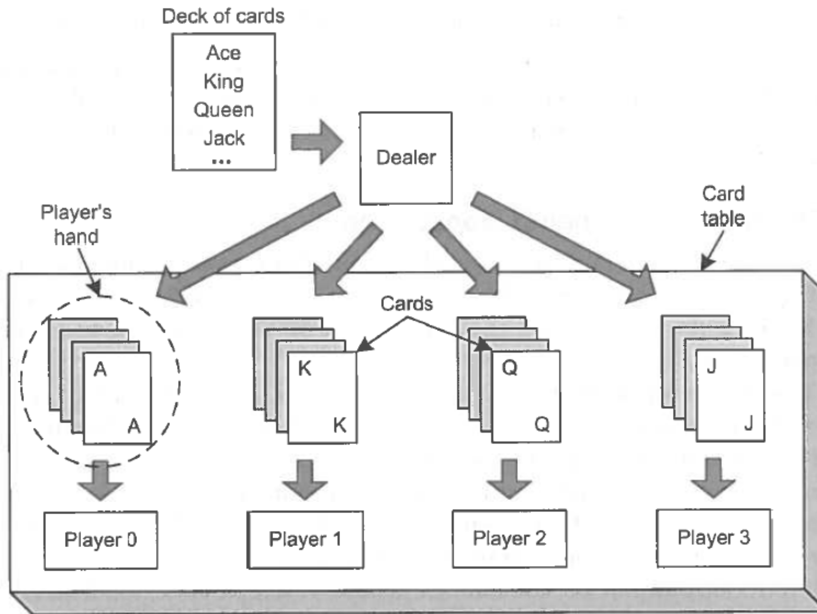


Figure 6. A pictorial representation of a game of cards, from [9, p. 9]

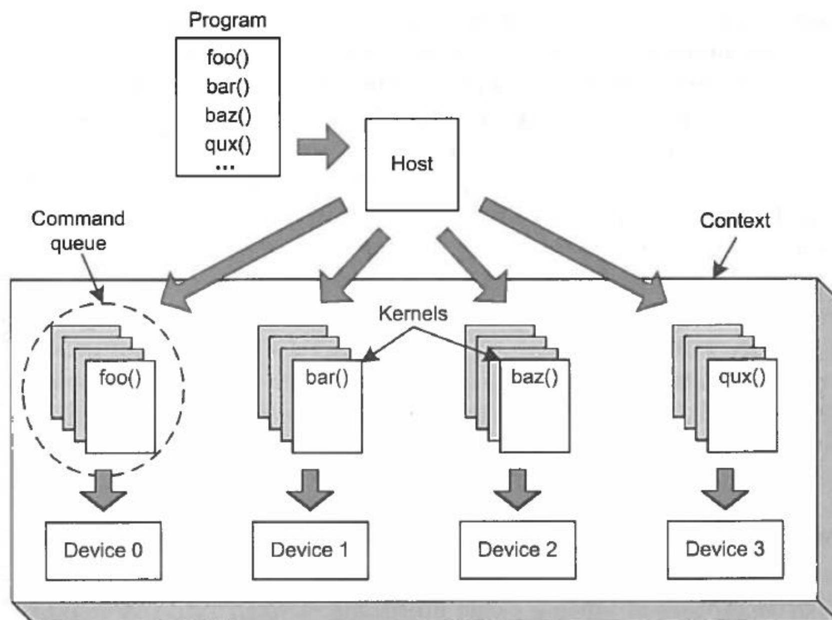


Figure 7. A pictorial representation of kernel distribution among OpenCL-compliant devices, from [9, p. 8]

In addition to the organization of the application, an understanding about the break-up of the device is important to the OpenCL programmer. When a kernel is sent to the device, it runs multiple times according to the method by which it is called. Each

instantiation of the kernel runs on the smallest OpenCL processing element, the *work item*. Work items are organized into larger *work groups* [9, p. 65], with potentially thousands of work groups per *compute device*—the applications target device. A final important OpenCL term is the *compute unit*, which is the processing block handling the work group [9, p. 66]. A compute device contains multiple compute units, which can only execute on a single work group at a time [9, p. 66]. This is shown in Figure 8.

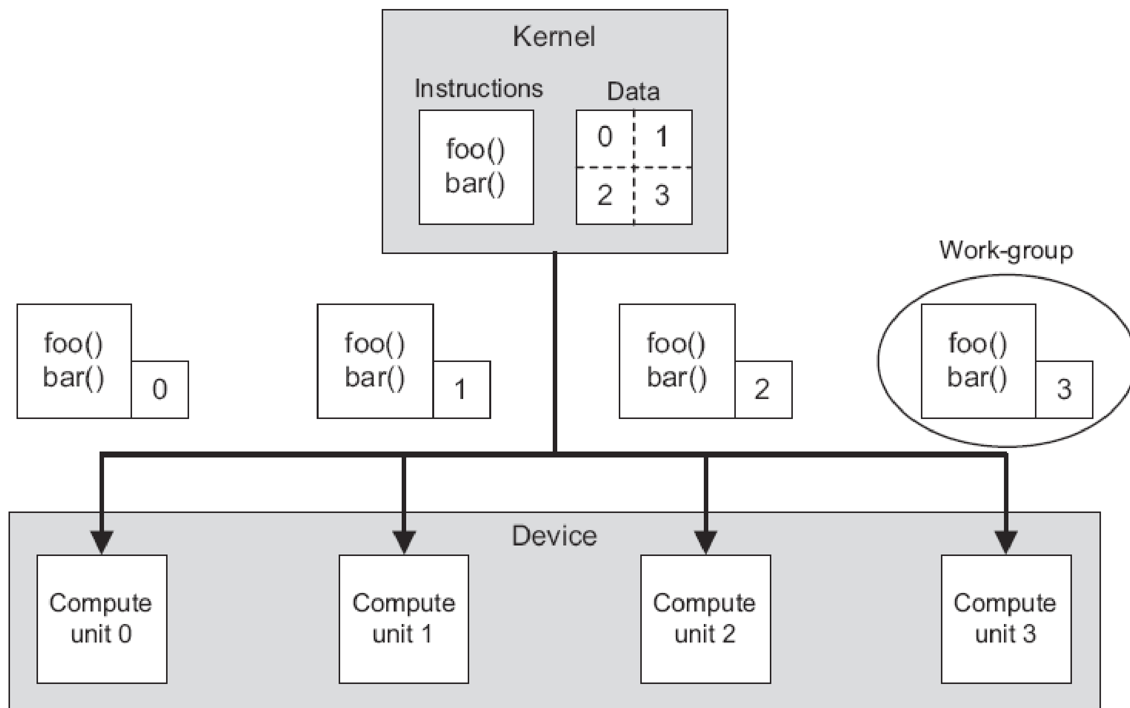


Figure 8. A visual representation of the deployment of work groups to compute units, from [9, p. 66].

Finally, the OpenCL memory model is extremely important. In OpenCL, memory is classified as one of three types: *global*, *local*, and *private*. Global memory can be accessed by any work item of any work group, whereas local memory can only be accessed by work items of a specific group [9, p. 88]. Finally, private memory can only be accessed by a single work item [9, p. 88]. Global memory exists only in *buffers* that must be created by the host prior to the launch of the kernel. Local memory is normally initialized by the work items after the kernel has been launched, but local memory arrays

must first be defined by the host. Private memory has the smallest size and quickest access times of the three types while global memory is the largest and slowest [9, p. 88]. A pictorial representation of the types of memory is shown in Figure 9.

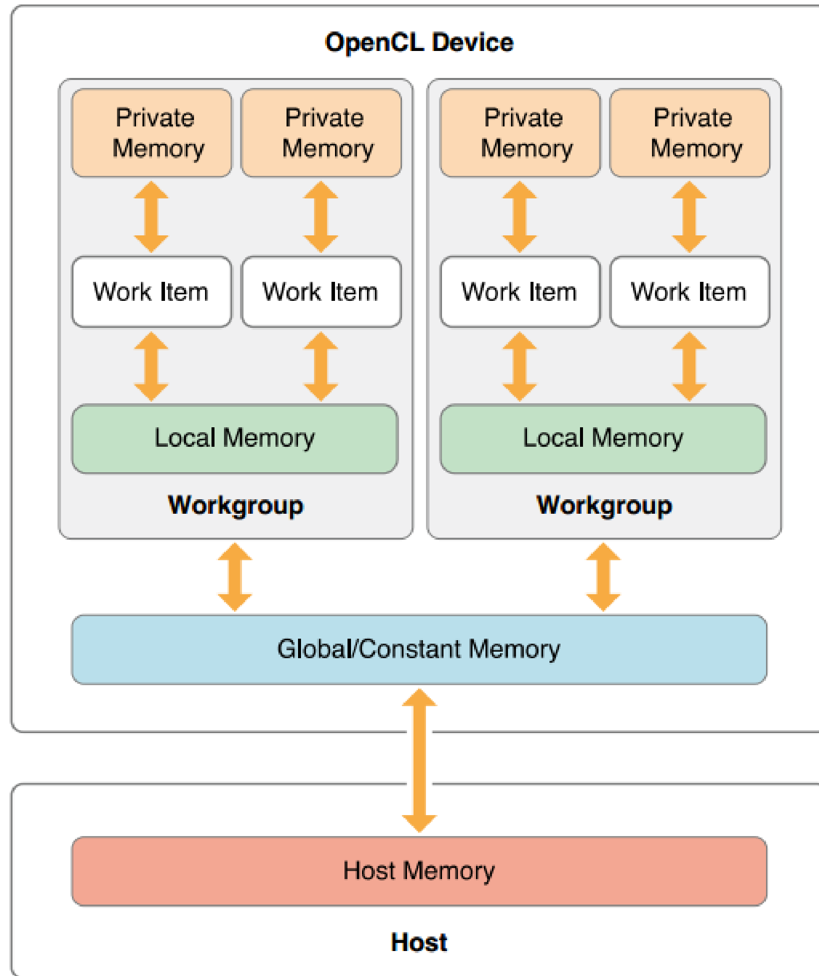


Figure 9. The OpenCL memory model, from [19].

B. DEVICES USED

For this research project, three different processors were used from two different manufacturers. Two GPUs were used. The Tesla K20c and GeForce GTX 650 were both designed by NVIDIA. The one CPU used, a Xeon E5-2643, was designed by Intel. The

majority of the processor specifications to follow were obtained through OpenCL, while memory bandwidth and power consumption numbers were obtained through product datasheets.

1. NVIDIA GPUs

Both the Tesla K20c and GeForce GTX 650 use NVIDIA's latest compute architecture, named Kepler. Designed for maximum performance, the Kepler architecture is built with several features that give it strength in the area of data processing. The Kepler architecture utilizes newly designed "Streaming Multiprocessors (SM)" [20] (NVIDIA's organizational unit for groups of compute cores). These SMs each contain 192 CUDA cores [20]. In addition, the Kepler architecture supports "dynamic parallelism," enabling the "GPU to dynamically spawn new threads by adapting to the data without going back to the host CPU" [20]. Finally, the architecture allows for multiple CPU cores to utilize a single GPU at the same time [20]. Although the last two features were not utilized in the course of this project, both introduce areas for potential further research.

Because both GPUs utilize the same architecture, many of their OpenCL capabilities are the same. At the time this research was conducted, the NVIDIA OpenCL compiler supported the OpenCL standard only up to version 1.1. The GPUs both have maximum work group sizes of 1024 and can use memory addresses up to 32 bits in length. The final relevant capability similar across the two GPUs is the size of their local memory, 49 kB.

a. Tesla K20c

The Tesla family of GPUs is NVIDIA's current high-performance GPU line designed solely for general-purpose computing on GPUs (GPGPU). In addition to the specifications listed above, the K20c has a clock frequency of 705 MHz, 13 SMs (equivalent to OpenCL compute units) for 2496 CUDA cores [21], and 5 GB of global memory. It also has a memory bandwidth of 208 GB/s [21] and can draw up to 225 W of power.

b. GeForce GTX 650

Unlike the Tesla, the GTX 650 is a standard GPU, designed for video gaming. Nonetheless, it is still powerful for GPGPU applications. The GTX 650 has a clock frequency of 1.058 GHz, approximately 300 MHz faster than the K20c. Many of its other specifications, however, are (as expected) lower than the members of the Tesla family. It contains only 2 SMs for a total of 384 CUDA cores [22] and has only 1 GB of global memory, as well as having a memory bandwidth of only 80 GB/s [22]. It does, on the other hand, require much less power—the GTX 650 draws a maximum of only 64 W [22].

2. Intel Xeon E5-2643

The Xeon family of CPUs is the current line of Intel processors designed for use in servers. The E5-2643 that was used is a four core processor [23] with a clock speed of 3.3 GHz. The Intel compiler supports OpenCL standard version 1.2, and the E5-2643 has 16 compute units and a maximum work group size of 8192. Its local memory size of 32 kB is smaller than the NVIDIA GPUs, which is acceptable as memory transferring is much more expensive for GPUs, even though the E5-2643 has a memory bandwidth of only 51.2 GB/s. Intel lists the E5-2643 as having a cache size of 10 MB [23]. OpenCL lists the E5-2643 as having a global memory size of 33 GB, but this is actually the system RAM. The CPU global memory, therefore, is not physically on the CPU. OpenCL classifies the workstation RAM as part of the host device, which the E5-2643 also serves as in this case. The CPU therefore has access to it. The Xeon line of CPUs can access memory up to 64 bits in length [23]. Finally, the E5-2643 draws a maximum of 130 W [23].

3. Device Comparison

In order to more easily compare the three devices used, several of the important specifications are shown in Table 1. An additional important factor for consideration, although not one that has an impact on performance, is the cost of each processor. The Tesla K20c is far and away the most expensive of the three devices at a cost of $3,000 per processor. The GeForce GTX 650 is only $130, while the Xeon E5-2643 is $885.

Table 1. Device specifications relevant to the conducted research. Note that the listed memory for the Xeon E5-2643 is actually the system RAM.

	Global Memory (GB)	Local Memory (kB)	Max Work Group Size	Memory Bandwidth (GB/s)	Clock Frequency (MHz)
Tesla K20c	5	49	1024	208	705
GeForce GTX 650	1	49	1024	80	1058
Xeon E5-2643	33*	32	8192	51.2	3300

4. FPGA

Originally, an FPGA-based processor from the manufacturer Nallatech (which uses the Altera OpenCL compiler, as previously mentioned) was to be included in the research. The board, however, did not arrive in time for testing to be conducted on it.

C. TESTING ALGORITHMS

In order to quantify the performance of each of the processors, two data processing algorithms were used. Both were chosen on account of their importance in the area of communications. A binary phase-shift keying (BPSK) modulation scheme was chosen because before a signal can be transmitted, it must be modulated. The FFT was chosen because the Fourier transform is one of the first steps of signal analysis.

1. Binary Phase-Shift Keying

a. Motivation

Phase-shift keying (PSK) is a method of digital modulation that operates by changing the phase of a carrier sinusoid according to the symbol being conveyed. In binary phase-shift keying, two symbols can be encoded, where each symbol is a single bit. A block diagram for a basic BPSK modulator is shown in Figure 10, where the middle block represents the modulation of the phase of the carrier signal.

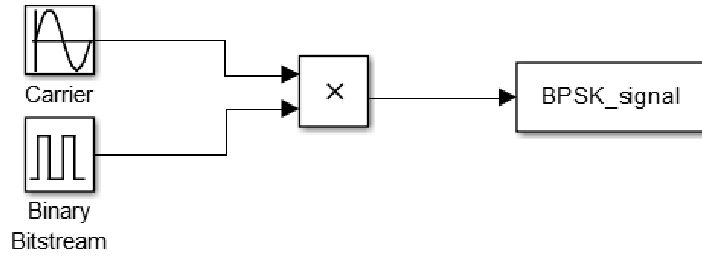


Figure 10. A block diagram of a BPSK modulator.

The mathematical approach for BPSK modulation is relatively simple. Consider the equation

$$x_n(t) = \sqrt{\frac{2E_b}{T_b}} \cos\left[\omega_c \cdot t + \pi(1-n)\right] \tag{8}$$

where $x_n(t)$ is the modulated signal at time t for bit n, E_b is the energy per bit, T_b is the duration of each bit, ω_c is the signal's carrier frequency, and n is a given bit in the bitstream. In the case of Equation (8), a zero in the bitstream shifts the carrier signal phase by 180°, whereas a one does not shift the original phase. An example of a BPSK modulated signal is shown in Figure 11.

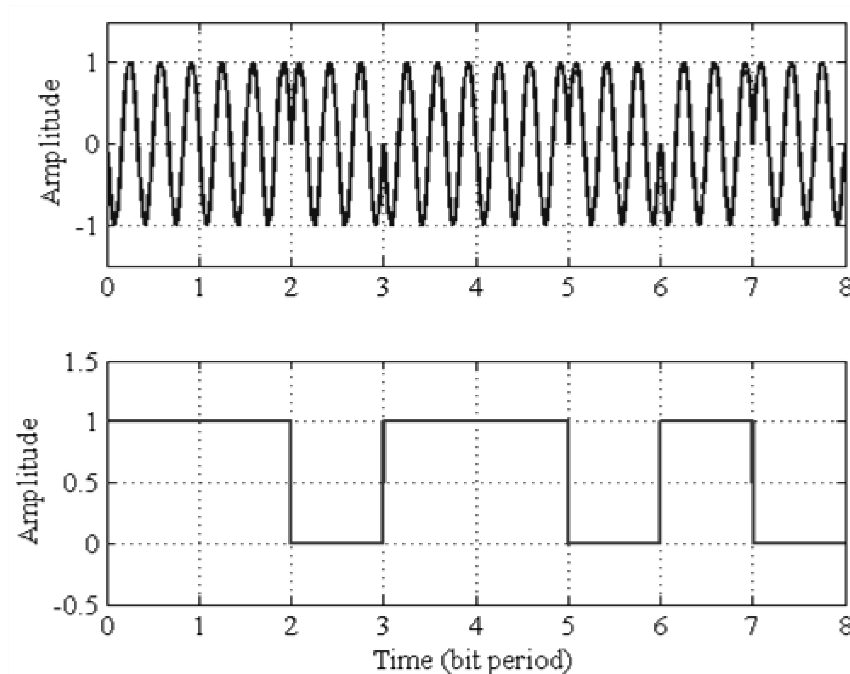


Figure 11. An example of a BPSK modulated signal.

As with other forms of demodulation, BPSK demodulation is done by matching the received signal with the original carrier wave. The aforementioned necessity of synchronicity, however, is shown in Figure 11. The modulated ones and zeroes vary only by their phase. If the incoming signal is not synchronized to the carrier wave, the demodulated bitstream will be inaccurate.

b. OpenCL Algorithm

Although the BPSK modulation algorithm is simple, converting it into parallel code required some consideration as to the values of the modulation parameters in order to achieve the best possible performance. To parallelize the processing, each work group is created to process one symbol. It must be noted that this is not a BPSK modulation symbol of 1 bit but instead is defined as an integer of 32 bits, as per the C programming language, of the transmitting data. In essence, one parallel processing symbol is used to execute 32 BPSK modulation symbols. Each work group is comprised of F_s work elements, where F_s is the number of samples per data bit. The value F_s can vary depending on the need of the granularity of the transmitting sinusoid as given by Equation (8), where time t has sample space $1/F_s$. Each work item then computes the modulation 32 times (one work item per bit of the integer).

The application calls the kernel using the OpenCL command `EnqueueNDRangeKernel`, providing the predetermined values for the number of work groups and the size of each work group. Passed to the kernel is a global memory buffer for the modulated signal, a global buffer for the signal that has already been converted to binary by the host device, the number of samples per bit F_s, the number of bits per symbol, and a global buffer of time steps. A local memory array of time steps is also defined as it can be accessed much faster than the global buffer, which is then only read once for the initialization of the local array. The two time step arrays have the same length $T = F_s$ with the distance between each step $1/F_s$. As a result, each kernel has an unchanging location in the modulated signal for each bit of its group symbol relative to the other groups.

Inside the kernel, and for each bit, a check is performed to determine whether that bit is a zero or a one. The value for a sine wave of length T with the appropriate phase offset (either zero or π radians) is then created for a specific point in time and saved into the global buffer for the modulated signal in the appropriate position. The kernel then moves on to the next bit after incrementing its specific element of the local time array.

c. Demodulation

The demodulation kernel uses slightly different parameters than the modulation kernel in order to operate more efficiently. There are no local buffers required as the kernel only reads from the time step buffer. In addition, the kernel does not require the parameter for the number of bits per symbol. This is because for the demodulation, the number of work items in a work group is equal to that number (i.e., 32). Each work item processes one bit out of the 32 provided. Each work item processes the associated sample points for that bit, which maps from the sinusoid to the digital bit value. To map the sample points from the received sinusoid to the digital bit value, the work item adds each sample point r_i to a corresponding sample point k_i from a sinusoid with known amplitude and phase. If C_i is the sum of the two sample points at time i then

$$C_i = r_i + k_i . \tag{9}$$

For a given bit duration there are F_s number of sums. Hence, to determine of the sinusoid represents a one or a zero, the summation

$$X = \sum_{i=1}^{F_s} |C_i| \tag{10}$$

is compared to a threshold.

2. Fast Fourier Transform

a. Motivation

As has already been mentioned, the Fourier transform is a very important algorithm in signal processing. As the Cooley-Tukey algorithm remains one of the top FFT algorithms, it is a wise algorithm to use as a benchmark for parallel performance.

b. Algorithm

The source code for the FFT algorithm used is primarily from source code provided by M. Scarpino [9, p. 295], which employs the Cooley-Tukey algorithm. Although small changes were made as necessary to the host application, the kernels provided by Scarpino remain unchanged and calculate the FFT of a complex sequence. The sequence is required to be a power of two. If this is not already the case, the sequence is padded with zeroes. This implementation of the FFT employs three different kernels. The first kernel performs four-point FFTs on groups of four elements, merging the calculated values with the ones obtained from the other work items across the same group, using the local memory of the work group. Each group created by the first kernel calculates the FFT on 1024 floats (or 512 doubles). If the length of the signal is greater than these values, the first kernel launches multiple work groups. It then passes the FFT results from these work groups to the second kernel for stitching and merging of the FFT results. The third kernel is called only when the inverse FFT (IFFT) is desired, dividing each value of the output of the final instantiation of the second kernel by the number of points of the sequence.

Launching the FFT kernel from the host device requires more setup than the BPSK kernel requires. Some of this setup is where the differences with the original source code exist. As mentioned previously, the work group size is required to be 1024 work items when using the float data type. This is reduced to 512 when using the double data type. This was discovered after extensive testing, noting that when using the Xeon as the compute device the FFT algorithm was not working in certain cases. The NVIDIA GPUs, however, were. The original algorithm used the maximum work group size of the target device, which for the GPUs was suitable. The local memory for each work group is calculated by obtaining the maximum local memory that a device allows per work group and dividing that maximum by twice the size of the data type used (i.e., four bytes for a float and eight bytes for a double). This local memory is used by the work groups to perform the FFT operation.

The FFT kernels each receive fewer parameters from the host than the BPSK kernel, with each kernel further in the stage requiring fewer parameters than the previous

one. The initial kernel uses a global buffer in which the initial sequence is stored and the final FFT is saved. A local buffer is defined for the work group and values are passed for the number of points per group and the FFT total size and direction (specifying whether the FFT is being calculated or the inverse FFT). The second kernel does not require local memory or the total size of the FFT but does require knowing the stage in the FFT at which the kernel is. The third kernel requires only the global buffer for the data, the number of points per work group, and the scaling value by which to multiply the FFT values.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

In each case, performance was evaluated using the profiling built into OpenCL. Using this requires the device queue be created using the `CL_QUEUE_PROFILING_ENABLE` command. Following this, any function added to the queue can be profiled. This includes each kernel that is launched and the reading of the memory buffers from the compute device back to the host. The initial writing of the buffers, however, cannot be profiled. As a side effect of using the inherent OpenCL profiling, functions that are carried out only on the host device are not evaluated. This is both a positive and a negative. The most important aspect of the algorithm, the kernel(s), gets evaluated while any overhead functions are not. These overhead functions could add a substantial amount of time to the overall performance of the application. However, as this overhead is the same for each of the devices used, and the goal of this research is to directly compare the performance of the three processors, this overhead can be ignored. MATLAB was used to process the resulting data. MATLAB functions were also used to verify that the output of the algorithms was correct.

Due to the parallel nature of the GPUs, it was hypothesized that the two NVIDIA processors would outperform the CPU. In addition, it was assumed that the K20c would outperform the GeForce, as it is designed specifically for GPGPU. The hypothesis, and this assumption, is shown to be inaccurate in some cases.

A. FFT RESULTS

After making the necessary modifications to the host code, tests were done on complex sequences of various lengths to quantify the performance of each device. For each signal length, the FFT algorithm was run on each of the three devices 10 times, with those 10 runs averaged together. An initial run was also conducted as a “warming run” which was not included in the average. It was found that after an unspecified amount of time where a device was not in use (or occasionally when the application was updated and recompiled) the runtime for the algorithm would be much larger than the following time. In one case, for example, the initial runtime for a simulation on the GeForce took

approximately 350 µs, while the average of the following five was approximately 70 µs. This is not a result of memory getting carried over across iterations, as between iterations all memory is cleared. This shows the necessity of a warming run.

1. Performance: Float Data Type

Shown in Figure 12 are the results for the FFT simulation while using the float data type. The number of complex integers input to the FFT is given, in a base two logarithmic scale, on the horizontal axis, while the average runtime for the FFT is given, in microseconds, on the vertical axis.

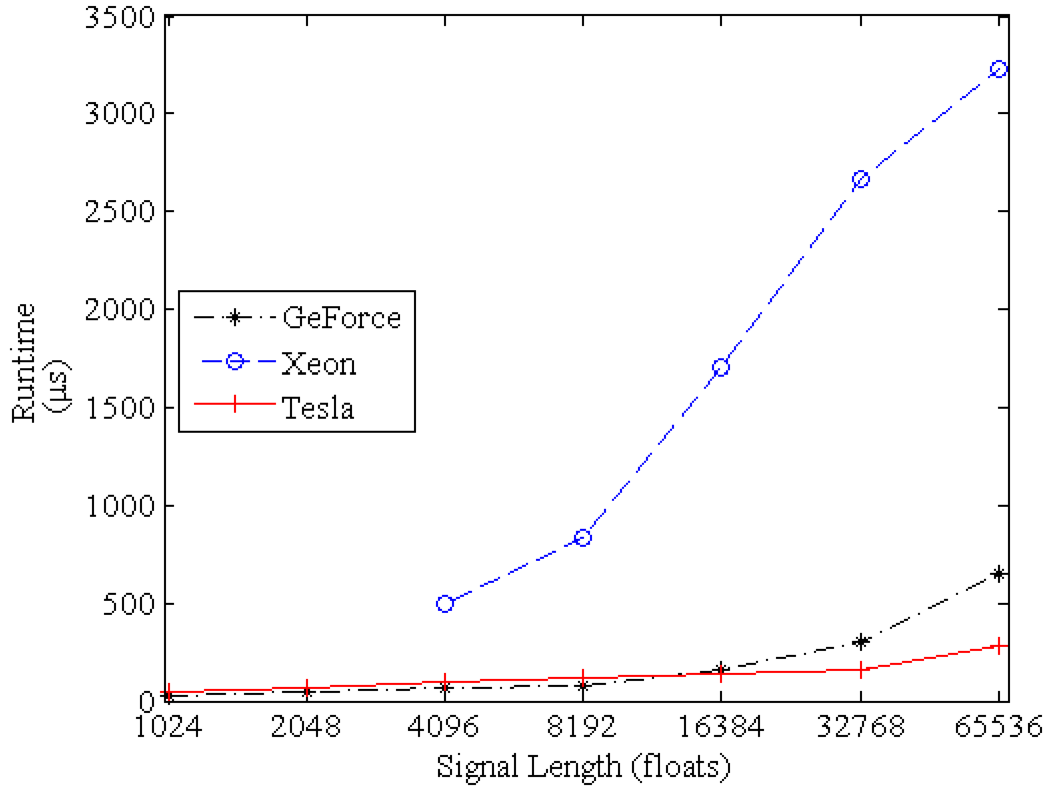


Figure 12. Runtime results of the FFT algorithm on all three devices using the float data type.

As expected, the GPUs outperform the CPU. This is particularly true at higher signal lengths, where the Xeon performance deteriorates much more quickly than the

GPUs. Interestingly, the GeForce outperforms the Tesla at lower signal lengths. At a signal length of 8192, for example, the Tesla takes 116.1 μs to complete the kernel while the GeForce takes only 80.8 μs. The Xeon takes 829.1 μs. The GeForce only begins to perform worse than the Tesla upon reaching the signal length where the algorithm begins to break down (this is discussed in detail below). The results for only the NVIDIA GPUs are shown in Figure 13 to more easily compare their performance. The differences in runtimes for the lower signal lengths, where the GeForce outperforms the Tesla, are small—less than 40 μs.

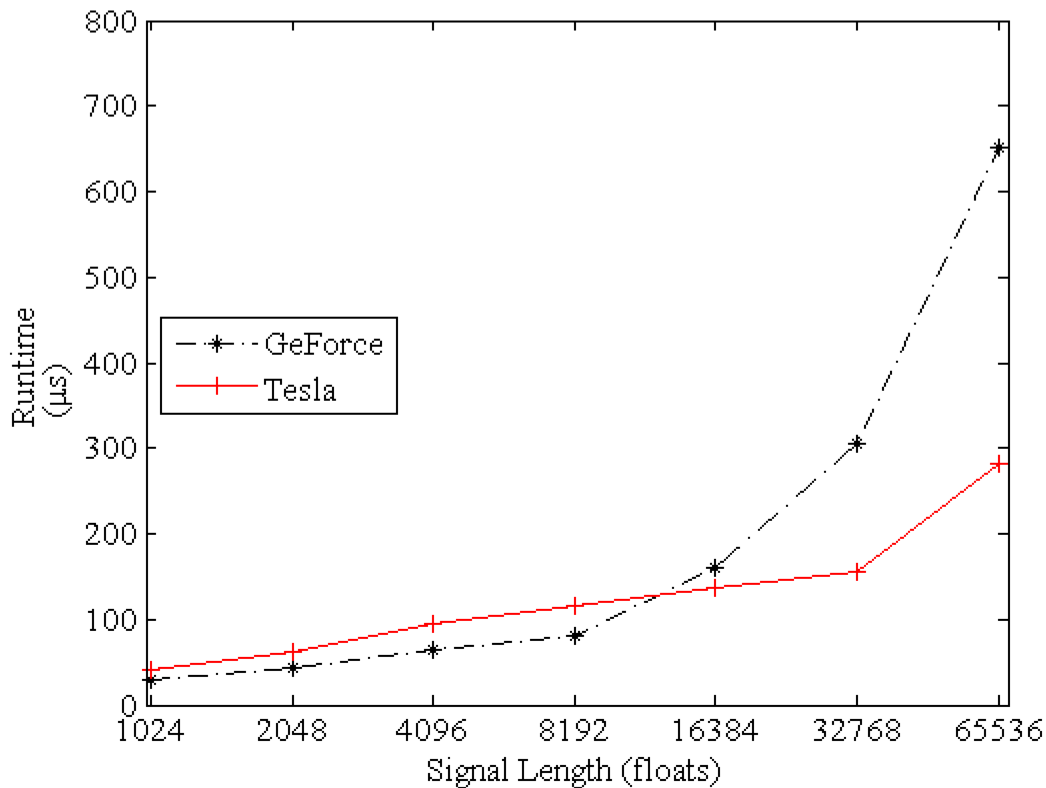


Figure 13. Runtime results of the FFT algorithm on the NVIDIA GPUs using the float data type.

For the Xeon at signal lengths smaller than 4096, the application does not complete, returning a segmentation fault. This is caused by the requirement that the

number of complex floats per work group equal the maximum work group size, which is 8192. Without meeting this strict requirement for the number of complex floats per group, the FFT does operate using the Xeon.

In addition to this problem on the Xeon, the FFT begins to break down at larger signal lengths, as mentioned above. This breakdown occurs in one of three ways. The algorithm, when run on the GPU at higher signal lengths, continues to operate, but the results are inaccurate. At the first length this occurs, a visual inspection of the results reveals a pattern between the output and the correct FFT (as calculated by the CPU and verified using MATLAB). This implies that the output is possibly recoverable. An example of this pattern is shown in Table 2.

Table 2. Output of the FFT algorithm for the Tesla showing the possible recoverability of the incorrect GPU output.

	CPU Output	**GPU Output**
X[0]	33550336.00 + j 0.00	33288192.00 + j 0.00
X[1]	−4095.91 + j 10680706.69	0.00 + j 0.00
X[2]	−4095.84 + j 5340352.46	−8191.47 + j 10680704.27
X[3]	−4096.00 + j 3560234.42	0.00 + j 0.00
X[4]	−4095.92 + j 2670174.68	−8191.74 + j 5340349.04
X[5]	−4095.80 + j 2136138.77	0.00 + j 0.00
X[6]	−4095.96 + j 1780114.92	−8192.25 + j 3560229.67
X[7]	−4095.87 + j 1525810.97	0.00 + j 0.00
X[8]	−4095.96 + j 1335084.20	−8191.87 + j 2670168.24

As can be seen, the GPU produced value for $X[0]$ is accurate. The value for each odd numbered point is equal to zero and incorrect. Then, for each even point, the real portion is equal to twice the value it should be, while the complex portion is equal to the correct value from the point $X[N/2]$. The GeForce is the first to break down in this manner, doing so at signals of length 16384. The Tesla does so at signals of length 32768. For signal lengths greater than these, the results are completely unrecoverable. For the GeForce, this is for 32768 floats, and for the Tesla, this is for 65536. Finally, at signals of length 65536, the Xeon outputs the correct FFT approximately 50 percent of the time, while the rest of the outputs are unrecoverable numbers. The pattern of the breakdowns is more easily seen in Table 3.

Table 3. A comparison of the signal lengths at which the FFT algorithm breaks down on the three devices with the float data type.

	Signal Length						
Device	1024	2048	4096	8192	16384	32768	65536
Xeon	SEGFAULT		Correct operation				50% correct output
GeForce	Correct operation				Possible fix	Unusable output	
Tesla	Correct operation					Possible fix	Unusable output

The cause of the high signal length breakdowns was discovered to be a cause of the second kernel in the algorithm. This is the case for both the CPU and two GPUs. This was determined by comparing the data output from the first kernel. The intermediate data output from the first kernel in the cases where the final FFT output was correct was equal to the intermediate data when the final output was incorrect.

2. Performance: Double Data Type

As a comparison to the algorithm operation using the float data type shown in Figure 13, the results for the FFT algorithm run using the double data type are shown in Figure 14. Doubles use twice the amount of memory as floats, and it is expected, therefore, that the algorithm functions only on smaller signals. This is the case, although the breakdowns in algorithm operation did not occur in a similar fashion. This is shown in Table 4. The GeForce only output useable results for signals of length 1024 and 2048. The Tesla produced similar results to the GeForce but, interestingly, for a signal of length 16384, reliably produced accurate output values. The Xeon performed the same as when using the float data type, where it did not operate at the low signal lengths (although it did operate for signals of length 2048 in this case due to the size of the double data type being twice that of the float) and produced accurate FFT results at the highest signal length (32768 in this case) 50 percent of the time. The runtimes for the double data type were also slower, in general, than the runtimes for the float data type. At a signal length of 8192, in comparison to the previous results, the Tesla took 117.6 μs, the GeForce 144.5 μs, and the Xeon 956.9 μs.

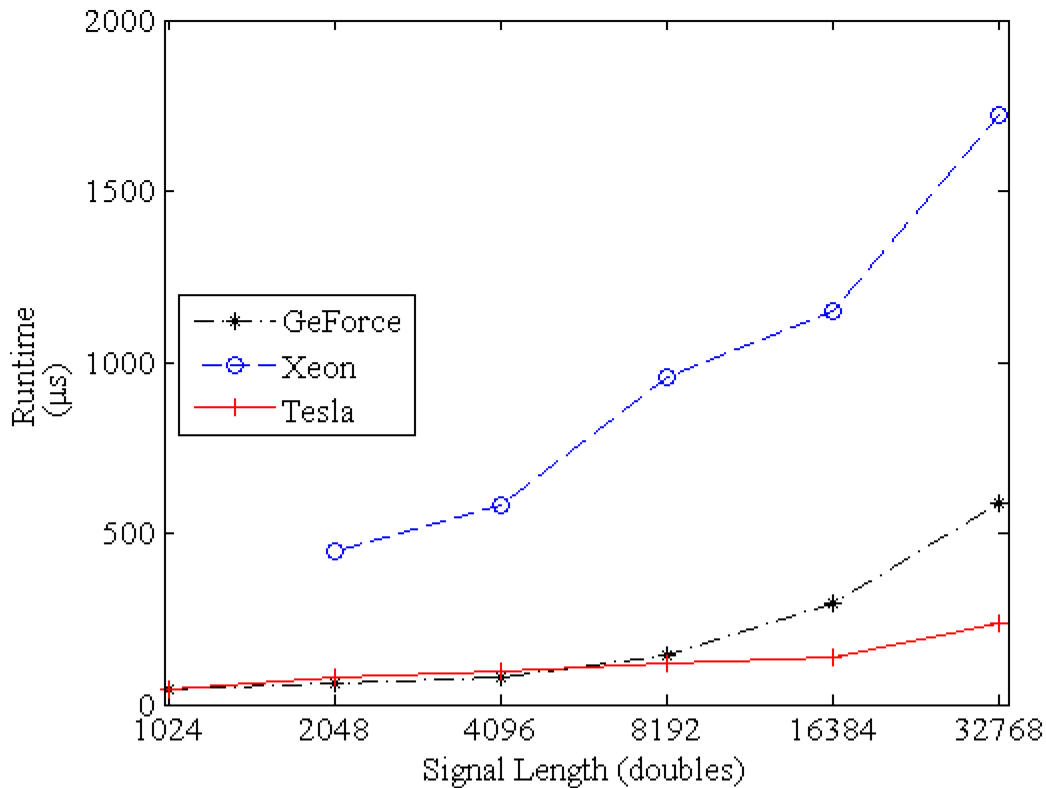


Figure 14. Runtime results of the FFT algorithm on all three devices using the double data type.

Table 4. A comparison of the signal lengths at which the FFT algorithm breaks down on the three devices with the double data type.

	Signal Length					
Device	1024	2048	4096	8192	16384	32768
Xeon	SEGFAULT	Correct operation				50% correct output
GeForce	Correct operation		Possible fix		Unusable output	
Tesla	Correct operation		Possible fix		Correct operation	Unusable output

3. Algorithm Output

When it is actually correct, the output of the FFT algorithm is correct to a high degree of accuracy. An example of this accuracy is shown in Figure 15 where the

(truncated) output of the OpenCL algorithm is compared to the (truncated) output of the FFT function built into MATLAB. The sequence passed through the FFT algorithms is the function

$$rect[n] = \begin{cases} 1, & |n| < \dfrac{N}{4} \\ 0, & otherwise \end{cases} \tag{11}$$

where in this case $N = 4096$. The top plot $X_M(\omega)$ is the output of the MATLAB function, while the bottom plot $X_O(\omega)$ is the output of the OpenCL algorithm. As can be seen, the output of both algorithms is visually the same. This visual similarity is the case regardless of the device used for the OpenCL computation.

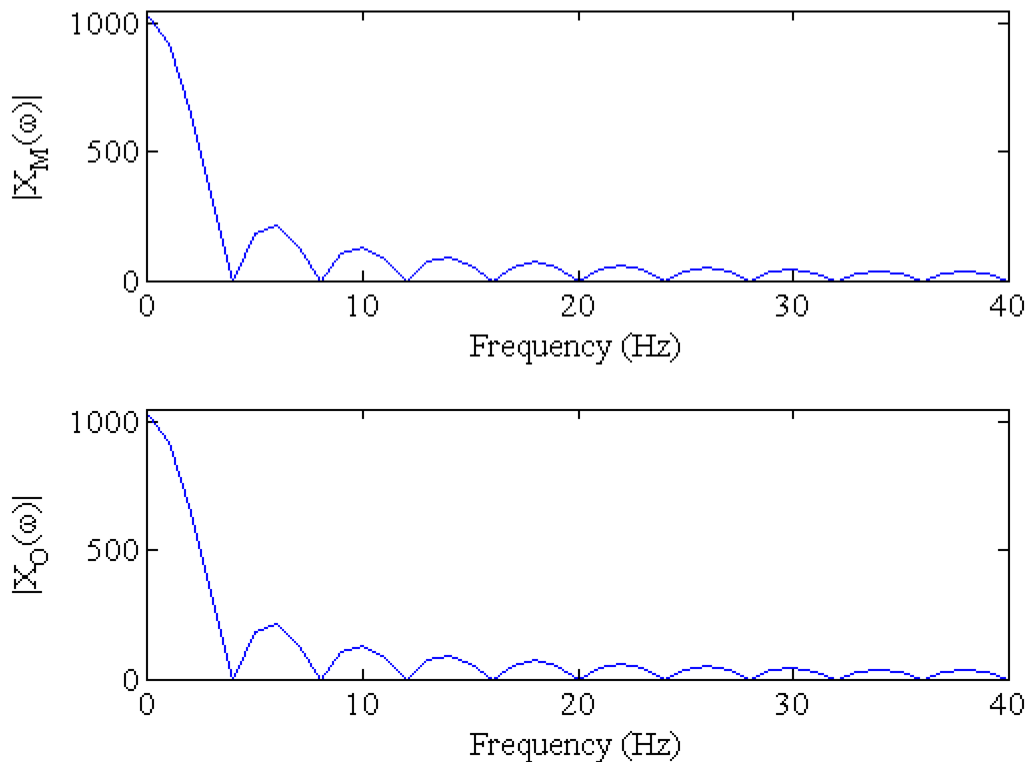


Figure 15. Comparison of the FFT output for a signal $rect[n]$. The output of the MATLAB FFT is shown by $X_M(\omega)$ and the output of the OpenCL algorithm is shown by $X_O(\omega)$.

An additional illustration of the accuracy of the OpenCL FFT algorithm is shown in Table 5. The root mean-square error (RMSE) of the output of the Xeon and the NVIDIA GPUs was calculated with respect to the output of the MATLAB function. Here a ramp function of length 16384 is used and the error calculated in the case of both the float and double data types. The Xeon has less error in each case. It should be noted that the two GPUs output the same numbers and, therefore, have the same RMSE values. In addition, when the double data type is used, the RMSE is slightly less, although the error in output of both the GPUs and CPU is negligible in each case.

Table 5. RMSE values for the FFT algorithm using a signal length of 16384. The algorithm does not operate on the GeForce for a length 16384 signal using the double data type.

Data Type	Device	RMSE
Float	GeForce/Tesla	0.1063
	Xeon	0.0858
Double	Tesla	0.0819
	Xeon	0.0761

B. BPSK PERFORMANCE

For the evaluation of the three devices using the BPSK algorithm, various tests were run, alternating between keeping the number of symbols (integers) input to the algorithm constant and keeping the number of samples per bit (F_s) constant. As in the evaluation of the FFT algorithm, the simulation is run with a warming run preceding the runs averaged together to calculate the performance.

1. Initial Simulation

Shown in Figure 16 is the first test run where F_s is held constant (at 100 samples per bit) while the number of symbols increases linearly. Five runs were averaged together in this case. The results produced here are different than the results from the FFT

algorithm. In this case, as is the case for all the BPSK simulations, the GeForce takes the longest to run the algorithm. As originally hypothesized, the Tesla performs the quickest. The runtimes of all three devices increase linearly as the number of symbols increases linearly, albeit with different slopes. One important consideration for the results of the BPSK algorithm is the scale of the vertical axis. Here, the scale of the vertical axis is in milliseconds, while the FFT algorithm produced runtimes on the order of microseconds.

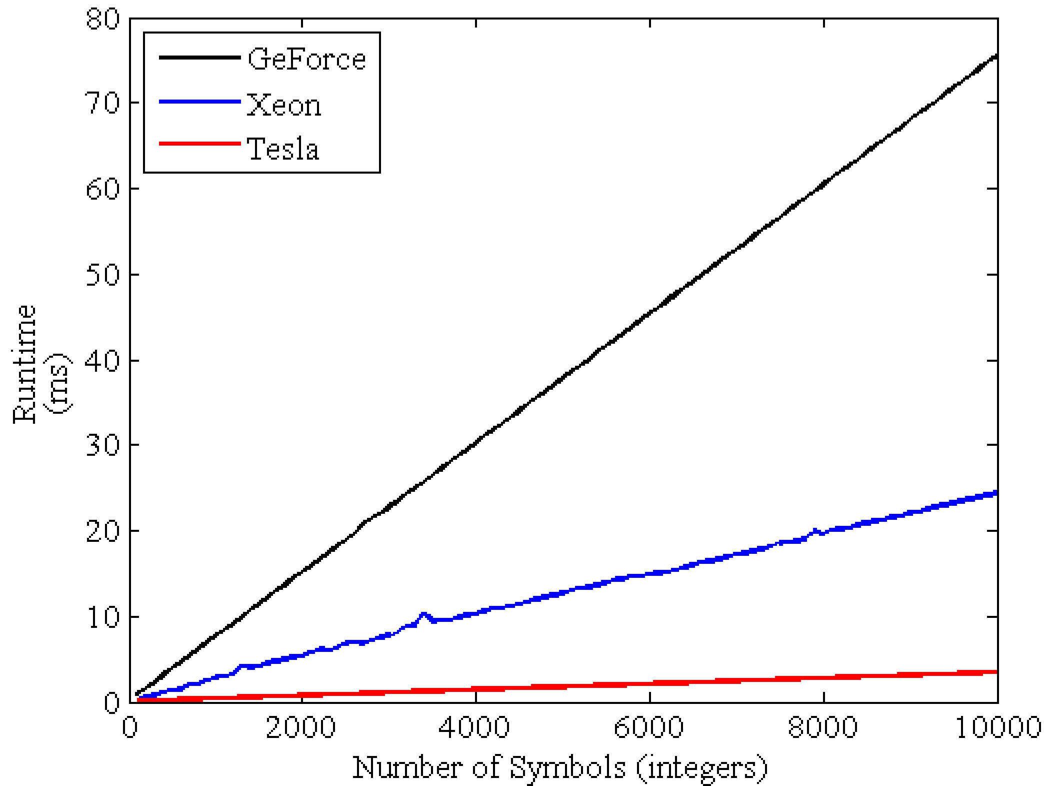


Figure 16. Performance results for the BPSK algorithm for all three devices. The number of samples per bit is held constant at 100.

Shown in Figure 17 are the results from the second test, where the number of symbols was held constant at 1000 and F_s allowed to grow linearly. As in the previous test, the GeForce performed the worst and the Tesla the best. In this case, however, the increase in runtime is not linear. The performance of the Tesla appears constant, while the performance of the GeForce is constant with the exception of large jumps every time

F_s reaches another multiple of 32. The significance of this arises when the maximum work group size of the GeForce (and Tesla) is considered. First, recall that the input to the kernel for desired work group size is F_s and that for both NVIDIA devices, the maximum number of work items per work group is 1024. If 1024 is divided by F_s (i.e., 32), the result is 32, which is the value for the number of bits per symbol. Therefore, it can be deduced that every time F_s iterates past a multiple of 32, a new work group is required. This should be the case for the Tesla because its properties are the same, and in fact it is, as shown in Figure 18. This cannot be seen in Figure 17 as the scale for the vertical axis is too large. This result is interesting and can likely be utilized by the wise programmer.

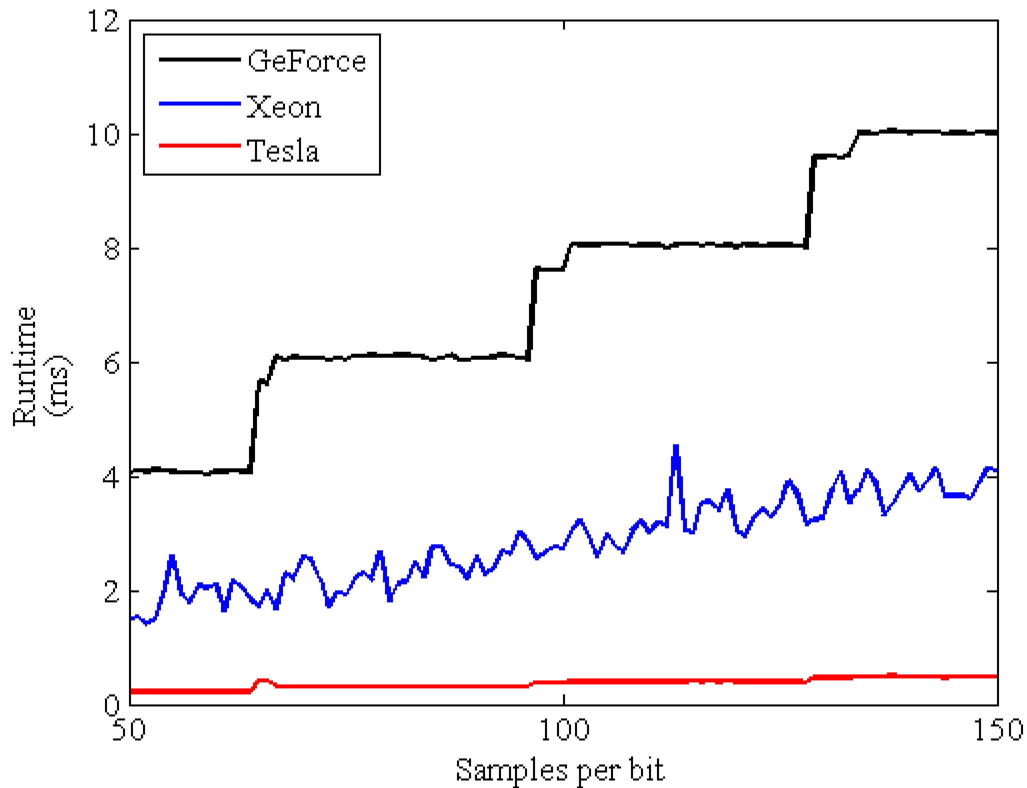


Figure 17. Performance results for the BPSK algorithm for the three devices. The number of symbols is held constant at 1000.

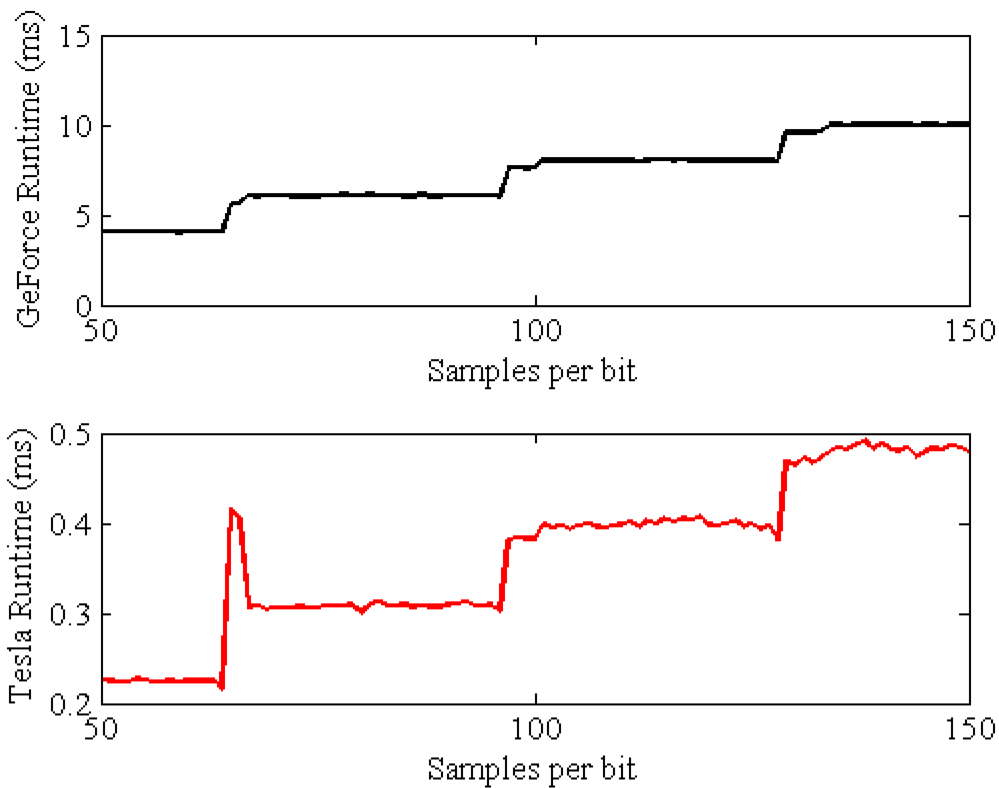


Figure 18. Performance results of the BPSK algorithm for the NVIDIA devices with a fixed number of symbols (1000).

If one attributes the small scale variations for the performance of the Xeon to a low number of samples averaged together (as only five runs were used), then the runtime appears to be slowly increasing linearly. However, one would also assume that, based on the performance of the GPUs, the runtime should increase greatly when the product of F_s and the number of bits per symbol reaches the maximum work group size. For the Xeon, this maximum value is 8192. In Figure 19, this is shown to be the case.

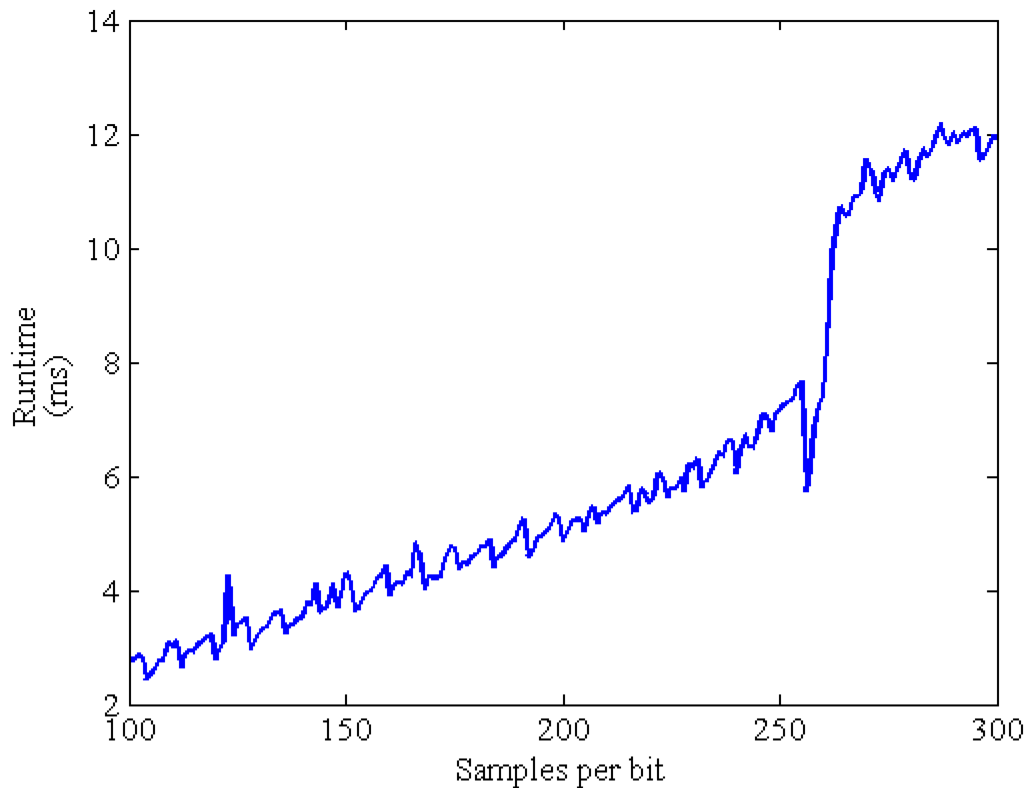


Figure 19. Performance of the BPSK algorithm on the Xeon to show similarities with the NVIDIA results. A fixed number of symbols (1000) is used.

An additional important result of the algorithm that should be considered is its output. When the float data type is used with 100 samples per bit and 512 symbols, the algorithm output totals 1.0 MB in size. If the double data type is used, the size of the output is twice that. There are differences between the values of the output numbers when the two data types are compared, but, as is the case with the FFT algorithm, the differences are small.

2. Additional Simulations

After the initial simulations, two changes were made to the host-side processing of the algorithm, the results of which are shown in Figure 20. It was first determined that using 100 samples per bit was an excessive waste of processing power. This number was decreased to 16, which in addition to being much smaller (and, therefore, creating less data) had the added benefit of being a power of two. This correlates to creating signal

lengths for a carrier wave that are also powers of two. In addition, it was decided to increase the number of runs averaged together to ten, instead of five, in order to better quantify the performance. It should also be noted that now the scale of the vertical axis is microseconds.

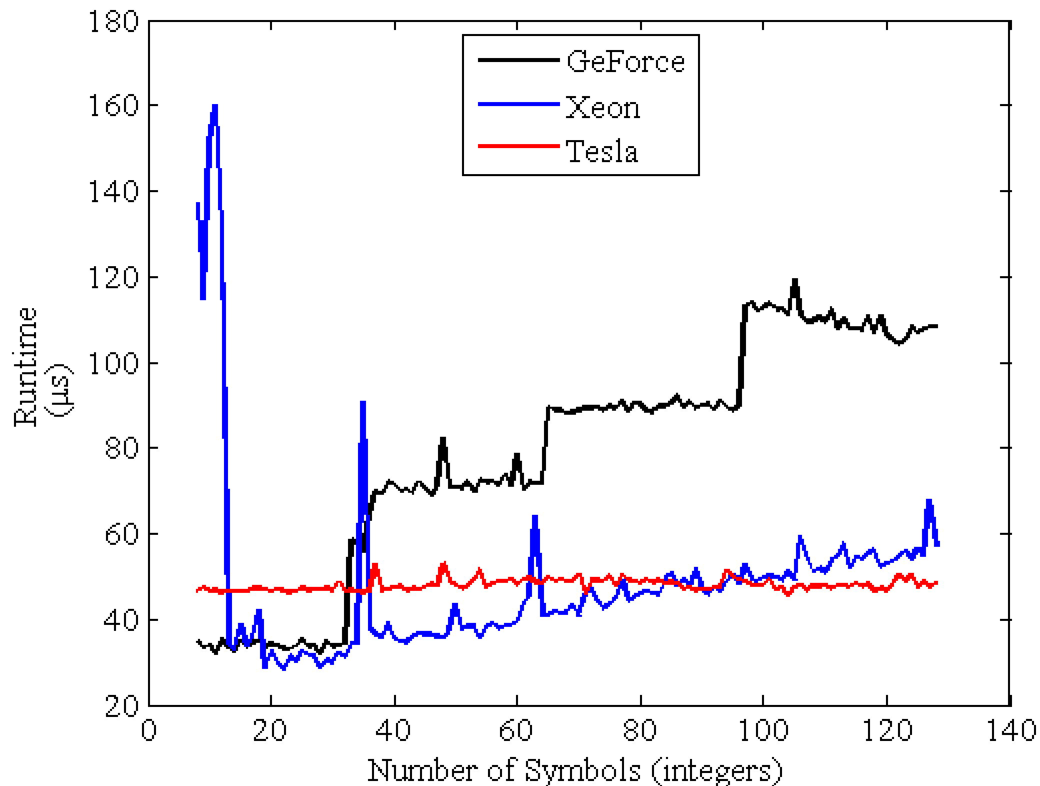


Figure 20. Performance of the BPSK algorithm on all three devices with 16 samples per bit and a variable number of symbols. The plotted performance is the average of 10 runs.

The decrease in the number of samples per bit greatly decreases the runtime for the kernel. This is expected, as decreasing the number of samples decreases the amount of numbers the device needs to process. An interesting side effect of decreasing the quantity of numbers is shown by the plot of the Xeon performance. Unlike the GPUs, because the CPU has many fewer cores to perform calculations on, it is required to perform context switching. This is the act of changing the data a given core is working on. This is mitigated by proper scheduling. With fewer consecutive numbers to operate

on, however, context switching begins to play a much larger role in the total execution time for the processor. This has been documented previously by Lee et. al [24], and it can be seen that as the number of symbols being modulated increases, the runtime returns to its expected linearly increasing behavior. Both the GeForce and the CPU, however, for a small region of the test, perform quicker than the Tesla. The behavior of the Tesla is similar to the previous simulation.

3. Demodulation

The demodulation kernel was created separately from the modulation kernel. As such, it was necessary to profile it separately. Shown in Figure 21 are the results from the initial evaluation of the kernel's performance. The vertical axis shows the average runtime of the kernel in microseconds, while the horizontal axis shows the number of integers resultant from the demodulation of the processed signal. The number of samples per bit was again 16.

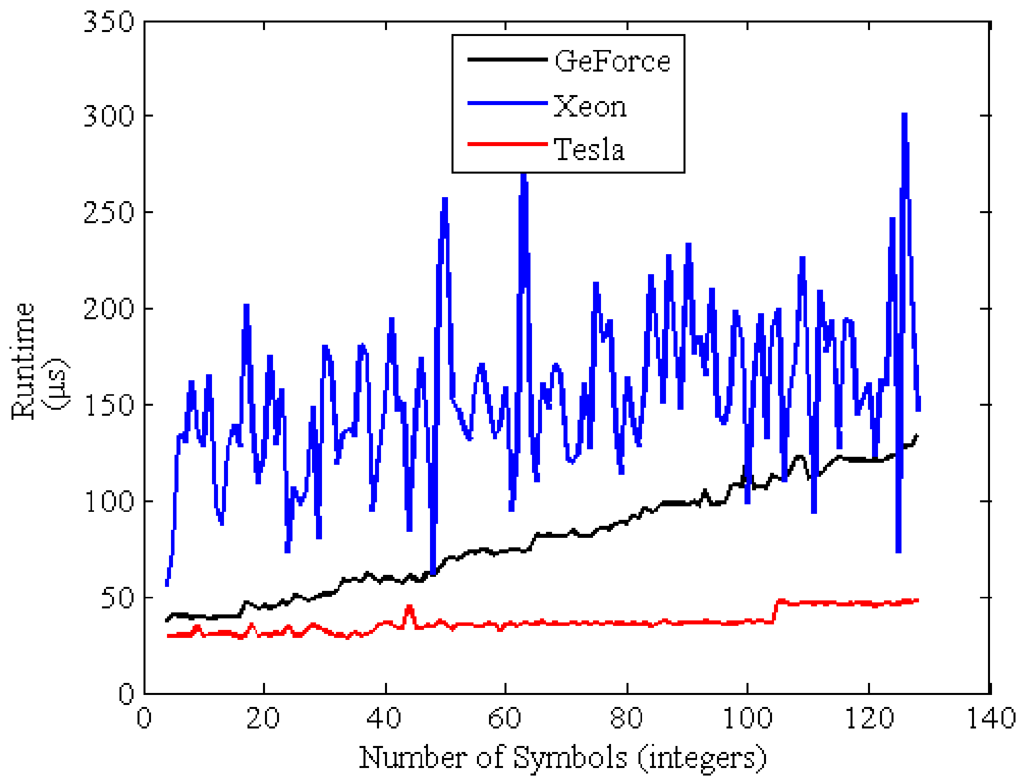


Figure 21. Performance of the demodulation kernel on all three devices. The horizontal axis is the number of integers resultant from the

demodulated signal.

The general performance results for the demodulation kernel are similar to the modulation kernel. The GeForce performs the worst and the Tesla the best. The performance of the two GPUs is not the same as it was for the modulation kernel, however. The runtime of the GeForce increases linearly while the Tesla is constant, excepting a jump at 105 symbols. This jump is not mirrored by the performance of the GeForce. In addition, Xeon performance is extremely erratic. The mean of the performance appears to remain relatively constant. This erratic behavior was repeated after an additional simulation with the same parameters. For this reason, it was decided to increase the number of runs averaged together to 15, the results of which are shown in Figure 22. The length of the simulation was also extended to 256 symbols.

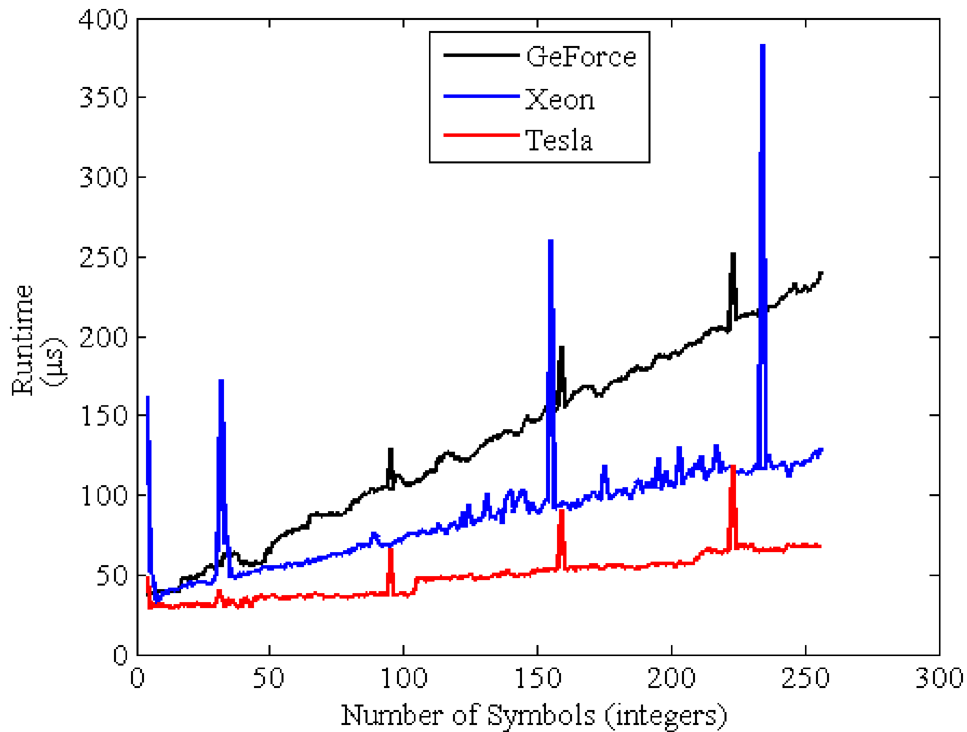


Figure 22. Performance of the demodulation kernel on all three devices. The horizontal axis is the number of integers resultant from the demodulated signal. The plotted performance is the average of 15 runs.

Although the plot of the Xeon performance has smoothed out and lost its erratic nature, several curious behaviors have emerged. One is the fact that the runtime of the Xeon, which was previously higher than the GeForce, has dropped below it. The cause of the erratic behavior in the previous plot, then, is truly anomalous. The second curiosity is present in each of the three plots. As the plots are the average of 15 runs, it can be said with a strong degree of certainty that the spikes present in the plots are not anomalous. For the GPUs, this is enforced by the fact that they occur at regular intervals and further enforced by the fact that the spikes in both the Tesla and GeForce plots occur at the same signal lengths. This suggests that the spikes are a result of some value related to the Kepler architecture. No relation between the spikes occurrence every 64 symbols and an OpenCL parameter, however, can be found; as such the cause of the runtime spikes is unknown. The general trend of the performance of the NVIDIA processors is otherwise the same, with the runtime of the GeForce increasing linearly and the runtime of the Tesla remaining constant except at specific increases. The spikes in the performance of the Xeon, on the other hand, do not occur at consistent intervals.

As a result of the spikes in runtime, it was decided to rerun this simulation. The results of the simulation are shown in Figure 23. Here, the length of the simulation was kept at 256 symbols, but the number of runs averaged together dropped back down to ten. The initial Xeon performance mirrors that of Figure 21, with erratic performance but a constant mean. It drops, however, and begins to mirror the performance of Figure 22 at around 52 symbols. This higher early performance could again show an example of the context switching problem present when utilizing the CPU. Additionally, the spikes in performance remain but are now in different positions than the previous simulation. The locations of the spikes in the runtimes of the two NVIDIA devices are once again in the same relative position but now have different intervals (93 symbols) between them. This suggests that the spikes are the result of a runtime factor rather than a compilation factor and not related to OpenCL directly.

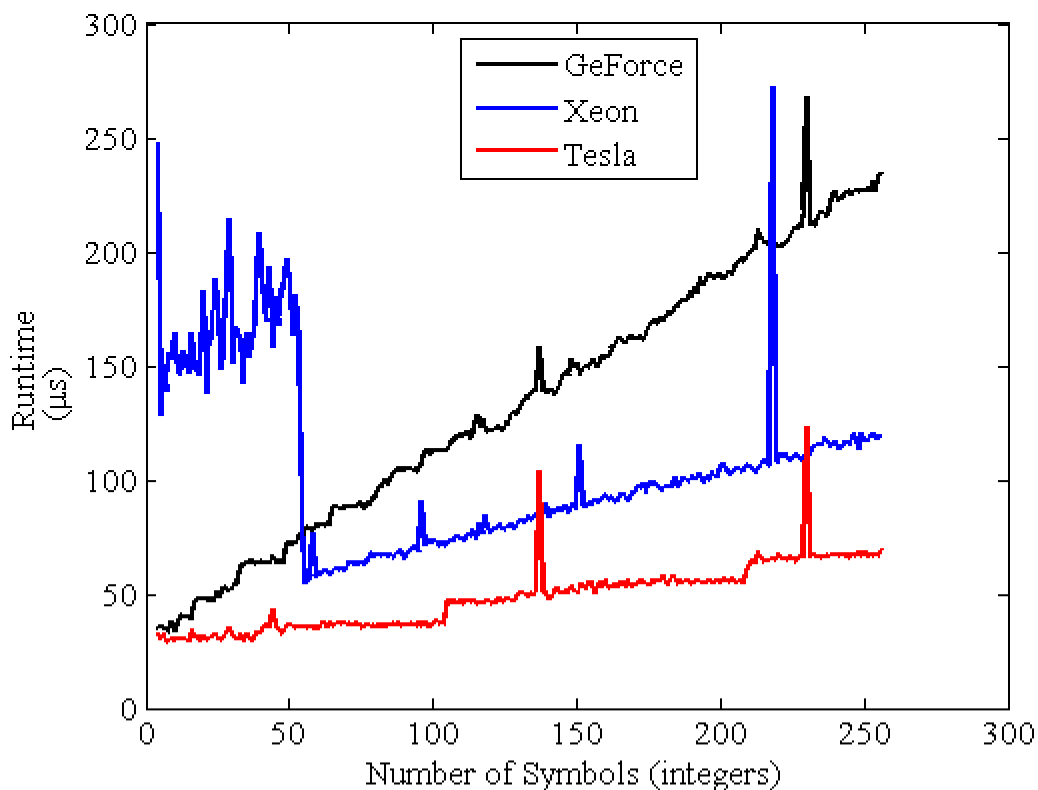


Figure 23. Performance of the demodulation kernel on all three devices. The horizontal axis is the number of integers resultant from the demodulated signal. The plotted performance is the average of 10 runs.

C. COMBINED ALGORITHM PERFORMANCE

After creating and testing both algorithms, the two applications were combined into a single, larger application. The order in which the signal is passed through each kernel is shown in Figure 24. Due to the limits placed on the size of the input to the FFT kernel, the testing of the combined application was limited to test sequences that were fairly small. The size of the input N_x to the FFT kernel is

$$N_x = F_S \cdot N_S \cdot N_B \tag{12}$$

where $F_S = 16$ is the predetermined number of samples per bit, $N_B = 32$ is the number of bits per symbol and N_S is the number of symbols input to the modulation block. For only two modulated symbols, the length of the sequence input to the FFT algorithm is 1024.

This is not a large number of symbols, but if just 32 symbols are used, the FFT breaks down on the GeForce.

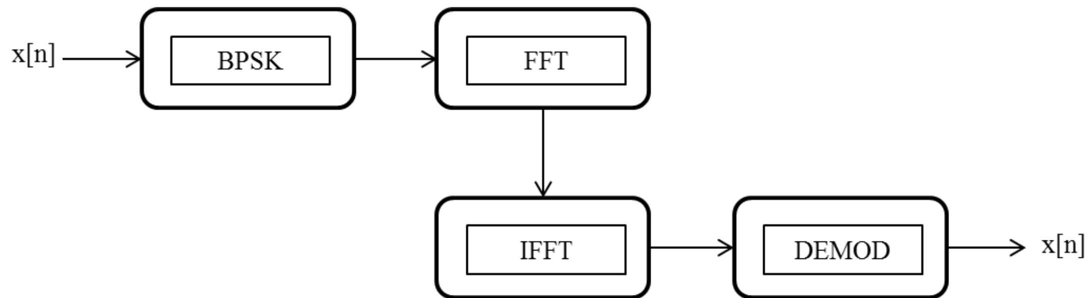


Figure 24. Flow of the combined FFT and BPSK algorithms.

The performance of the combined application on the three devices is shown in Figure 25. Only the float data type was evaluated. In general, the results mirror the results of the FFT algorithm at higher runtimes.

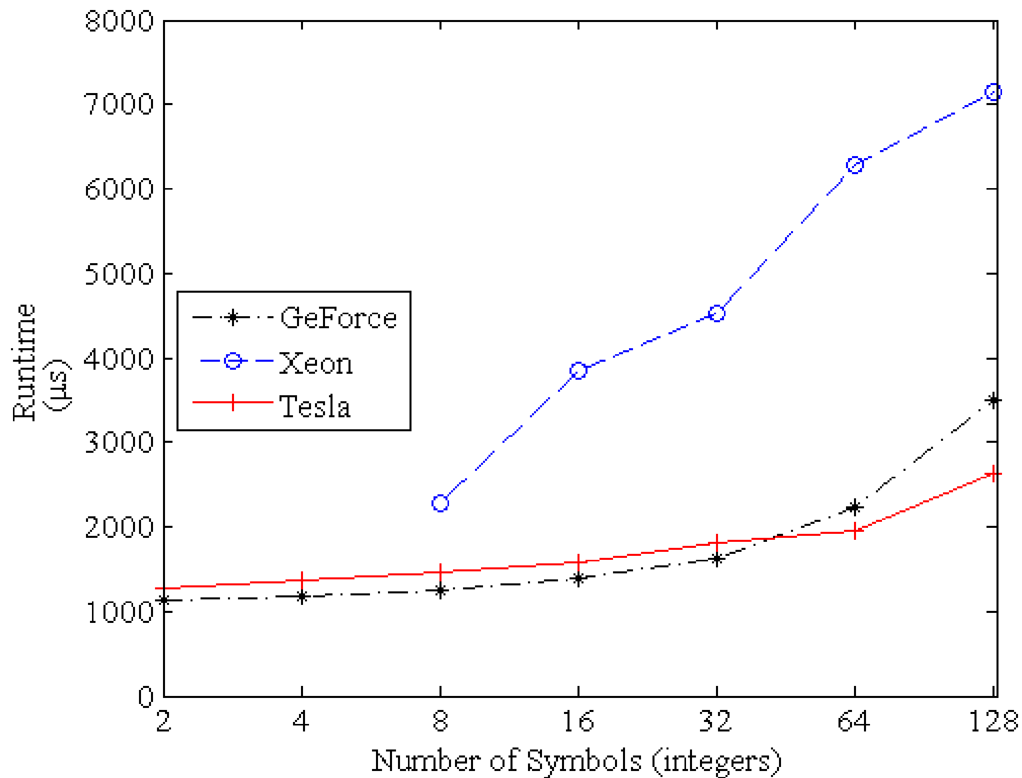


Figure 25. Performance of the combined algorithm on all three devices using the average of 10 runs.

In this case, again, the best performance for the sequences of smaller length is seen by the GeForce. The gap between the performance of the GeForce and Tesla, however, is much smaller in this case. This is likely a result of the much slower performance of the GeForce during the modulation and demodulation kernels. It was also decided to test the performance without the modulation and FFT kernels, leaving just the IFFT and demodulation kernels to perform on the data. It should be noted that the first two kernels were still performed in order to generate the data but only the last two were profiled. The results for this simulation are shown in Figure 26. It is expected that as these two kernels are half of the total application, they should take roughly half as long as the results shown in Figure 26. This is the case.

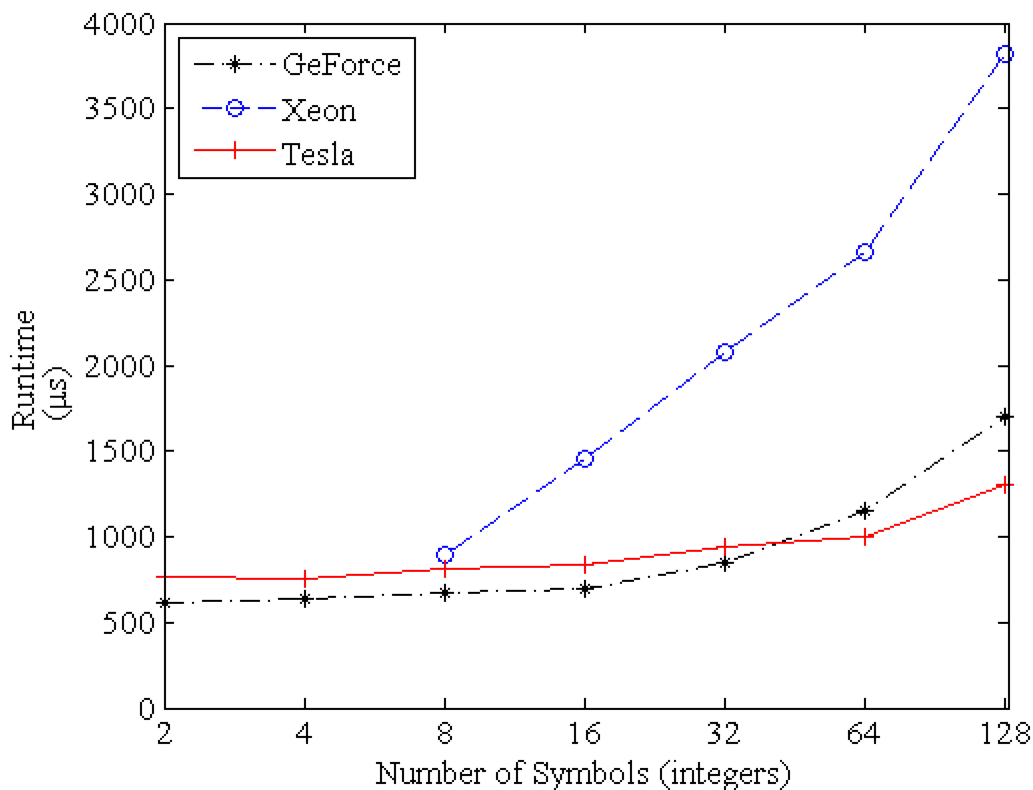


Figure 26. Performance of the IFFT and demodulation blocks of the combined algorithm on all three devices.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION AND RECOMMENDATIONS

A. FINDINGS AND CONCLUSIONS

One important difference between the BPSK algorithm and the FFT algorithm that has an effect on the performance of the algorithms is the amount of input data to each algorithm. For the FFT algorithm there is one complex number—two numbers of the data type in use—produced, whereas for the BPSK algorithm F_s numbers are produced for every bit of every number sent. When $F_s = 50$ and 1024 integers are modulated, the output is 34 MB in size. This is huge and is the main reason why the NVIDIA Tesla outperforms the other two processors on the BPSK algorithm. It was designed for processing large amounts of data. Even though the GeForce finishes executing the FFT algorithm sooner than the Tesla with smaller signal lengths, it has a lower ceiling for the size of the signal length. The Xeon, on the other hand, performs inconsistently approximately 50 percent of the time. This is not a feature that is desirable in a data processing system. The Xeon's performance is hampered by its inherently serial nature and is further hampered by the fact that, in addition to executing the kernel, it is required to act as the host device of the application, executing the potentially high overhead associated therewith.

Based on this research, the Tesla is the best processor out of the three provided. Although not the fastest processor in every case, its consistent, high performance processing make it the optimal processor. This, of course, does not take into consideration the cost of the processor. If cost is an important factor, the GeForce is likely the device that would be considered best. This is due to the fact that the cost of the GeForce is an order of magnitude less than the cost of the Tesla. This decision also does not consider the power consumption, which could also be a strong deciding factor in an eventual selection.

This research explored the performance of three specific devices for two specific algorithms. This conclusion, therefore, may not be applicable for every situation. This is reinforced by the highly circumstantial results of the Xeon on a single algorithm and the GeForce across the two algorithms.

B. RECOMMENDATIONS FOR FUTURE WORK

1. FPGA

As no FPGA-based accelerator was available for testing, it is highly recommended that the two separate algorithms, and the combined application, be tested on one. Nallatech numerical accelerator boards use Altera FPGAs and the Altera OpenCL FPGA compiler to run numerically intensive applications. Including FPGAs as a potential hardware device is particularly necessary if the end goal for the selected hardware is in a situation where minimal power consumption is vital.

2. Additional Algorithms

It is recognizable from the results that the specific algorithm used has a high impact on performance. Because the GeForce performs well when executing the FFT algorithm but has the slowest execution for the BPSK algorithm, it is obvious that any specific algorithm to be used must be fully developed prior to device selection. Of course, there is little evidence to suggest that the consistently high performance of the Tesla would not also be extended to other algorithms.

3. Utilizing Multiple Devices

Utilizing multiple devices in parallel is also a possibility for further work, depending on the algorithm used. Although the same data cannot be operated on in two different locations at the same time, in certain situations it may be possible to further parallelize an application.

4. Recovery of the FFT Output

As mentioned in the discussion on the FFT algorithm, the results for the GPU for larger signal lengths are incorrect but possibly recoverable. It was determined that the

cause of each breakdown in the algorithm is caused by the second kernel of the algorithm. Further work can be done to find a solution. Possible solutions include the examination of, and revision of, the second kernel or the creation of an entirely new FFT algorithm.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Task Force ASW. (2004). Anti-submarine warfare concept of operations for the 21st century. [Online]. Available: http://www.navy.mil/navydata/policy/asw/asw-conops.pdf

[2] S. I. Erwin. (2005, March). Shrewd tactics underpin Navy strategy to defeat diesel submarines. [Online]. Available: http://www.nationaldefensemagazine.org/archive/2005/March/Pages/UF-Shrewd_Tactics5844.aspx

[3] Y. Zhang, M. Sinclair II, and A. A. Chien,"Improving Performance Portability in OpenCL Programs," in *28th Int. Supercomputing Conf.*, Leipzig, Germany, 2013, pp. 136–150.

[4] Q. Lan, C. Xun, M. Wen, H. Su, L. Liu, and C. Zhang, "Improving Performance of GPU Specific OpenCL Program on CPUs," in *13th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies*, Beijing, China, 2012, pp. 356–360.

[5] J. Preshing. (2012, February). A look back at single-threaded CPU performance. [Online]. Available: http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance

[6] K. Dixit, J. Reilly and J. Henning. (2011, September). SPEC CPU2006: Read me first. [Online]. Available: http://www.spec.org/cpu2006/Docs/readme1st.html

[7] J. Palacios and J. Triska. (2011, March). A comparison of modern GPU and CPU architectures: And the common convergence of both. [Online]. Available: http://web.engr.oregonstate.edu/~palacijo/cs570/final.pdf

[8] D. Curd. (2007, February). Power consumption in 65 nm FPGAs. [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp246.pdf

[9] M. Scarpino, *OpenCL in Action*. Shelter Island: Manning, 2012.

[10] Intel Corporation. (2014). Intel Atom processor for smartphone and tablet. [Online]. Available: http://ark.intel.com/products/family/70095/Intel-Atom-Processor-for-Smartphone-and-Tablet

[11] Intel Corporation. (2014). Intel Xeon Phi coprocessor 7100 series. [Online]. Available: http://ark.intel.com/products/series/75809

[12] NVIDIA Corporation. (2013, October). NVIDIA Tesla GPU accelerators. [Online]. Available: http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf

[13] J. Sanders and E. Kandrot, “Why CUDA? Why now?” in *CUDA by Example*, Boston: Addison Wesley, 2011, pp. 4–7.

[14] N. Trevett. (2012, November). Khronos overview. [Online]. Available: http://www.khronos.org/assets/uploads/developers/library/overview/khronos_overview.pdf

[15] N. Trevett. (2013). *OpenCL Introduction*. [Online]. Available: https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf

[16] B. P. Lathi and Z. Ding, *Modern Digial and Analog Communication Systems*, 4th ed. New York: Oxford Univ. Press, 2009.

[17] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” in *Math. of Comp. 19.*, 1965, pp. 297–301.

[18] Y. Li, Y. Zhang, H. Jia, G. Long, and K. Wang, “Automatic FFT performance tuning on openCL GPUs,” in *17th Int. Conf. on Parallel and Distributed Systems*, Tainan, Taiwan, 2011, pp. 288–235.

[19] Apple Inc. (2013). OpenCL programming guide for Mac. [Online]. Available: https://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCL_MacProgGuide.pdf

[20] NVIDIA Corporation. (2012, May). NVIDIA Kepler GK110 next-generation CUDA compute architecture. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf

[21] NVIDIA Corporation. (2013, October). NVIDIA Tesla GPU accelerators. [Online]. Available: http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf

[22] NVIDIA Corporation. (2014). Specifications: Geforce [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-650/specifications

[23] Intel Corporation. (2014). ARK: Intel Xeon processor E5-2643. [Online]. Available: http://ark.intel.com/products/64587/Intel-Xeon-Processor-E5-2643-10M-Cache-3_30-GHz-8_00-GTs-Intel-QPI

[24] J. H. Lee, K. Patel, N. Nigania, H. Kim, and H. Kim, “OpenCL Performance Evaluation on Modern Multi Core CPUs,” in *IEEE 27th Int. Symp. on Parallel & Distributed Processing Workshops and PhD Forum,* Cambridge, MA, 2013, pp. 1177–1185.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California