2013

On Performance of GPU and DSP Architectures for Computationally Intensive Applications

John Faella
University of Rhode Island, jfaella@gmail.com

ON PERFORMANCE OF GPU AND DSP ARCHITECTURES FOR COMPUTATIONALLY INTENSIVE APPLICATIONS

BY

JOHN FAELLA

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENE

IN

ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2013

MASTER OF SCIENCE THESIS

OF

JOHN FAELLA

APPROVED:

Thesis Committee:

Major Professor Dr. Jien-Chung Lo

Dr. Resit Sendag

Dr. Lutz Hamel

Nasser H. Zawia
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2013

ABSTRACT

This thesis focuses on the implementations of a support vector machine (SVM) algorithm on digital signal processor (DSP), graphics processor unit (GPU), and a common Intel i7 core architecture. The purpose of this work is to identify which of the three is most suitable for SVM implementation. The performance is measured by looking at the time required by each of the architectures per prediction. This work also provides an analysis of possible alternatives to existing implementations of computationally intensive algorithms, such as SVM. Some performance improving methods were proposed and examined for the given DSP and GPU architectures.

The 4-class and 7-class implementations of the SVM algorithm were examined. On the system with an Intel i7-2720QM CPU at 2.2GHz, the execution times on a per prediction basis were 364μs for the 4-class implementation, and 410μs for the 7-class implementation.

On the Spectrum Digital TMS320C6713 DSP development board at 225MHz, the 4-class SVM implementation uses 125ms and the 7-class version needs 165ms. After careful examination of the DSP architecture, the following are implemented to improve the performance: (1) number of memory accesses is greatly reduced via programming technique, (2) the L2 cache is better utilized, and (3) the number of branch statements is reduced. As a result, the run time for 4-class SVM is improved from125ms to only 9ms, and from 165ms to 11ms for the 7-class implementation.

On the Nvidia Geforce GT 540m graphics card at 1334MHz, the 4-class SVM needs 798μs, and the 7-class implementation requires 845μs. Again, the GPU's architecture is investigated and the following are used to improve the performance: (1)

eliminating excessive memory accesses, (2) taking advantage of memory coalescing, and (3) the use of the reduction method. The improvements resulted in a decrease in the execution time from 798μs to 175μs for the 4-class SVM implementation and from 845μs to 200μs for the 7-class implementation.

Because the three architectures studied here are incorporated in three very different systems, running at different clock speeds, a direct comparison of the run time is not possible. The DSP system runs at roughly 10 times slower clock speed than the Intel i7 core system, and achieved more than 20 times slower run times. We cannot directly extrapolate this result; however, we observed that DSP does have its drawbacks when implementing the SVM algorithm. The DSP processor was designed specifically to support computationally intensive DSP algorithms. However, SVM algorithm is somewhat different from traditional DSP algorithms and thus some DSP architectural features are not applicable.

From the experimental results, we may observe that GPU is most suitable for the SVM algorithm. Even though it runs at a lower clock speed, about 60% of that of Intel i7 core, with the performance improvement techniques, the GPU outperforms the i7 counterpart. This may be attributed to the GPU's architectural support for parallel computations and its flexibility to adapt to various computationally intensive algorithms.

ACKNOWLEDGMENTS

I would like to thank my major professor Dr. Jien-Chung Lo for his help, time, and support. I would also like to thank the other members of my committee Dr. Resit Sendag, Dr. Lutz Hamel, and Dr. Otto Gregory, for their help and time.

I would also like to thank my colleague at the Naval Undersea Warfare Center Robert Hernandez for all his help and for encouraging me to complete my Master's degree. I would also like to thank my branch head Ruth Fisher and the Naval Acquisition Career Center for giving me the opportunity to finish my Master's degree.

Lastly, I would like to thank my parents Charlie and Kathy Faella for loving and supporting me throughout my whole life, without which I wouldn't have even started this endeavor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

There are many different types of computing architectures that exist in the world today. Each is developed for its own specific reasons and its own purpose in mind. Applications can be developed to run on any of these different architectures. However, in order to achieve the best performance possible on a given system the program must be made to best utilize the advantages of that specific design. In this study several different architectures are used to implement and run an SVM application.

The most commonly used and the best known architecture is the central processing unit or CPU. The CPU is the driving force behind all modern consumer computers be they desktops or laptops. A CPU is also used in a number of other electronic devices such as cellphones, video game consoles, and even cars. The CPU can be and is used for a very wide array of applications. Anything from running simple word processing applications such as the one used for this paper, to complex research studies that seek to find the medical treatments of the future. CPUs are readily available and come in a variety of different performance ranges. A low end CPU might be used in a cellphone, while a super computer would use a high end CPU that is highly optimized and uses all of the newest technology to provide the highest possible performance. Since the CPU is so commonly used, and is the major piece of a consumer product that almost everyone uses, the most effort has been made into increasing the performance of the CPU. During the lifetime of the CPU there have been many breakthroughs and its performance has increased tremendously. Similarly,

since CPUs are so widely available they are the architecture most often used and the architecture that is easiest to use from a programmers perspective. There are many tools available and many different programming languages that can all be easily used to develop and run programs on a CPU. Also, since there are so many different possible uses for a CPU most are designed to be able to do a wide variety of things. For this reason it is often a good choice for general purpose programming as the CPU can be used to execute most any action that a programmer might want.

A digital signal processor or DSP is a more specialized device than the CPU. A DSP is a device designed with the intention of performing some sort of digital signal processing. That is, it is designed with the purpose of accepting incoming data samples, performing an operation on them, and then outputting the result. This makes a DSP ideal for processing different signals that are commonly used in everyday life. Examples of situations where a DSP may be used include such things as a cellphone to process the incoming signal, a radio, or for image processing. With these sort of applications in mind a DSP is also tailored to perform these types of operations. Although used in a wide variety of everyday devices the DSP is not as well-known as a CPU due to its more specific purpose. For this same reason it is also less often used for general purpose applications.

The last architecture examined in this study is the graphics processing unit or GPU. A GPU is included in every laptop and desktop as well as most video game consoles. It is used to perform the graphics processing that is required to manage the display of the system. The degree of processing needed depends on the application. For simple everyday use such as running a word processing application or looking up

information on Google the GPU is not put into heavy use. However, for graphics intensive processes such as running a game or 3D graphics manipulation the GPU is very important. For these applications many operations have to occur on a data set as quickly as possible in order for the display to keep up with the rest of the program. GPUs were designed to fulfill this purpose and for a long time that is all that they were used for. It was not until the last several years that programmers began to realize that the nature of these devices could be taken advantage of to perform operations for applications other than graphics processing. With this discovery work began on creating general purpose programs for the GPU. This is known as GPGPU programming or general purpose graphics processor unit programming. Due to the design of the GPU programs can be written in new ways to be able to perform actions at speeds not previously accomplishable on other architectures.

Each of these architectures was designed with their own unique purpose in mind. However, the same application can be made to run on each of them. This may require different programming tools and different programming languages but it can often be done. When creating a program it is important to look at how the architecture affects the programs performance. By keeping this in mind it is possible to create a program that takes advantage of an architecture's opportunities while avoiding as much as possible those things that could potentially degrade the performance. This study seeks to accomplish this goal with an application for artificial leg control. The study shows how the application was modified such that it would not only be able to execute on each of these architectures, but also achieve a high performance. To accomplish this, the study looks at the different advantages of each of the architectures

and breaks down the program into smaller steps to detail how each of the architectures affected the different actions that were required in each step of the program.

Chapter 2

Architectures and Applications

The two architectures that this study focuses on, the DSP and GPU, have a wide array of uses. Both have their own unique attributes that make them ideal for different applications. Therefore it is important to look at how exactly these devices work, some common applications, and some past work that has been completed utilizing these architectures. It is also necessary to present the application that is used for this study.

2.1 DSP Overview

Digital signal processing is the sampling of and mathematical processing of inputted data to produce some sort of output signal. This is a very useful process for a large number of applications. It can be useful for telecommunications, mass-storage, cameras, hearing aids, and consumer audio gear [2]. It's used for image processing and medical instruments. As many of these applications are time dependent it is very important to be able to perform this processing as fast as possible. Digital signal processors were designed to fulfill this need. Many of the operations required for digital signal processing, such as convolution and FFTs, can be performed as a series of multiply accumulate operations, DSPs were designed to be able to perform these actions in an efficient and fast manner [2]. The result is a processor that is very good for signal processing and can perform such actions as FFTs and convolution as well as such operations as sine and cosine at high speeds.

For most DSPs this ability is accomplished by including multiple algebraic units. In this way the processors are capable of performing more than one of these

crucial operations at once. There are several different ways that DSPs try to take advantage of this added capability. One method that DSPs use is known as single instruction multiple data or SIMD. In a DSP that utilizes SIMD each instruction issues the same operation to be performed on multiple sets of data samples [3]. This allows the necessary operations for multiple data samples to be issued and executed at the same time. Another common method, and the one utilized by the processor in this study, is the very long instruction word or VLIW architecture. In a VLIW system each instruction word is very long and actually contains more than one instruction. Each of the instructions in the instruction word is issued and executed in parallel [3]. Again, this allows for parallel execution and allows for multiple operations to be completed at once. In VLIW each instruction does not have to be the same operation, this allows for multiple different operations to be completed at once.

Originally DSP programs were mainly programmed in assembly as the programmer was much more capable of creating a more efficient program than any compiler. With the creation of the SIMD and VLIW assembly programming for these devices greatly increased in difficulty. A programmer has to keep track of what operations can be grouped together and executed in parallel. It is much easier for a compiler to do this work than a programmer and thus more programs are written in high level language now [3].

While DSPs do have several advantages over a CPU when it comes to digital signal processing, they also face several major disadvantages. One is that the average DSP has a much lower clock speed than the average CPU. This means that even with the ability to issue multiple instructions per clock it is still hard to execute as many as

with a CPU. Also, DSPs usually have much smaller memory sizes than a CPU, both for RAM and cache. This means that less data can be stored on a DSP and it often takes longer to access it [3].

The advantages of the DSP architecture do give opportunities to provide a potential performance increase. However, with CPUs being such a big consumer product that everyone knows about there has been a lot of work done on speeding up CPUs. This means that even with the advantages that a DSP has for specialized applications it still has trouble competing with the general purpose CPU for these applications. As can be seen in [3], the Texas Instruments TMS320C67xx (the same series as used in this study) didn't even keep up with an Intel Pentium III CPU running at 1.13GHz when performing FFTs and FIRs. Today's CPUs can run at even higher speeds, the one used in this study runs at 2.2GHz or almost double the speed of the Pentium III. During this time algorithms for such things as out of order executing, branch prediction, and pre fetching have all also improved. These types of optimizations are not even present in the DSP making it increasing difficult of the DSP to compete.

This doesn't mean that DSPs cannot compete with CPUs. There are still many applications where a DSP is ideal. There simply is not a need for many of the features that a CPU uses for many applications. Using a DSP instead provides a low power and low cost alternative that can still function at high speeds due to its specialization. As [4] discusses, although DSPs have seemingly diminished in importance it is simply because they are more specialized than ever and are more often called something else due to this new level of specialization.

One thing that some past studies focus on is proposing new DSP designs that could be used as possible improvements for certain applications. An example of this is [5] where the authors created a new architecture and instruction set for a DSP that would be especially efficient at performing FFTs. Their results showed that their architecture could potentially outperform current DSP architectures for this specific task. Many other studies focus on the use of DSPs for various algorithms that they have designed for different applications. Examples include [6] where a DSP is used to process sinusoidal signals for an application that could be used for mobile measurement equipment. Another example is the [7] in which a cellular neural network implemented on a DSP is used to analyze images of partial discharge in a high voltage insulation system. While these studies generally focus on whether or not the DSP can perform their desired application in an acceptable time they do not compare the DSPs performance to other architectures. They are examples of the DSPs capabilities rather than a performance analysis. This study will look at the DSPs ability to execute the given application, how it compares to other architectures, and what factors affected its performance.

2.2 GPU Overview

Graphic processor units were designed to be able to handle the many calculations that are required to manipulate and render the graphics that are created and displayed by a computer. Often times this requires the execution of the same calculation on a large set of data. Since this processing often has to be done in real time it must be completed as quickly as possible. The GPU was the answer to how to do this. A GPU consists of many cores. These cores are all capable of executing a

thread. Generally each thread will be performing an operation on a data sample. By allowing a large number of threads to each perform the same operation on their own data sample a set of data can be operated on in parallel.

Since the GPU was designed specifically with graphics processing in mind this was the primary application for GPUs for a long time. Recently programmers recognized that this ability for high levels of parallelization presented an opportunity for a great performance increase for many general purpose programs. Thus began what is known as general purpose graphics processor unit programming or GPGPU programming. Since GPUs were designed with the idea of graphics in mind most early work on GPUs required thinking of a way to execute the program using graphics operations. With the increased interest of GPGPU programming however, GPU designers such as NVIDIA began creating GPUs with general purpose applications in mind and created a toolkit that could be used to program these GPUs. This toolkit, known as CUDA, allows programmers to create applications that can run on the GPU to be developed in the CUDA programming language which is extremely similar to C. This allows for a far easier implementation for many applications as well as creating an increase in the number of algorithms that can potentially make use of the GPU.

As previously mentioned the GPU's main advantage is its ability to execute a program in parallel. When programming on a GPU it is important to locate all of the potential parallelization in the algorithm. This will allow for the creation of the program that takes best advantage of the GPUs potential. The downside to using a GPU is the fact that it must be coupled with a CPU. The GPU itself is not a standalone unit. In order for a program to be executed on a GPU there has to be a

CPU present to manage the execution of the program. The CPU is responsible for determining which portions of the application are completed by the GPU and what the parameters are for the operation. The CPU is also responsible for the memory management of the GPU. In order for the GPU to get the data that it is to operate on the CPU must copy the data from its memory to the GPU memory. Similarly, when the GPU is finished executing the given operation the data must be copied back from the GPU to the CPU. This is time costly operation to carryout. This means that it is important to limit this operation as much as possible.

The GPU also suffers from several other drawbacks. Similarly to the DSP, the GPU has a slower clock speed than the CPU. It also does not have the same amount of memory or cache. It does not implement branch prediction or any of that type of optimization. For this reason a GPU cannot keep up with a CPU in serial execution. That is why it is important to identify which portions of the program are serial in nature and have them execute on the CPU while the parallel sections run on the GPU. If an operation has to be executed enough times and it can be done in parallel, then the GPU can more than compensate for its slower serial execution.

There have been a large number of studies using GPUs in recent years as researchers look into how a wide range of applications fare on the GPU. The new ease of use has also contributed to an increase in the number of people wanting to conduct research into the viability of a GPU implementation for their application. Studies also look at the various ways that the GPU can be utilized to improve performance. Unlike many DSP studies, which are usually applications designed specifically for the DSP, many of the GPU studies are done using algorithms that were previously performed on

a CPU and are now looking for a performance boost. An example is study [8] which looks at a very simple application as implemented on a GPU, a low end CPU, and a high end CPU. It discusses the application's performance on the GPU relative to the CPU and also looks at several of the different variables that can be changed to improve the GPU's performance. It also looks at what factors make a program more likely to be better suited for a GPU than a CPU. Another study [9], is more an example of a study that just looks at how the GPU can handle applications that were previously implemented on a CPU. This study looks at the GPU design and discusses the possible performance improvements offered for the GPU. It then looks at how the GPU did for some specific applications such as in-game physics and computational biophysics.

2.3 SVM Application

This study looks at the implementation of a support vector machine (SVM) classifier algorithm on both the DSP and GPU. An algorithm to be used for artificial leg control was developed and outlined in study [10]. The application of this algorithm is to correctly predict user intent in order to properly control an artificial leg so that the user can walk normally on both flat surfaces and sloped surfaces, as well as up and down stairs. The algorithm uses a support vector machine classifier based off of previously collected training data to classify new data samples as one of these movement types.

SVM is used to separate non-linear data points into distinct classes. This is accomplished by creating hyper-planes to describe a dividing border between data points belonging to different classes. The process for doing this is described in [11].

A control experiment can be used to develop what is known as a training set. Data points are collected in such a way that their correct classifications are known. Using these samples with their correct classifications, the support vectors can be fit to the data in order to correctly separate the data as accurately as possible. Once this training set is developed, new data points can be compared to this existing data in order to determine the correct classification of the new samples.

In the case of the application explored in this study two different models were used. In each the data is divided into different classes representing the different terrains that a person could be navigating. In one model there are 4 classes representing standing, flat walking, stair up, and stair down. The 7-class system adds the sitting, ramp up, and ramp down classes. The data is further divided into four different phases. These phases describe the movement phases of the leg when walking. They are initial double limb stance (phase 1), single limb stance (phase 2), terminal double limb stance (phase 3), and swing (phase 4) [12]. Using data collected in support of study [12], a training model was formed to be used to make the predictions for this study.

There are several different ways of implementing multi-class SVM. In this case a one-against-one structure was used. This means that a binary hyper-plane was created to separate each pair of classes [10]. For example, a plane was created to separate the stair up and stair down classes, another was used to separate the stair up and ramp up classes, and so on. This results in six separate classifiers to be used to separate the classes for the 4-class system and 21 classifiers for the 7-class system. Furthermore, a set of each of these classifiers is required for each of the four phases. The model was

created using the radial basis function (RBF) kernel. The SVM gamma parameter chosen was .0015 [12].

For this study an offline analysis was performed. Rather than using real time data, previously collected data, specifically the data collected for study [12], was used to test the functionality of the application. In this way the results, both in terms of accuracy and performance, could be compared to those found with the alternative implementations discussed here. The data was collected on thirteen different channels, seven electromyographic (EMG) channels and six mechanical channels. The EMG channels collect signals sent from the brain to the leg muscles to communicate the action the muscle is to take. The mechanical channels collect data

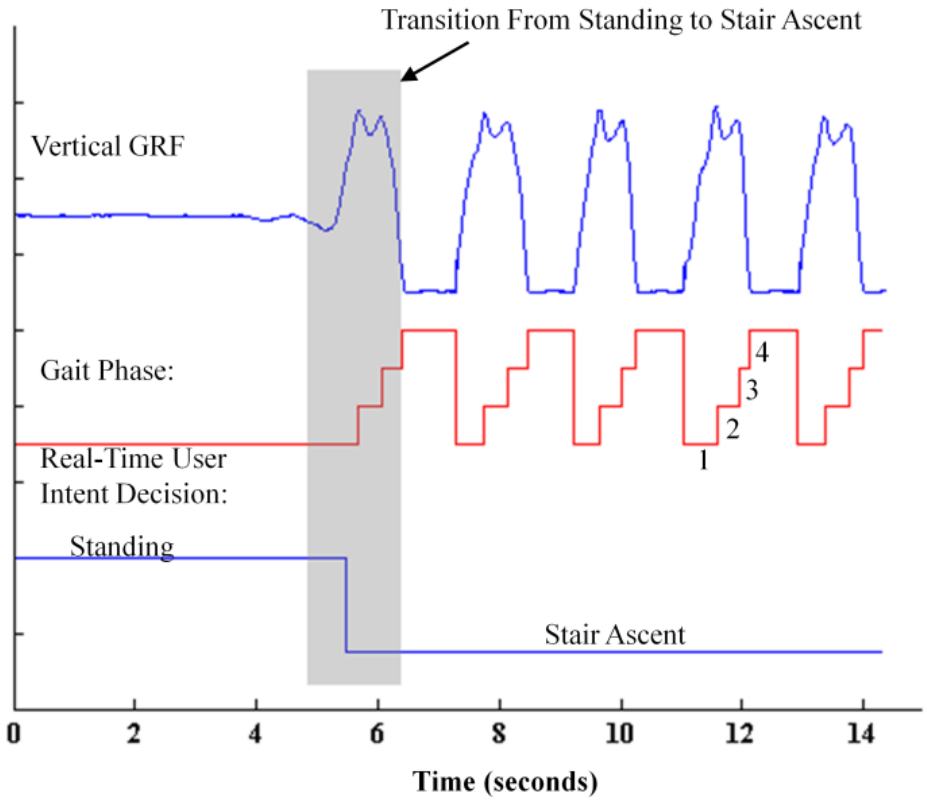

Figure 1: Gait Phase Determination [12].

using six DOF loadcells located on the prosthetic limb [10]. Each prediction is performed using a sliding window of 50ms which is composed of all the data collected on each of the thirteen channels.

The first step to making an accurate prediction of user intent is to identify the current movement phase of the user's leg. This process is performed by analyzing the vertical ground reaction force (RBF) as shown in fig. 1 [12]. The next step in the classification process is to remove any DC offset from the EMG channels. The classification algorithm requires that the EMG signal be centered around zero; however the nature of the measurement process for these channels introduces the chance of an offset. To remove this offset the mean of each channel is found and subtracted from each of the data samples in the window.

For the prediction algorithm the classifier separates data based on forty six features extracted from the channels. For each of the seven EMG channels four features need extraction. These features are the mean absolute value, the number of zero crossings, the waveform length, and the number of sign slope changes for the current window [12]. Each of the six mechanical channels requires the extraction of three features; these features being the mean, min, and max value of the channel's data for the current window [12]. Once the features have been extracted from the data they must be normalized. This is done using the normalization factors calculated from the training data [12]. The feature extraction is the first of two major sections of the program. Pseudo code is provided on the following pages to better illustrate the processing required for the feature extraction.

EMG Channel Feature Extraction:

```
/*Find the mean of each channel and subtract it from each data point to remove any
DC offset introduced from the EMG signals*/
For (index = 0 to window length)
        Sum_ch_n += ch1_data[index]
Mean_ch_n = sum_ch_n/window length
For (index = 0 to window length)
      Subtract the mean from each data point for each channel
/*Find the mean of the absolute value for each channel*/
For(index = 0 to window length)
        Sum_ch_n += abs_value(ch_n_data[index])
Mean_absolute_value_ch_n = sum_ch_n/window length
/*Find the zero crossings, waveform length, and slope turns for channel 1*/
For(index = 0 to window length – 2)
        /*Find absolute values*/
        Current = abs_value(ch1_data[index])
        Next = abs_value(ch1_data[index+1])
        Next_Next = abs_value(ch1_data[index+2])
        /*Determine if zero crossing*/
        Flag1 = 1
        Flag2 = 1
        If((ch1_data[index] >= 0 and ch1_data[index + 1] >= 0) or
           (ch1_data[index] <= 0 and ch1_data[index + 1] <= 0))
```

```
                    Flag1 = Flag2
        Else if(Current <= 0.025) and
                Next <= 0.025))
                    Flag1 = Flag2
        Else
                Flag1 = -(Flag2)
        If(Flag1 != Flag2)
                Ch1_zero_crossings = Ch1_zero_crossings + 1
        /*Determine if slope change*/
        If(((Next > Current) and (Next > Next_Next)) or
           ((Next < Current) and (Next < Next_Next)))
                /*make sure not just noise*/
                If((Next – Current >= 0.015) or (Next – Next_Next >= 0.015))
                        Ch1_Slope_Changes = Ch1_Slope_Changes + 1
        /*Determine Waveform Length*/
        Ch1_Len += square_root(((Current – Next)/20)^2 + (1/Window Length)^2)
/*Normalize Features*/
Normalize Ch1 Features by dividing by set constants
Repeat for each of the 7 EMG channels
```

Mechanical Channel Feature Extraction:

```
/*Find each feature for channel 1*/
For(index = 0 to window length)
        /*Find the sum in order to calculate the mean*/
        Ch8_Sum += Ch8_Data[index]
        /*Find the min*/
        If(Ch8_Data[index] < Ch8_Min)
                Ch8_Min = Ch8_Data[index]
        /*Find the max*/
        If(Ch8_Data[index] > Ch8_Max)
                Ch8_Max = Ch8_Data[index]
Ch8_Mean = Ch8_Sum/Window Length
/*Normalize Features*/
Normalize Ch8 Features by dividing by set constants
Repeat for each of the 6 Mechanical Channels
```

The identified current phase is then used to select which phase's set of classifiers if to be used. The forty six features are then used by the twenty one binary classifiers for the current phase. Each classifier chooses between one of the two classifications that it separates. This creates a set of either 6 or 21 "votes", with each of the possible user intents receiving votes from each classifier. The option that receives the most votes is then officially chosen as the prediction for the current window [10]. A block diagram showing the steps of the prediction process can be viewed below in fig. 2[12]. Also provided on the following page is pseudo code describing the steps required for classification of a data point.

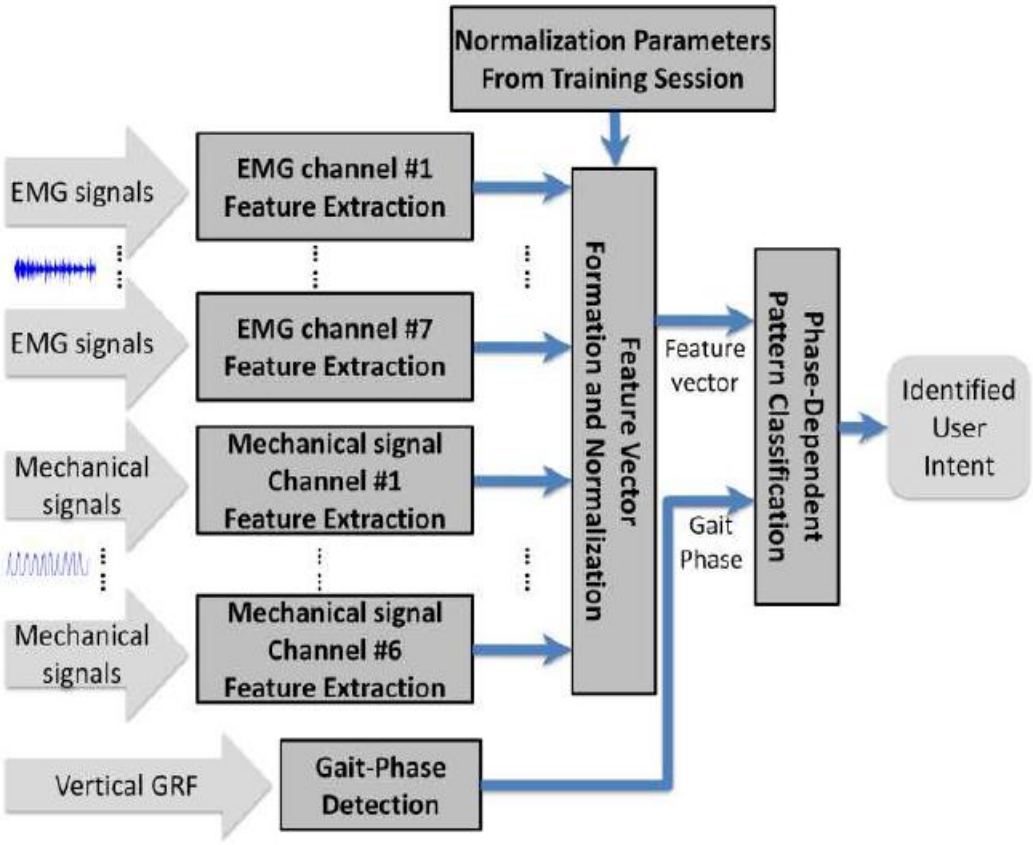

Figure 2: Program Flow [12].

Classification Steps:

/*Use the SVM RBF kernel to calculate values for each of the support vectors*/

For(index = 0 to Number of support vectors in model)

    /*Find the sum of the square of the differences for each of the 46 features between the support vector and the extracted features*/

    Sum += (SV[index].Ch1_Slope_Changes – ExtractedFeatures.Ch1_Slope_Changes)^2

    Repeat for each of the 46 features

    /*Find the kernel value for this support vector*/

    Kerenel_Value[index] = $e^{-gamma*sum}$

/*Use the kernel values to do the classification*/

For(predictNum = 0 to number of possible one vs one comparison)

    /*Look at support vectors for the first class in the comparison*/

    For(index = 0 to number of support vectors that describe first class)

        Sum += Model_Coef[index] * Kernel_Value[index]

    /*Look at support vectors for the second class in the comparison*/

    For(index = 0 to number of support vectors that describe the second class)

        Sum += Model_Coef[index] * Kernel_Value[index]

    /*Determine Vote*/

    If(sum > 0)

        Vote[predictNum] is for first class

    Else

        Vote[predictNum] is for second class

Chapter 3

Methodology

As previously stated, the study was performed as an offline analysis using data collected in support of study [12]. All data was collected and training models for the data created prior to this study. Since the work here was done as an offline analysis, the data was provided in full to the program from the beginning and then the program divided the data into windows of the correct size in order to perform the classifications. For all architectures the timing information is a measure of the average amount of time that was required for the application to make a prediction. The prediction time is defined as the amount of time required to perform the phase detection, feature extraction and normalization, and classification for each window. Further breakdowns of the performance of the two most costly portions of the program, the feature extraction and classification sections, are also included.

3.1 TMS320C6713

The digital signal processor used for this study is a Texas Instruments TMS320C6713 included on a Spectrum Digital TMS320C6713 development board. The TMS320C6713 processor uses a 225MHz clock [13]. The TMS320C6713 uses the very long instruction word (VLIW) architecture. With the VLIW architecture multiple operations are all grouped together into a single instruction [13]. This allows for a large amount of instruction level parallelism as multiple operations can all occur simultaneously. Another great advantage of the VLIW architecture type is the increased ability for programming in high level languages [13]. The VLIW

architecture comes with a smaller simpler instruction set. This means that it is far easier to design compilers that can target it. This is important as it eliminates the need for assembly language programming which can greatly increase programming time and difficulty. In this case, Code Composer Studio version 3.1 was used as the complier. Code Composer Studio allows for the development of C language programs that can be implemented on the TMS320C6713. The program was compiled using register level compiler optimization.

A major advantage of the TMS320C6713 is its inclusion of eight independent functional units. The TMS320C6713 contains two fixed point arithmetic logic units (ALU), four floating or fixed point ALUs, and two multipliers [14]. With these units the processor is capable of executing each of the multiple operations that are issued with each instruction at the same time. This again leads to increased parallelism as there is no need to wait for a previous independent operation to complete before being able to begin the next operation.

A downside to the TMS320C6713, and indeed many DSPs, is its reduced clock speed. At just 225MHz it pales in comparison to the CPU's 2.2GHz speed [14]. This results in a significantly lower number of serial instructions being issued in a similar time frame as the CPU. The question is whether the parallel mathematical abilities of the DSP, its multiple operations per instruction and eight functional units, can make up for this difference.

Another disadvantage that the DSP faces is its small memory size. The TMS320C6713 development board comes with 16MB of synchronous DRAM [15]. The memory map for the chip can be found in fig. 3 [15]. The chip has a 4KB L1

program cache, and a 4KB L1 data cache. Portions of internal memory can be configured by the software to allow for the use of an L2 cache.

A last disadvantage of using a DSP for this program is the lack of branch prediction, pre-fetching, and other algorithms of this nature used by modern CPUs. These improvements allow for a higher number of instructions to be in the pipeline at any one time and thus increase instruction throughput. This simply does not exist for most DSPs. The result is worse performance for each branch that is present in the program and worse performance overall.

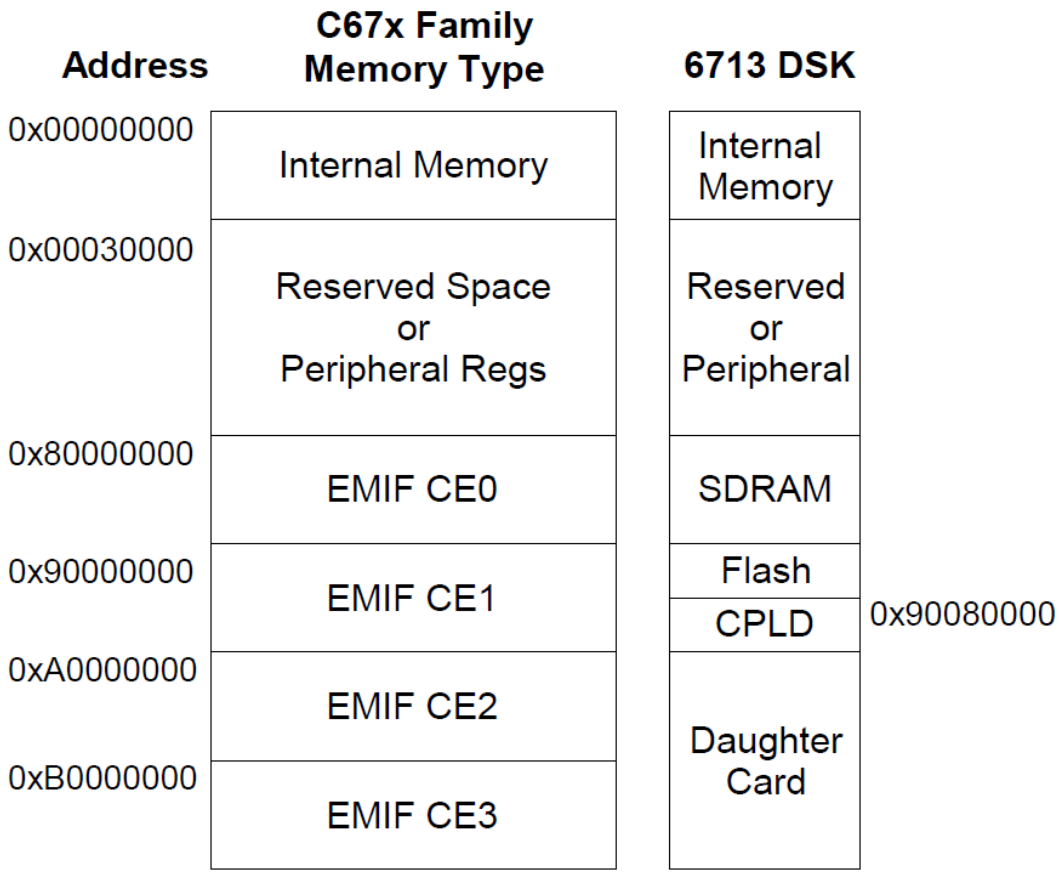

Figure 3: DSP Memory Map [15].

3.2 Geforce Gt 540m

For this study an NVIDA Geforce Gt 540m graphics card was chosen. This card was readably available in a currently owned Dell XPS laptop with an Intel i7-2720QM as the supporting CPU. This GPU is a good example of a GPU that would be easily available to anyone and represents a GPU that is middle of the road performance wise in terms of NVIDIA GPUs. The 540m runs at a clock speed of 1334MHz [16].

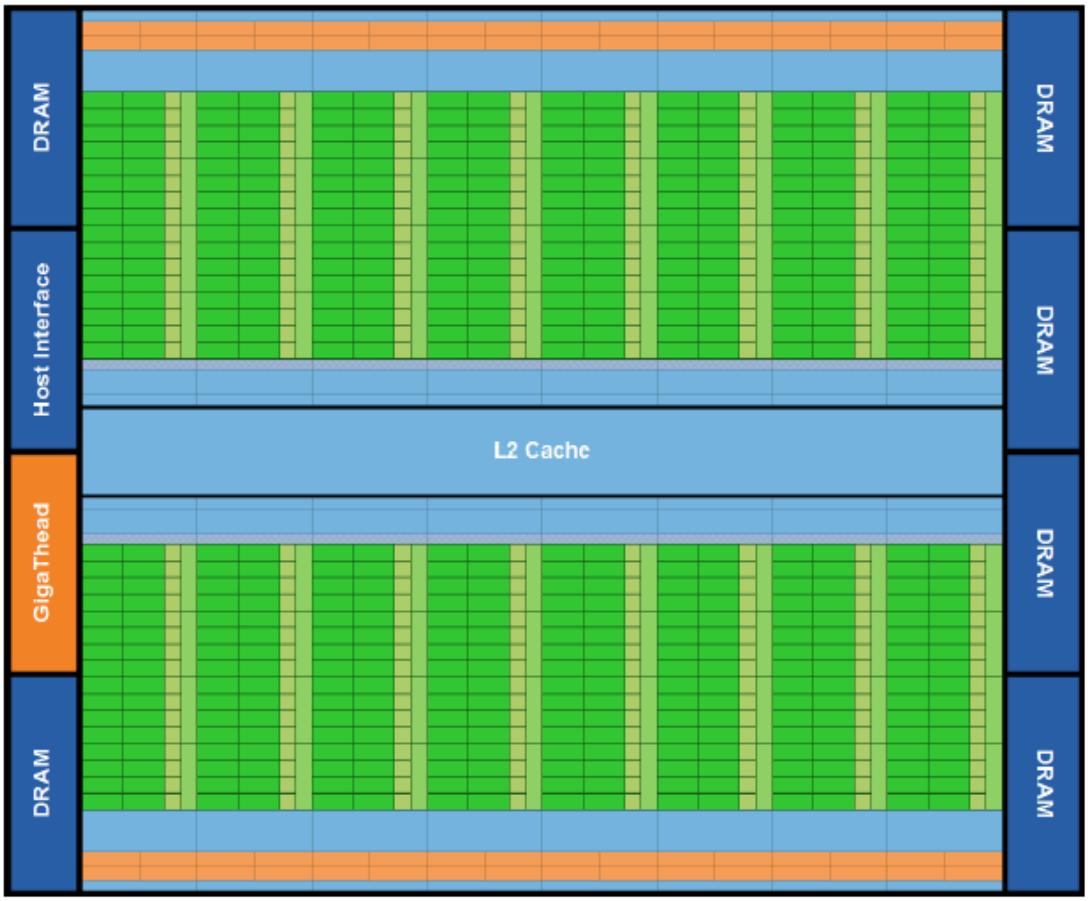

Figure 4: Example Fermi Architecture, each SM is a vertical rectangular strip that contains an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (cache) [17].

The Geforce Gt 540m was designed using a refresh of NVIDIA's Fermi graphics processor unit architecture. The Fermi architecture is NVIDIA's third generation of GPU architecture. Each Fermi device is divided into up to 16 streaming multiprocessors (SM) [17]. In the case of the refresh of the Fermi architecture each of these multiprocessors is further broken down into 48 CUDA cores as opposed to the 32 included in the original version of the Fermi architecture. Each of these cores is capable of executing a thread. In the case of the Geforce Gt 540m, there are two SMs with 48 CUDA cores each. This means that up to 96 threads can be executed in parallel at the same time. A Fermi device also contains a unified L2 cache that is shared between the SMs as well as DRAM and a PCI-Express interface to the controlling CPU. An example of a Fermi GPU with 16 SMs each with 32 CUDA cores can be seen in fig. 4[17].

Each of the 48 cores inside of a SM functions as a processor for the threads of the program and each contain a fully pipelined arithmetic integer unit and floating point unit. These units are capable of performing both single precision and double precision operations. Each SM also contains 16 load/store units allowing for addresses to be calculated for up to 16 threads per clock cycle. Each SM has four special function units for calculating such functions as sin, cosine, and square root [17]. An illustration of a 32 core SM can be seen in fig. 5 [17].

Threads are scheduled inside of a SM on a per warp basis. Threads are broken up into sets of 32, with each set of 32 being called a warp. Each warp shares a program counter (PC) as well as shared memory and some other resources. An SM contains two warp schedulers and two instruction dispatch units. Each scheduler chooses a warp to execute and dispatches an instruction to either 16 cores, the 16 load store units, or the 4 special function units. Since there is no dependence between warps, using this method allows for the simultaneous execution of two warps per SM. An illustration of this scheduling process can be seen in fig. 6 [17]. Since the threads

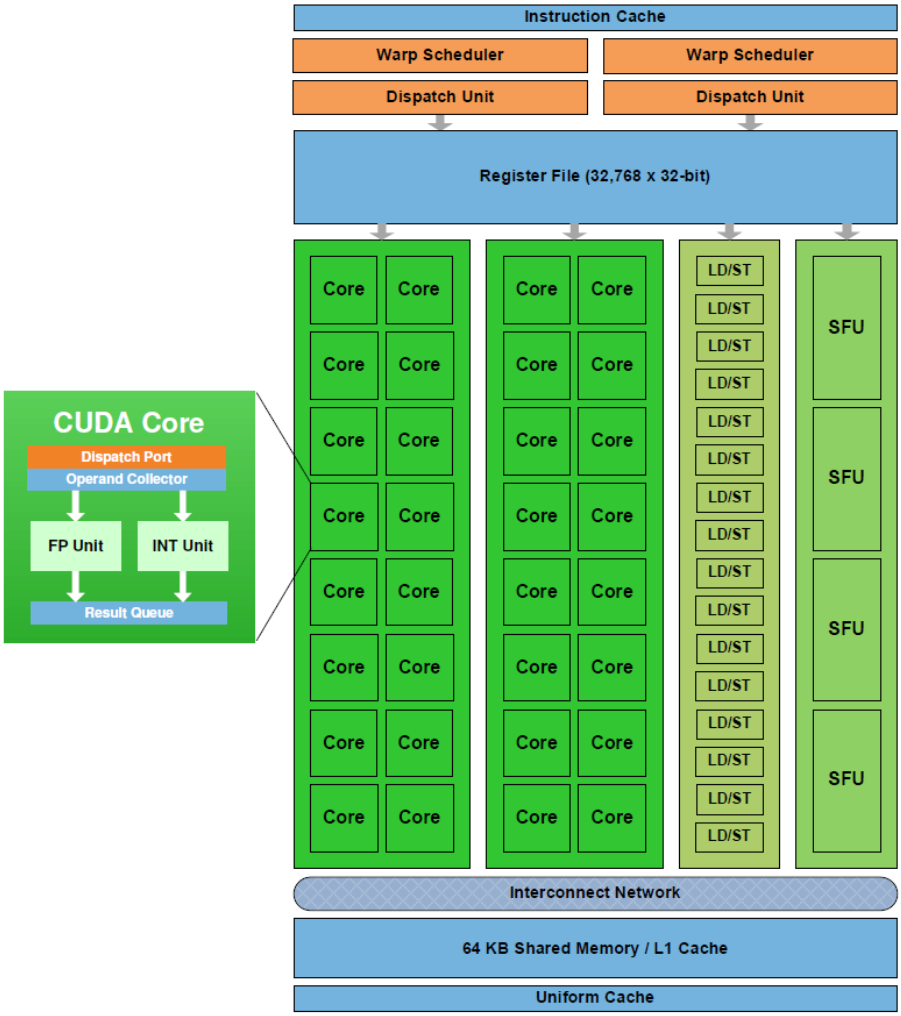

Figure 5: Fermi Streaming Multiprocessor [17].

of a warp share the same PC it is important that each thread in a warp takes the same path through the code. When there is a conditional branch in a segment of code that causes threads of the same warp to take a different path it is called a divergent warp. Divergent warps are important to avoid because they cause the threads of a warp to execute serially until all threads are back to the same location in the code. This removes the prime advantage of using a GPU [18].

The last advantage of the Fermi architecture is the ability to launch concurrent kernels. As will be discussed in the next section, a CUDA kernel is a function that contains code that all threads in the kernel will execute. In previous versions of NVIDIA GPU architecture only one kernel could execute at any one time.

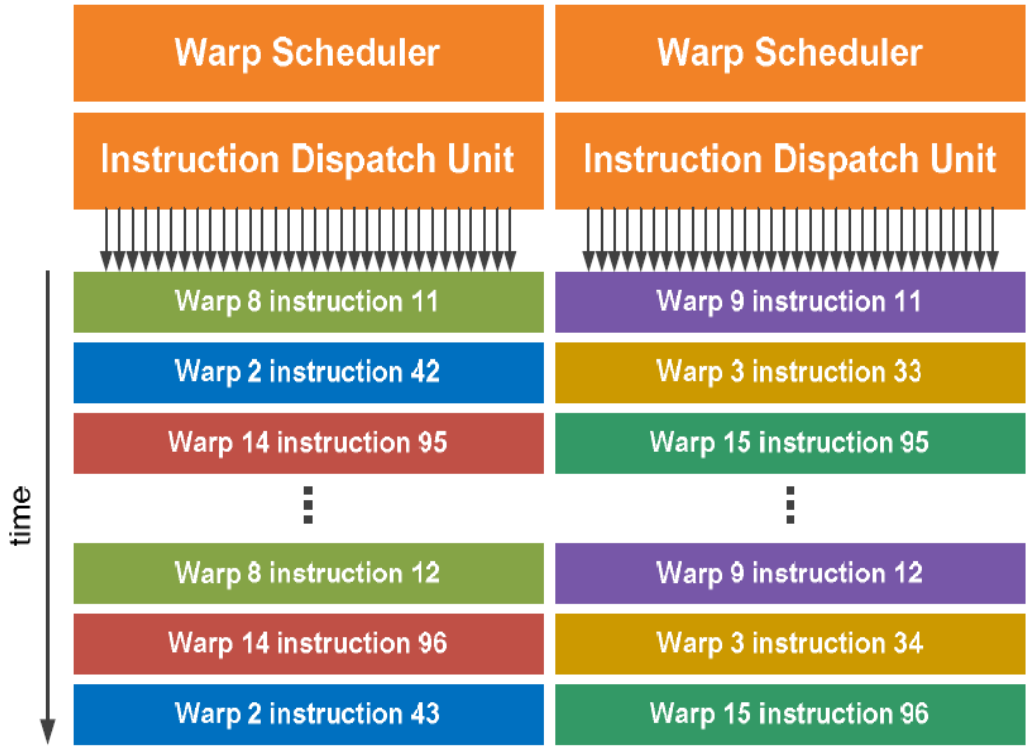

Figure 6: Warp Scheduling Example [17].

This meant that only one set of instructions could execute at any one time. This prevented independent sets of instructions from executing in parallel. With the Fermi architecture there are 16 separate CUDA streams. Each stream is its own independent serial execution schedule. Kernels issued to the same stream will be executed serially, but in parallel to any kernels executing in the other streams [17]. Any copying of data between the GPU and CPU will first wait for execution in all streams to end before executing. Similarly, any kernel launched to the default stream of 0 will wait for all previous kernels to end and will not allow for the execution of kernels in other streams while it is executing [18].

3.3 CUDA Programming Language

CUDA refers both to NVIDIA's GPU architecture and the programming language that can be used to develop programs to run on this architecture. The CUDA toolkit is an add-on that can be used to develop C like programs that can run on a CPU. For this study, the CUDA 4.2 development kit for Microsoft Visual Studio was used. This allows for the development of programs in the Microsoft Visual Studio environment. In this case Microsoft Visual Studio Professional 2010 was used as the development environment.

Since a CUDA program necessarily executes on both a GPU and CPU it is necessary to indicate which portions of the program are processed by which architecture. A CUDA program starts like any other C based program by executing in the main function. The parts of the program that are best executed on the CPU are written just as they would normally be written for any application. When there is a part of the program that the developer wishes to use the parallel nature of the GPU for

he or she must launch a CUDA kernel to perform these steps. Each CUDA kernel is similar to a C function. It takes parameters and the code is written in C syntax. CUDA functions can come in two forms. The first is a global function; this is the type of function that can be launched as a kernel from the CPU portion of the program. By definition all global kernels have a void return type. The second kind is the device function which can be called by global functions and other device functions but not by the CPU. Device functions can have any return type and are by definition inline functions [18].

When launching a CUDA kernel the CPU must determine several parameters for the launch. These parameters have to do with the number of blocks and threads that should be launched in order to execute the given kernel. All of the threads launched for the kernel will execute the same code; that being the code included in the chosen CUDA function. A block is a group of threads that can share the same set of shared memory and can also be synced together regardless of whether they are in the same warp or not. Threads in different blocks have no connection to each other and cannot be synced nor share memory. Also, all threads in a single block will be executed on the same SM so in order to fully take advantage of the two SMs included in the Geforce Gt 540m at least two blocks should be launched. For this reason it is important to consider the configuration of threads into blocks when thinking about the use of memory and the scheduling of threads. The two parameters that are required when launching a CUDA kernel are the number of blocks to be launched and the number of threads per block. Two additional parameters can also be defined. The first of these two values is the grid number. This would be used for systems using

more than one GPU and thus could execute multiple grids at once on separate devices. The second number is the stream that this kernel will be performed in. Leaving out these parameters will default to grid 0 and stream 0. An example of the syntax for these calls is included below:

```
Name<<<Num Blocks, Threads Per Block, Grid Num, Stream Num>>>(arg1, arg2)
```

Another important concept in CUDA programming is memory management. The memory for the CPU and GPU is not shared, meaning that the CPU cannot access the GPUs memory, nor can the GPU access the CPUs memory. This means that the sharing of data between the two devices can only be accomplished through the copying of data using the cudaMemcpy function. This function allows for the copying of data back and forth between the two devices. Space must be allocated in GPU memory prior to any data being copied to a location in GPU memory. This is done using a call to the cudaMemAlloc function by the CPU. This allocates the required space and returns a pointer to this address space. This pointer can be used both to copy data, and as an argument to a kernel launch to inform the GPU of where to find needed data in its memory. Since memory allocation and the copying of data both require communication between the CPU and GPU and the movement of data across the connecting bus, it is a very expensive process. From a performance perspective this means that it is best to perform this operation as little as possible.

3.4 CPU Results

This application had previously been created for another study where the program was executed on an Atom CPU [12]. The advantage of an Atom CPU is that it is an extremely low power mobile CPU. This makes it ideal for this application as it could be easily connected to a prosthetic limb. The results of this study determined that the program implemented on the Atom CPU required an average of 869us per prediction.

For this study, it seemed important to not only compare the results to the performance of the low power CPU, but also to compare them to the performance of a CPU that was more likely to be found in a modern computer. For this reason the program was run using Microsoft Visual Studio Professional 2010 on a Dell XPS laptop with an Intel i7-2720QM processor. The i7-2720QM processor runs at 2.2GHz. It was found that the i7 could execute one prediction every 410µs. Further breaking this down, it was found that the extraction of features from the EMG and mechanical channels took 224µs of that time. The execution of the kernel function, the classifiers, and the voting required another 185µs of this time.

Chapter 4

DSP Implementation

For the DSP the program was written using Code Composer Studio. This allowed for the development of the program on a laptop using the C programming language. The program could then be compiled and loaded onto the board using a USB programmer.

4.1 Direct Conversion Approach

As previously mentioned, one advantage of using the TMS320C6713 development board is the ease of use in both coding the application and programming the board. This is important to mention, as this study looks at the advantages of each of the architectures and ease of use is certainly important especially in industry where each hour of manpower designated to development costs valuable time and money. Therefore, the original approach was to see how quickly the program could be made to work by simply using the existing code from the CPU implementation. This was done using the 7-class version of the program. The conversion required time for familiarization to the development environment, several minor changes in syntax, a development of a memory map file, and changes to the time keeping method. There were also some changes made to the way that data was stored in the feature vector. In the previous version data was moved back and forth between strings. In this case the data was just moved directly to the vector without the use of strings.

The last and biggest problem in this implementation was the small amount of memory provided to the DSP. Since the size of SRAM is relatively small, there is only room for enough data for about fifteen predictions. This would not be a problem

in the case of a real time implementation of the algorithm as data would be fed into the program in only small samples, but in the case of an offline analysis such as this where all of the data is stored in memory, it did require a workaround. The solution was to write a small program to truncate the data and then use this smaller size to create a data set that could actually fit in the limited space provided to the DSP.

The results from this original implementation show that simply converting a program from Microsoft Visual Studio to Code Composer Studio and programming the DSP that way, while fast on the programming side, is not efficient from a performance standpoint. The average time per prediction for this original version of the program was 161.83ms. The breakdown of this result was that 81.54ms were required to do the feature extraction portion of the program, and 79.89ms were required for the kernel function and classification portion. The total time required per prediction for this version of the program was about 394 times longer than that of a prediction performed by the CPU. This result showed that just copying a program from Visual Studio to Code Composer was relatively quick to do, but was not a viable option from a performance standpoint.

4.2 Elimination of Memory Accesses

Believing that the time to access memory was an extremely limiting factor in the program execution time, it was determined that a good strategy for reducing the runtime would be to limit the number of memory accesses made by the program. It was found that the place where the most memory accesses were occurring was in the EMG feature extraction portion of the algorithm. This was especially true of the calculation for the number of zero crossings and slope changes. In this part of the

code a loop was used to go through all of the data samples for the current window. Each time through the loop three separate memory accesses took place. The first access was to load the previous data sample, the middle access loaded the current sample, and the third loaded the next sample. This allowed for comparison of data point locations, either positive or negative for determining zero crossings, and their locations relative to each other to determine slope. The absolute value of each of these data points was also being calculated each time through the loop.

Each of these memory locations did not have to be accessed each time through the loop. Instead, the program was redesigned to accesses only the next data point. The value of the previous data point could be reassigned to be the value of the current data point, and the value of the current data point could be assigned the value of the next data point. In this way only the value of the next next data point had to be read from memory each time through the loop. Similarly, by reassigning the absolute values that had been calculated in the same way, only one absolute value had to be calculated each time. Example pseudo code for calculating the slope changes using this method is included on the next page. The previous method can be found on pages 15 and 16. This method would also allow for a reduction in the number of reads for the zero crossings and waveform length calculations.

Eliminating Memory Accesses:

```
/*This loop would calculate the zero crossings, slope changes, and waveform length,
shown here is just the calculation for the slope changes*/
/*First read in initial values*/
Current = Ch1_Data[0]
Next = Ch1_Data[1]
Next_Next = Ch1_Data[2]
Abs_Curr = Abs_value(Current)
Abs_Next = Abs_value(Next)
Abs_Next_Next = Abs_value(Next_Next)
For(Window Length – 2)
        /*Determine if slope change*/
        If(((Abs_Next > Abs_Current) and (Abs_Next > Abs_Next_Next)) or
           ((Abs_Next < Abs_Current) and (Abs_Next < Abs_Next_Next)))
                /*make sure not just noise*/
                If((Abs_Next –Abs_Current >= 0.015) or
                   (Abs_Next – Abs_Next_Next >= 0.015))
                        Ch1_Slope_Changes = Ch1_Slope_Changes + 1
        /*Get next set of values*/
        Current = Next
        Next = Next_Next
```

```
Next_Next = Ch1_Data[index + 3]

Abs_Current = Abs_Next

Abs_Next = Abs_Next_Next

Abs_Next_Next = Abs_value(Next_Next)
```

The result of this new version of the program showed a per prediction time of 161.26ms with the feature extraction time being 80.91ms and the kernel functions and classification taking 79.95ms. This result showed just a very modest increase in performance for the new version of the program. Despite the need for a minor increase in overhead to accomplish this goal there was still a slight decrease in the time required for the feature extraction. This shows that the memory access time is greater than the times for other operations, but cannot be limited that much by this type of optimization.

4.3 Enabling L2 Cache

The next optimization that was used yielded significantly better results and also confirmed that memory accesses were indeed the largest cause of delay for the DSP. It was discovered that the L2 cache for the TMS3206713 is only used if a portion of the internal memory is setup to act as a L2 cache in software. In the software the configuration of a 64KB L2 cache was setup and the program was executed again.

The addition of L2 cache saw a significant performance gain for the DSP. Runtime decreased to 13.94ms, with 7.05ms of that being for the feature extraction and 6.84ms needed for the classification. This result showed a performance increase of 11.58 times as compared to the no cache version of the program. Without configuring a portion of memory to act as an L2 cache the TMS320C6713 was limited to just the small 4KB L1 cache. This was not enough space to sufficiently store a lot of the data that is needed for the application. The inclusion of a portion of memory as a 64KB L2 cache provided the application with a significant amount of space in which

data that was regularly used could be kept in between uses. This meant that far less memory accesses were required to make a request to DRAM. These results also showed just how slow memory accesses to DRAM are for DSP. By providing a way to reduce the number of accesses to DRAM the program increased in speed significantly.

The application was also applied to a 4-class system. In this case there are just 4 different possible motion types. This means that the number of support vectors required to separate the data classes in the model is generally lower than the number required for the 7-class system. It also means that fewer classifiers have to run as there are only 6 comparisons that have to be made rather than the 21 required in the 7-class system. The change does not result in any decrease in the number of features that have to be extracted from the data. For this reason it is expected that the change will only lead to a decrease in the amount of time required for the classification portion of the program. The results back up this hypothesis. In the version of the program where cache was not enabled the 4-class system required 124.63ms per prediction. 81.45ms was required for the feature extraction portion of the program which is similar to the 79.89ms for the 7-class system. 42.8ms were required for the classification section of the program which is significantly less than the 79.95ms needed in the 7-class model. With cache enabled the total time dropped from 13.94ms to 10.71ms, the feature extraction remained relatively constant at 6.98ms versus 7.05 in the 7-class, and the classification time fell to 3.69ms versus the 6.48ms for the 7-class version.

4.4 Reducing the Number of Branches

A last technique that improved the performance of the DSP was an attempt to reduce the number of branches that occur in the algorithm. This was first done in the classification portion of the algorithm. In the portion of the program where the classification algorithm is run there are two separate loops used to multiply and add the values that are calculated using the kernel functions with the coefficients of the model. By grouping these loops together two things happened. One was that two processes using a lot of add and multiply operations were grouped together allowing for the DSP to take advantage of its multiple arithmetic units. Second was that the number of branches was reduced as only one loop was used and thus all the branches for the second loop were removed. Since this process is done for each of the classifiers this eliminates many branches from the program.

Another improvement that was made to the classification portion of the algorithm was to change the way that the kernel functions were run. The kernel function code was written in such a way as to work for any number of features and to work regardless of whether the user correctly ordered the features in the feature vector. Knowing that a set number of features was used and that they were correctly loaded into the feature vector allowed for the elimination of several branches in the kernel function code.

The results of these improvements lead to a decrease of the classification time from 3.62ms to 2.49ms for the 4-class system and from 6.85ms to 4.66ms for the 7-class system. The decrease represented an about 32% reduction in classification time for both systems.

This method was also applied to the feature extraction portion of the program. For the calculation of the number of zero crossings for each EMG channel, 4 if statements were used. This was consolidated into just 1 statement. Similarly the number of if statements used to calculate the number of slope turns for each of the EMG channels was reduced from 2 to 1. This eliminated a number of branches during the feature extraction process. Unfortunately, there are still many branches in the feature extraction section of the code since they are unavoidable for many of the features such as the min and max calculations. Even in the situations that were improved a branch is still required. The result of these changes was a decrease in the feature extraction time for the 7-class system from 7.17ms to 6.89ms, and a similar decrease from 7.04ms to 6.94ms for the 4-class system. This change resulted in a much smaller improvement than the changes made to the classification portion of the algorithm. It is possible that some optimizations were already made by the compiler. Also, the number of branches eliminated in this case was much lower than the number of branches eliminated in the kernel function calculation code as in that case the number of kernel functions run was anywhere from 100 to 400 whereas there are only 7 EMG channels so this code is iterated over far fewer times. This means that these improvements had less of a chance of making as large of a difference in the execution time.

An example of how the number of branches was reduced is the calculation of the number of zero crossings. The original code for this calculation can be found on page 16. In the new version provided on the following page the three if statements are combined into one. This reduces the number of stalls introduced by branching.

Combining of If Statements for Slope Change Feature Extraction:

```
/*Determine if zero crossing*/
For(index = 0 to window length – 2)
        If(!((Current <= 0 and Next <= 0) or (Current >= 0 and Next >= 0) or
            (Abs_Value(Current) < 0.025 and Abs_Value(Next < 0.025))
                Ch_Zero_Crossings += Ch_Zero_Crossings + 1
```

A last improvement that was made to the DSP was to change some of the special functions to use single precision. Specifically the absolute value and square root functions in the feature extraction portion of the program were changed to use single precision instead of double precision. This change led to a decrease in the feature extraction time to 6.24ms for the 7-class system and 6.35ms for the 4-class system. The total time for the two systems was reduced to 11.11ms for the 7-class system and 8.88ms for the 4-class system. To make this a fair comparison the same functions in the CPU version of the program were also changed to use single precision. However, this change actually resulted in an increase in the execution time for the CPU version.

4.5 Results

In this case it was found that the best improvements that could be made to the DSP came from decreasing the number of memory accesses to DRAM. This was accomplished not as much from the way the code was written but by ensuring that the architecture was configured in such a way as to be able to best optimize its memory accesses. The inclusion of L2 cache allowed for this to happen and resulted in a large improvement over the original implementation of the program which was just a direct port form the CPU. Another good performance optimization for the DSP is the reduction of the number of branches. This allows the pipeline to stay full more often than when a lot of branches are used. Since the DSP has no branch prediction, branch statements lead to many stalls being entered into the pipeline and thus wasting time. The elimination of branches also often leads to grouping more arithmetic statements

together which allows for better use of the DSPs multiple arithmetic units and VILW architecture.

For this application, the DSP is not the best option from a performance standpoint. It simply cannot keep up with the CPU implementation in terms of speed. Part of the reason for this is due to the application. Much of the application does not take best advantage of the opportunities provided by the DSP architecture. For instance, the DSP was designed to improve the performance of multiply-accumulate operations. In the feature extraction portion of the program there is little opportunity to do this. The operation that takes place is often a comparison rather than a multiply or add. This doesn't allow for the parallel operations that the DSP is capable of and also leads to decreased performance due to branching. Despite these problems, the ability to program the DSP in straight C does make it an attractive option as it allows for ease of programming.

Chapter 5

GPU Implementation

By design graphic processor units present a highly paralleled architecture. The reason for this is that GPUs are meant to perform the same operation on a large data set all at the same time. Originally this was so that a GPU could perform operations on the large number of pixels in an image. Recently it was found that this functionality could also be useful for more general applications. A GPU consists of a large number of small cores so that many threads can run in parallel to perform the needed operation. In this way large data sets that would require a long time to be processed using a serial algorithm on a CPU can instead run in parallel on a GPU.

Some disadvantages of using a GPU include slower clock speeds than a CPU, no branch prediction, and smaller memory sizes. Another large disadvantage of a GPU is the increased amount of time required by memory accesses to global GPU memory. Each thread has only a few registers and then data is often stored in global memory. Accesses to global memory are time costly and a good program limits the number of these accesses. The biggest disadvantage of using a GPU is the need for a host CPU. A GPU program cannot run on its own, in this way a GPU implementation is more of a hybrid approach than a strictly GPU approach. The GPU requires a CPU to determine when and with what parameters a portion of the program should be executed on the GPU. Also, data must be moved between the CPU and GPU as neither device can access the other's memory directly. This movement of data is very costly to performance and needs to be reduced to just when it is absolutely necessary. Since the CPU is needed anyways it is best to make those portions of the program that

must be performed serially be executed by the CPU while the sections of the application that can be improved by parallelization are executed on the GPU.

5.1 Parallel Channels

Unlike with the DSP it is not possible to simply use the existing program with the GPU. Most of the existing code can be used as the basis for the program though. The existing code can be used for the portion of the program that exists on the CPU and then CUDA functions have to be created to control those processes that occur on the GPU. This means that the first step is to identify which sections of the application can be executed in parallel.

In the initial review of the code it was determined that the feature extraction and classification portion of the program could be implemented in parallel. For the features, each of the 7 features that are needed can all be executed in parallel. Similarly, each of the 13 channels can be executed in parallel resulting in 46 parallel processes. For the classification part of the application, each of the 21 classifiers can be executed in parallel.

In order for these sections of the application to run on the GPU the first step was to get the data that was needed for them to correctly operate on the GPU. This means that the first step in the program is to copy all of the data samples and the model to the GPU. It also requires the allocation of memory space in the GPU in order to control the copying of data to and from the GPU, as well as to create pointers that can be passed to GPU functions. The copying of the data samples from the CPU to the GPU was a straightforward task. However, the copying of the training models does not work well with the cudaMemcpy function because these functions cannot

copy over dynamically allocated 2-dimensional arrays that are part of a structure. This means that a kernel function has to be written that can load the models directly on the GPU. To do this first the model data is copied to the GPU, and then the model load kernel is called to load the data into the appropriate structure.

The first approach to the feature extraction portion of the program was to use the existing code to develop kernels that could be executed in parallel. To this length two kernels were created. The first kernel calculated the mean of each of the EMG channels and used this value to remove the DC offset from the channels. At the same time the data for the current window from each channel was loaded into an array of just the data for the current window. This effort turned a process that was previously seven serial calculations into one set of seven parallel calculations. Once the offset is removed from the data the features can be extracted from the data. For this process one kernel was launched with many threads. Then a warp of threads was dispatched to handle the extraction of each feature, with each warp handling a different feature. Within each warp the different threads would each operate on a different channel and in this way each of the seven features would be calculated in parallel to the others, while at the same time each of the thirteen channels would also be handled in parallel.

Once the features have been extracted from the data the classification algorithm can run. The classification algorithm is broken up into two separate kernels. The first uses one warp to do some needed overhead, such as setting each classifier's vote to zero and arranging some values, while the second warp executes the SVM RBF kernel. By executing this portion of the program on the GPU each of the 21 SVM kernels that need to be processed can be done in parallel. The second CUDA

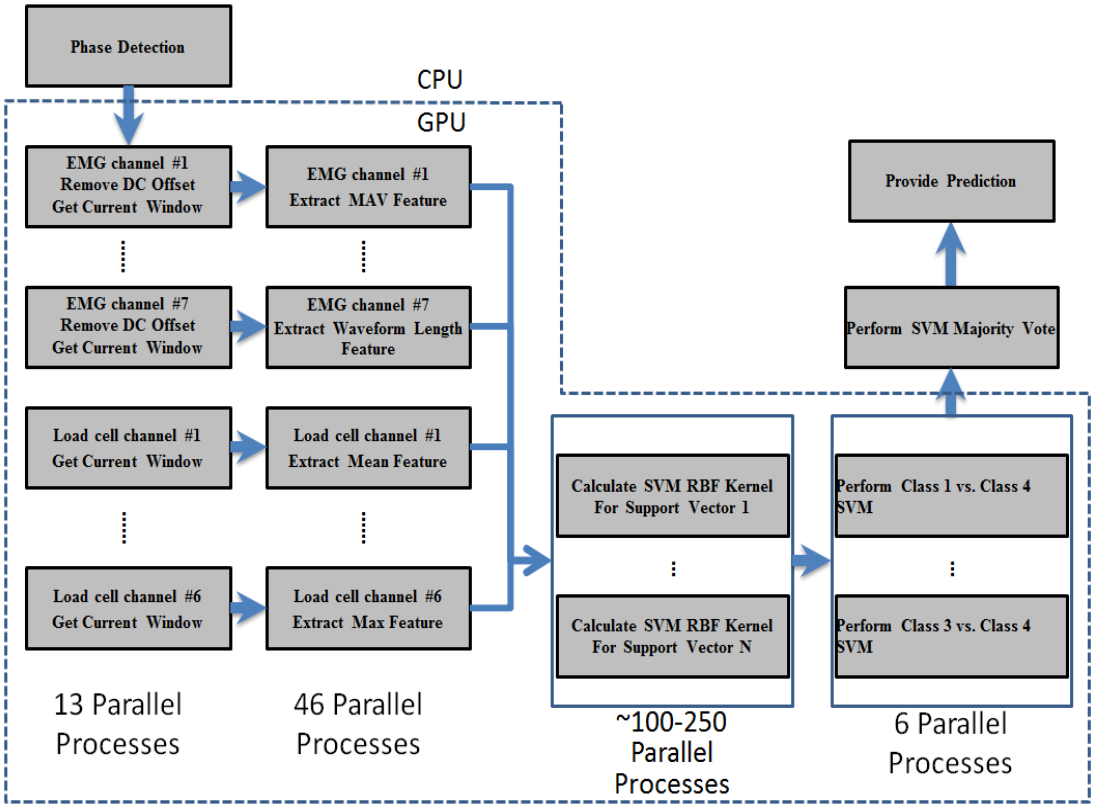

Figure 7: GPU Program Flow

kernel that is launched for the classification algorithm uses the values that were calculated by running the RBF kernel to determine the vote cast by each of the classifiers. At this point the only memory copy that is required during the prediction process copies the votes from the GPU to the CPU. The CPU then adds up the votes and determines the prediction for the current window. An illustration of this method is included in fig. 7 and pseudo code can be found on the following pages.

This original program was run on the 4-class version of the application. The results from this original program, using mostly the same code just parallelizing it where possible, were not as good as had been hoped. Each prediction required 1.48ms to complete. 944µs were required for the feature extraction process, 216µs were needed for the classification, 46µs for the data copy, and 177µs were spent doing other

various overhead processes. From this it was determined that a number of changes were required to really maximize the potential of the GPU implementation.

Parallel Channels Feature Extraction:

/*The following is an example of how this method would work for the mean of the absolute value feature, a group of threads would execute this code*/

/*The thread Id correlates to the channel number*/

```
If(threadId < 7)
        For(i=0 to window length)
                Sum += absolute value (data[threadId * window length + i])
Mav = sum/window length
```

Parallel Channels Kernel Values:

/*The following will calculate the kernel value for one specific support vector, enough threads would be created to handle all support vectors using this code, which support vector to process will be chosen based on the thread Id*/

/*Find the sum of the square of the differences for each of the 46 features between the support vector and the extracted features*/

Sum += (SV[threadId].Ch1_Slope_Changes – ExtractedFeatures.Ch1_Slope_Changes)^2

Repeat for each of the 46 features

/*Find the kernel value for this support vector*/

Kerenel_Value[ThreadId] = $e^{-gamma*sum}$

Parallel Classification:

```
/*The following code would be used to perform one of the one vs one predictions,
enough threads would be created to handle all of the predictions storing the vote for
tallying later*/
/*Look at support vectors for the first class in the comparison*/
For(index = 0 to number of support vectors that describe first class)
        Sum += Model_Coef[index] * Kernel_Value[index]
/*Look at support vectors for the second class in the comparison*/
For(index = 0 to number of support vectors that describe the second class)
        Sum += Model_Coef[index] * Kernel_Value[index]
/*Determine Vote, determine which classifier this was based off of the thread Id*/
If(sum > 0)
        Vote[threadId] is for first class
Else
        Vote[threadId] is for second class
```

5.2 Optimizing the Parallel Channels Approach

As the feature extraction took up the largest chunk of the execution time, it was determined that this would be a good area of focus for improvement. One big obstacle to obtaining the best performance is memory accesses. Accesses to global memory in a GPU are very slow. Since the data samples are stored in global memory repetitive accesses to these values are costly. The feature extraction process works by looping through the channel data for the current window. This led to many accesses to global memory and thus was a large contributor to the total execution time. One way that the designers of the Fermi architecture tried to resolve this issue was with the inclusion of memory coalescing [18]. If two threads in the same warp each access the same memory region then the data is broadcast to both threads resulting in just one memory access instead of two. Furthermore, two threads do not have to access the exact same memory location for this process to work. When a memory transaction takes place a read of up to size 128B is conducted, any accesses for the current warp within this area are processed by this single transaction [18]. Also of importance, this data is cached and thus results in faster accesses as long as it remains in the cache.

In the original version of the program this access pattern was not taken advantage of. The data for the current window was stored in a large 1D array where the data for each channel occupied the first set of indices with the data for channel two in the next set and so forth. However, each warp of threads was responsible for a separate feature rather than a separate channel. The threads within each warp were then each responsible for a different channel. While iterating through the data all the

threads of a warp accessed the same index number for its respective channel. However, these data points were separated by an entire channel worth of data. The result was that there was no memory coalescing. Each memory access for an EMG feature resulted in seven transactions, while each access for a mechanical feature resulted in six transactions. Having this many transactions so far apart also reduced the likelihood of the data remaining in cache until the next access.

To fix this problem the data arrangement was changed. The first sample for each channel was stored in the first 13 indices of the array. The next 13 spots were for the second data point from each channel and so forth. The result was that when reading these memory locations each thread in a warp would be accessing the same sample number for their channel and these samples would be close to each other.

Extraction Time (µs)	Classification Time (µs)	Total Time (µs)	Notes
944	216	1475	Original Implementation
877	216	1351	Change the memory storage for closer accesses
746	216	1163	Use single precision
421	216	864	Remove memory accesses
410	211	792	Make divides multiplies, change address calculation, move data around

Table 1: Summary of Changes to GPU Implementation on a 4-Class System.

With this new layout the number of transactions per warp was greatly reduced and the likelihood of reusable data remaining in cache was increased. With this change the time required for feature extraction was reduced by 67µs to 877µs. An example of how this was done can be seen below.

Memory Storage Changes:

```
/*Old way of storing data*/
Ch_array[0:149] = Ch1_Data[0:149]
Ch_array[150:299] = Ch2_Data[0:149]
/*Old way of accessing data for channel mean feature extraction CUDA kernel*/
ChannelNum = ThreadId
For(index = 0 to window length)
        Sum += Ch_array[ChannelNum*NumChannels + index]
/*New way of storing data*/
Ch_array[0] = Ch1_Data[0]
Ch_array[1] = Ch2_Data[0]
Ch_array[13] = Ch1_Data[1]
Ch_array[14] = Ch2_Data[1]
/*New way of accessing data for channel mean feature extraction CUDA kernel*/
ChannelNum = ThreadId
For(index = 0 to window length)
        Sum += Ch_array[13*index + ChannelNum]
```

Another costly operation to complete on the GPU is the sqrt function which calculates the square root of an expression. This function was required for the calculation of the waveform lengths. According to NVIDIA's CUDA Best Practice's Giude, this function is difficult for the GPU to perform with double precision [18]. The sqrtf function performs at just single precision accuracy but is not nearly as costly. Testing showed that this loss of precision did not result in lower prediction accuracy. Similarly, several absolute value functions that calculate the absolute value of a floating point number were changed to the single precision version without any reduction in accuracy. The change of the square root function reduced the feature extraction time by 124µs to 753µs while the change to the single precision absolute value function resulted in an additional reduction of 7us to 746us.

During this process it was discovered that some memory accesses could actually be eliminated altogether. For some features, such as the number of zero crossings and the number of slope changes, three memory accesses were being made for each iteration of the loop. The current data point, next data point, and the next point after that were each being accessed each time and stored in a variable. This process was reduced by simply reassigning the value of the next point to the current point, and the value of the next next point to the next point instead of reading these values from memory. This reduced the number of accesses from three to one. The absolute values of each of these points were also being calculated each time and the same process was used to reduce the number of absolute values that had to be calculated each time from three to one as well. Lastly, some of the conditional statements used for these features issued yet another read instead of using the value

stored in the local variable. These accesses were eliminated by instead using the variable which is stored in a register. The new feature extraction time after these changes was found to be 421µs, a reduction of 325µs or 44%. A few more changes were made to the feature extraction portion of this program such as changing divides to multiplies, changing how data was copied to the gpu so as to reduce the number of parameters that had to be passed to a kernel, and how some addresses were calculated. All these changes resulted in a very minor reduction to the final result for this version of the program with a feature extraction time of 410µs. This new version of the program averaged 798µs per prediction.

At this point, the test data was also run on the 7-class system. This increased the number of classifiers required for the SVM classification from six to twenty one. The result was an increase in the classification time from 216µs to 260µs for a total time of 842µs. Several changes to the classification implementation were attempted, but only the change of the kernel dimensions, number of blocks and threads per block, and the change of the exp function to expf had any effect and this was but minor. These changes reduced the classification time by 7µs to 253µs for the 7-class version. Thus for the 7-class system, the final version of the original code parallelized with as many GPU optimizations as could be found resulted in a final per prediction time of 835µs, 410µs for the features, 253µs for the classification, 37µs for data copying, and 135µs for operations on the CPU.

5.3 Reduction Method

The next step was to try to decrease the time required for the feature extraction by changing the way that the features were calculated. A good example of how the features were calculated is the mean absolute value feature. A loop was created that iterated through each data sample in the current window. For each data sample the sample was accessed in memory, the absolute value was calculated, and the then the value was added to a cumulative sum. This was a serial process. To fully take advantage of the parallel nature of the GPU the reduce algorithm was used instead.

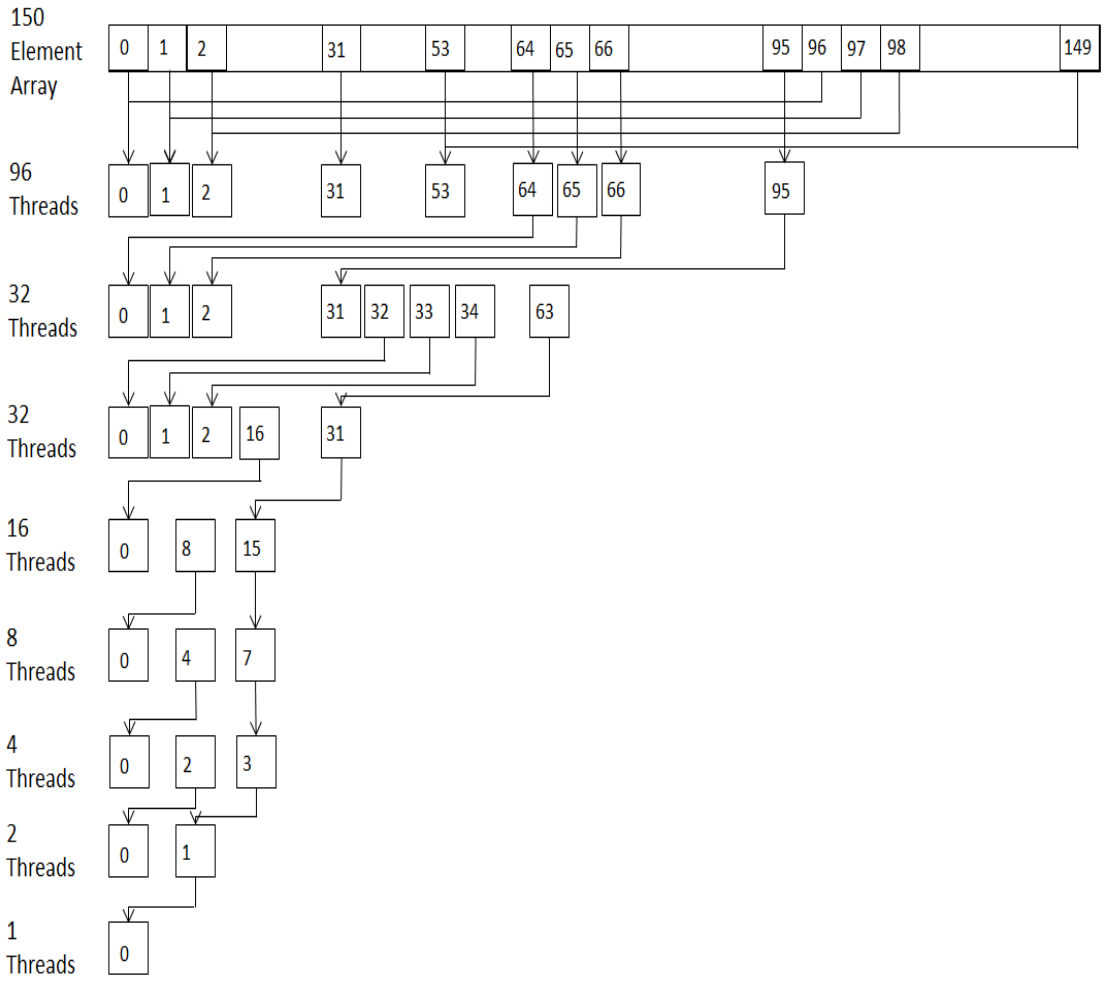

Figure 8: Reduce Method on a 150 Element Array.

The reduce method was found in the example programs that come with the CUDA development toolkit [19]. Using the reduce method a kernel is launched with enough threads to access half of the values in the current window for the given window. Each thread accesses a value in the first half of the window. It then accesses a value in the second half of the data set and adds this value to the first value that was read. At this point all of the values that have been calculated so far are moved to shared memory. Shared memory is memory that is shared by all threads in a block. Accesses to it are almost as fast as accesses to registers, but there is only a limited amount of it available and the data stored in it only remains valid for as long as the current kernel is executing. The first quarter threads now add their value to the value that is in the second quarter threads. This process continues until the complete sum resides in the first thread. This process can also be extrapolated to work for finding all of the features required. An illustration of this method is provided in fig. 8.

The advantage of this method comes from the parallelization of memory accesses and arithmetic operations. In this method memory accesses are coalesced very well as all the threads in a warp read at the same time from adjacent memory locations. Also, with this method all memory accesses for a given channel can happen in parallel. Each time the values are folded upon each other the required operation can be done in parallel. For this study the window size used resulted in 150 sample per window. The kernel dimensions that were found to result in the best performance used 1 block with 96 threads for each channel. This means that the first 96 values are read in parallel. Then 54 more values are read in parallel and added in parallel to the original values that were read in. Next the first 32 threads add their value to the last

32 threads so as to result in 64, a power of two, active threads. After this the active threads are the first 32, then 16, 8, 4, 2, 1. In this manner what once was 149 serial add operations is now just 8 and what was 150 serial global memory accesses is now just 2.

For this method to work the threads that are doing the reduction for a given channel must be highly synchronized. If any one thread gets ahead of another thread then it may try to use the value from the other thread before it is ready. Since all threads in a warp share the same PC this is not a problem for them. Since threads in the same block can be synchronized with a function call they can also be used. However, there is no way to synchronize threads in different blocks. This means that if more than one block is used for the same channel then each block is responsible for only a portion of the channel and a second kernel has to be used to add up the values calculated by each block. In this study it was found that 96 threads per block resulted in the best performance. This meant that only one block per channel was needed.

Since a slightly different kernel was needed for each feature it was no longer possible to use just one kernel launch for the entire feature extraction process. Instead the kernels were broken up into several kernels and concurrent launches were used to maximize the parallelization. First a kernel that calculates the means for all of the channels is launched to stream 1. At the same time a kernel that calculates the max and min values of the mechanical channels is launched to stream 2. Since the mechanical channels do not need to have a DC offset subtracted from them there is no need for these channels to wait for the means to be calculated. A kernel is then launched to stream 1, thus forcing it to wait for the completion of the mean

calculation, which subtracts the mean from each data sample of the EMG channels as well as storing the mean feature for the mechanical channels. Another kernel is then launched to stream 1 that calculates each of the EMG features. Lastly, a kernel is launched to stream 1 that accounts for any work that must be done based on the number of blocks per channel and also loads the features into the correct place in the features array that will be used by the classifier.

As was expected the reduction method was very successful in increasing the performance of the feature extraction process. The results of this new version of the program saw a feature extraction time of 57µs per classification. This was a reduction of 353µs or 86% over the original method. It was also determined that the values calculated for the use of the classifiers kernel function could also be calculated using this reduction method. The result was a reduction in the time of the classifier time by 71µs or 25% to 181µs.

Applying the reduction method to the classifier itself and not just the SVM kernel function was also useful in reducing the total time. For a seven class system there are 21 classifiers that have to run. Each of these classifiers uses the values created using the RBF kernel function and coefficients from the model to calculate two sums that are used to make the classification. These sums were previously found using two for loops. Each loop iterated through 46 values, the 46 values calculated from the 46 features. By instead using the reduction method with a block size of 64 threads the total time for classification was reduced from 181µs to 131µs.

There was some waste with the reduction method for the RBF kernel functions though. Since there are 64 threads per block but only 46 elements to sum there are 18

threads per block that do nothing. These values have to be calculated for each support vector in the model. Since one block was used per support vector and the number of support vectors used in the model varied from about 100 to 400 depending on the phase this is a lot of idle threads. Furthermore, most of the work is only done by the first warp of 32 threads. This leaves another 14 threads that only do a little work and then are left idle. Instead it was determined that a better approach would be to use a block size of 96 threads, which is equal to the total number of CUDA cores in the Geforce Gt 540m, and have each block run calculations for 3 support vectors. This meant that each warp in the block was responsible for 1 support vector. This results in a reduction in the total number of blocks required and the total number of threads. At the same time all of the threads have more work to do and do not spend a lot of time idle. The result of this new method was that the execution time for the classification portion of the program was further reduced to 75µs. By also eliminating some print statements that had accidently been included in the GPU version of the program the final overall time for the 7-class application was found to be 200µs.

An example kernel that calculates the mean of a channel using the reduction method is provided on the following page. This is an example of the code that one thread would execute. In this example the data for all channels is stored in an array in global memory. Each set of 96 threads is responsible for processing the 150 data points in the array that correspond to their channel number.

Reduction Method Example

```
/*Use the reduction method with 96 threads and a window of 150 data points to find the mean of a channel*/
Find the channel number based on the block ID
/*Read the value from global memory that corresponds to this thread and this channel*/
Sum = Ch_array[ChannelNum*150 + threadId]
/*If this thread is one of the first 54 threads then add the value at the index location that is 96 points away from the original location*/
If(threadId < 54)
        Sum += Ch_array[ChannelNum*150 + threadId + 96]
/*Store the data in shared memory*/
SharedMemory[threadId] = Sum
/*Now add all of the values together*/
If(threadId < 32)
        SharedMemory[threadId] += SharedMemory[threadId + 64]
        SharedMemory[threadId] += SharedMemory[threadId + 32]
        SharedMemory[threadId] += SharedMemory[threadId + 16]
        SharedMemory[threadId] += SharedMemory[threadId + 8]
        SharedMemory[threadId] += SharedMemory[threadId + 4]
        SharedMemory[threadId] += SharedMemory[threadId + 2]
        SharedMemory[threadId] += SharedMemory[threadId + 1]
ChMeans[ChannelNum] = SharedMemory[0] / window length
```

Using this implementation for the 4-class system resulted in an average feature extraction time of 55µs and a classification time of 48µs with a total time of 175µs per prediction. These results are as expected. The feature extraction time should not change because the number of features and channels remains the same. The classification time does decrease significantly though. This is because there are only 6 classifiers that need to run in a 4 class system. Also, generally each model requires less support vectors to define the separating hyper-planes between only 4 classes instead of 7.

5.4 Results

This last measurement for the 7-class system is 7.4 times faster than the original GPU implementation, 55.55 times faster than the DSP, 4.3 times faster than the low power CPU, and 2.05 times faster than the i7-2720QM CPU. The 4-class implementation is similar in that it is 50.82 times faster than the DSP and 2.08 times faster than the CPU. The ability to parallelize large portions of the program makes the

Method	Extraction Time (µs)	Classification Time (µs)	Total Time (µs)
Original	410	253	835
Reduce	55	75	200

Table 2: Summary of Improvements using Reduction Method in a 7-class System.

GPU a powerful option when it comes to attempting to decrease the execution time of a program. Its slower serial execution means that as much of the program as possible must be parallelized. However, when this can be accomplished it can perform at a much better rate. Expansion of the program to use more channels, more features, or a larger sample window would probably only increase the performance advantages of the GPU over the CPU.

Chapter 6

Results Comparison

The results show that for this application, using the given architectures the Geforce Gt 540m has the lowest execution time while the i7-2720QM CPU has the second fastest execution, the Atom CPU the third fastest, and theTMS320C6713 DSP the slowest execution time. A full listing of the results can be viewed in tables below.

Architecture	Extraction Time (µs)	Classification Time (µs)	Total Time (µs)
CPU	225	139	364
DSP	6350	2490	8880
GPU	55	49	175

Table 3: Summary of Best Results for a 4-Class System.

Architecture	Extraction Time (µs)	Classification Time (µs)	Total Time (µs)
CPU	225	185	410
DSP	6240	4830	11110
GPU	55	75	200

Table 4: Summary of Best Results for a 7-class System.

To better understand these results it is helpful to look closer at the comparisons between the three architectures and see how each performed given the application.

6.1 DSP vs CPU

System	Extraction Time Slowdown	Classification Time Slowdown	Total Time Slowdown
4-Class	28.22	17.90	24.39
7-Class	27.73	26.07	27.09

Table 5: DSP Slowdown Relative to CPU.

For this application the DSP cannot keep up with the execution speed of the CPU. The CPU simply has too many advantages over the DSP for this particular application for the DSP to overcome. The first disadvantage faced by the DSP is its slower clock speed. This does not allow it to perform nearly as many serial operations in a similar time frame to the CPU. The DSPs chance of overcoming this comes with its ability to perform multiply-accumulate operations in a fast manner using the multiple arithmetic units that comes with the device. However, this application does not have enough operations of this type to be able to take complete advantage of this feature. Similarly, this application requires many branch instructions which are a performance problem for the DSP. The DSP does not have any sort of branch prediction and thus each time that a branch is used the pipeline is forced to stall.

As can be seen from the DSP vs CPU table the DSP executes a prediction at a rate about 24 times slower than the CPU for a 4-class system and 27 times slower for a

7-class system. The increase to 7-classes increases the number of support vectors and the number of classifications that must be made. This causes the DSP to perform at an even worse rate relative to the CPU.

6.2 DSP vs GPU

System	Extraction Time Speedup	Classification Time Speedup	Total Time Speedup
4-Class	115.45	52.08	50.82
7-Class	113.45	64.49	55.55

Table 6: GPU Speedup Relative to DSP.

The DSP is even slower compared to the GPU than compared the CPU. Both architectures face similar disadvantages as compared to the CPU. Both have slower clock speeds and don't have such optimizations as branch prediction. Both architectures have smaller caches and face increased memory access times. However, the difference between the GPU and the DSP is that the GPU can take greater advantage of its strengths for this application than the DSP can.

As mentioned in the previous section the DSP has trouble performing enough operations in parallel for this application to be able to keep up with the CPU. On the other hand, using the reduction method the GPU is capable of surpassing the CPU. The GPU can also reduce its memory access problems by performing the memory accesses in parallel, something the DSP cannot do. The GPU is also able to break down the program using threads in such a way as much of it can be executed in

parallel. Its ability to accomplish this is based on its ability to execute any instruction in parallel not just arithmetic operations. The result is that the GPU outperforms the DSP by 50 times for a 4-class system and 55 times for a 7-class system. Similarly as to when comparing against the CPU the DSP performs at an even worse rate with the 7-class system. This is because the 7-class system requires even more work for the DSP. With the GPU a lot of this work can be performed in parallel and thus does not lead to as large of a relative execution time increase.

6.3 GPU vs CPU

Method	Extraction Time Speedup	Classification Time Speedup	Total Time Speedup
Original 4-Class	.238	.644	.247
Improved 4-Class	.549	.644	.439
Reduce 4-Class	4.09	2.84	2.08
Improved 7-Class	.549	.711	.485
Reduce 7-Class	4.09	2.47	2.05

Table 7: GPU Speedup Relative to CPU.

The parallelized nature of the GPU does allow it to be able to compete with the CPU in terms of execution time. Due to the disadvantages that the GPU faces in terms of serial execution it can only keep up with the CPU in those situations in which enough of the work is done in parallel. The evidence of this is in the results from the

first method that was used to implement the SVM algorithm. In this first attempt the parallelization came from performing the different channels and features in parallel. Although this allowed a large number of operations in parallel it was not enough. Even with the code arranged such as to take advantage of the characteristics of the GPU the execution time was still twice as long as was required by the CPU. However, when using the reduce method not only were the channels and features done in parallel, but the calculations for these operations were also parallelized. This lead to a large reduction in the amount of serial operations performed by the GPU and allowed the GPU to execute the program in half the time that the CPU took.

The GPUs advantage over the CPU slightly decreased when using a 7-class system versus a 4-class system. This is likely because with the 7-class system not all of the calculations could be performed in parallel. Since there are only so many CUDA cores in the GPU it is not possible for everything to actually run in parallel. There has to be some sharing of resources and switching of control of these resources. Even with the 4-class system not all of the work could be done in true parallel. Some higher end GPUs include many more cores and might be capable of performing a lot more of the work in parallel than the one used in this study. This was not true of the first implementation method. This is because with the original method not all of the CUDA cores were being utilized for the 4-class system and thus the 7-class system actually allowed for a greater amount of parallelization than the 4-class version. When using the reduction method many more calculations are performed in parallel, thus using more CUDA cores and making even the 4-class version utilize all of the cores.

Also of note from these results is that the speed ups of the sections of the program that are implemented using the GPU are greater than the overall speedup achieved with the GPU. This is due to the need to copy data back and forth between the CPU and GPU. This added overhead means that even though the GPU can improve upon the performance of some sections of the program, it also has drawbacks. If there is too much data that has to be copied between the two devices then the advantage of using the GPU will be reduced or even eliminated. In this case the amount of copying between the devices was not enough to outweigh the benefits.

Chapter 7

Conclusions

This work took an existing implementation of an SVM application and ported it to several different architectures. The program was then changed to be able to best utilize the advantages of these architectures. This allowed for a comparison of the suitability of the different architectures for this SVM application. It also provided insight into which techniques provided the best performance improvements for each of the architectures.

It was found that the DSP does not match the execution time of the CPU for this application. The DSPs performance can be improved upon by using such methods as enabling L2 cache, reducing the number of memory accesses, and eliminating branches. However, in this case the application does not meet the purposes of the DSP closely enough to be effective. The DSP was designed to be able to perform convolution and other such operations at a high rate. This was accomplished by parallelizing arithmetic operations through the use of multiple arithmetic units. The SVM application used in this work does not have enough of these operations to be able to take full advantage of this design. Similarly, the DSP is at a disadvantage when performing other operations such as branches. In the case of this algorithm there are many cases where avoiding the use of these operations is not possible. The combined inability to both take advantage of the DSPs strengths and avoid its weaknesses makes the DSP unsuited for the application relative to the other architectures that were explored.

The GPU, on the other hand, is able to outperform the CPU. The GPU is designed to allow for the parallelization of many operations. This means that as long as the program can be parallelized the GPU has the opportunity to improve upon the execution time of the CPU. In this case it was found that the application could be broken up into different sections that could be run simultaneously. Furthermore, it was found that the operations that took part inside of these sections could themselves be performed in a parallel manner. This allowed for much of the required processing to be done in an efficient manner. The ability to take advantage of these strengths caused the GPU to be the architecture that was able to achieve the lowest execution time for the chosen SVM application.

There are several things that could still be done to further this work. One would be to use other DSPs and GPUs than the ones presented in this paper. Both of the devices used in this study were approximately in the middle range for performance for their respective architectures. The use of higher performance devices for these architectures may have an effect on the results. Also, it would be interesting to attempt to implement the algorithm on several other architectures as well, such as an FPGA. This would allow for the development of an architecture specifically designed for the application. Another change could be made to the CPU implementation. The current CPU application is not multi-threaded and thus it may be possible to improve performance on the CPU itself.

Bibliography

[1] R. Hernandez, F. Zhang, X. Zhang, H. Huang, Q. Yang, "Promise of a Low Power Mobile CPU Based Embedded System in Artificial Leg Control", *Conf Proc IEEE Eng Med Biol Soc*, 2012

[2] Frantz, G.; , "Digital signal processor trends," *Micro, IEEE* , vol.20, no.6, pp.52-59, Nov/Dec 2000

[3] Eyre, J.; , "The digital signal processor Derby," *Spectrum, IEEE* , vol.38, no.6, pp.62-68, Jun 2001

[4] Frantz, G.; , "Signal Core: A Short History of the Digital Signal Processor," *Solid-State Circuits Magazine, IEEE* , vol.4, no.2, pp.16-20, June 2012

[5] Turley, Jim and Hakkarainen, Harri, TI's New 'C6x DSP Screams at 1,600 MIPS. *Microprocessor Report* (February 17, 1997), pp. 14-17.

[6] Dragos Nicolae Vizireanu: Quantised sine signals estimation algorithm for portable digital signal processor based instrumentation, *International Journal of Electronics*, 96:11, 1175-1181 Available:

[7] Javeed Ahmed Khan, S. Ravichandran & K. Gopalakrishnan (2010). "Cellular Neural Network on Digital Signal Processor: An Algorithm for Object Recognition", *Electric Power Components and Systems*, 38:10, 1111-1122

[8] P. Trancoso and M. Charalambous. "Exploring Graphics Processor Performance for General Purpose Applications" in *Proceedings. 8th Euromicro Conference on Digital System Deisgn* Aug. 30 2005-Sept. 3 2005, Porto, Portugal, pg. 306-313.

[9] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. "GPU Computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879-899, May 2008.

[10] Huang H, F Zhang, L Hargrove, Z. Dou, D Rogers, K. Englehart. "Continuous Locomotion Mode Identification for Prosthetic Legs based on Neuromuscular-Mechanical Fusion", *IEEE Trans Biomed Eng*, 58(1), pp 2867-75,2011

[11] Cortes C, Vapnik V. "Support-vector networks". *Mach Learning*. 1995;20:273–297.

[12] Robert Hernandez, Qing Yang, Jason Kane, Fan Zhang, Xiaorong Zhang, He Huang. "Design and Implementation of a Low Power Mobile CPU Based Embedded System for Artifical Leg Control", *Submitted to International Conference on Cyber-Physical Systems*, 2013

[13] M. Saghir, P. Chow, and C. Lee. A comparison of traditional and VLIW DSP architecture for compiled DSP applications. In *CASES '98*, Washington, DC, USA, 1998.

[14] Texas Instruments (2001, Revised 2005). "TMS320C6713 Floating-Point Digital Signal Processor Datasheet" [Online]. Available: http://www.ti.com/lit/ds/sprs186l/sprs186l.pdf [September 28, 2012].

[15] *TMS320C6713 Technical Reference*, Rev B, Spectrum Digital Inc., Stafford, TX, 2004.

[16] "Geforce Gt 540m Specifications." Geforce.com. Nvidia Corporation. http://www.geforce.com/hardware/desktop-gpus/geforce-gt-540m/specifications [July 14, 2012].

[17] NVIDA "NVIDIA's Next Generation CUDA Compute Architecture: Fermi" 2009, Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf [September 28, 2012].

[18] *NVIDIA CUDA C Programming Guide,* Ver. 4.2, NVIDIA Corp., Santa Clara, CA, 2012. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf [September 28, 2012].

[19] "CUDA Toolkit 4.2". *CUDA Downloads*. NVIDIA Corp. Web. http://developer.nvidia.com/cuda/cuda-downloads. [October 12, 2012].

CUDA C Best Practices Guide, Ver. 41, NVIDIA Corp., Santa Clara, CA, 2012. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf.

Delves, M., Tetley, D.. "Using Intel® Processors for DSP Applications: Comparing the Performance of Freescale MPC8641D and Two Intel Core™2 Duo Processors". http://www.nasoftware.co.uk/home/attachments/018_PPC_Intel_comparison_whitepaper.pdf.

Evaluating DSP Processor Performance, Berkeley Design Technology Inc., Berkeley, CA, 2002. Available: http://www.bdti.com/MyBDTI/pubs/benchmk_2000.pdf.

Huang, S., Xiao, S., and Feng, W.,. "On the energy efficiency of graphics processing units for scientific computing". In *IPDPS*, 2009.

Luebke, D. “Cuda: Scalable parallel programming for high-performance scientific computing.” *Biomedical Imaging: From Nano to Macro, 2008. ISBI2008. 5th IEEE International Symposium on*, pages 836-838, May 2008.

Manavski, S. A, “CUDA compatible GPU as an efficient hardware accelerator for AES cryptography.” In *ICSPC 2007: Proc. of IEEE Int’l Conf. on Signal Processing and Communication*, pages 65–68, 2007.

Nickolls, J., Buck, I., Garland, M. 2008. “Scalable Parallel Programming with CUDA.” *ACM Queue*, 6, 2, 40-53.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., And Hwu, W. W. 2008. “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA,” *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM Press, 2008, 73–82.

Trancoso P, Charalambous M. “Exploring graphics processor performance for general purpose applications,” *Euromicro Symposium on Digital System Design, Architectures, Methods, and Tools (DSD 2005)*; 2005.

Wan, X, Xiong, W, Zhang, Z, and Chang, F, "An online emission spectral tomography system with digital signal processor," *Opt. Express 17*, 5279-5284 (2009).