IMPLEMENTING A DEFORMABLE MODEL IMAGE SEGMENTATION ALGORITHM FOR A MULTI-CORE MICROPROCESSOR ARCHITECTURE

A Thesis
In STS 402

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment
of the Requirements for the Degree

Bachelor of Science in Computer Science

by

Adam Banda
March 29, 2007

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for papers in Science, Technology, and Society Courses.

Signed____

Approved ____ Date ______________
Prof. Kevin Skadron (Technical Advisor)

Approved ____ Date ______________
Prof. Brian Pfaffenberger (STS Advisor)

PREFACE

I first got involved with research in parallel computing in the summer of 2007, helping Prof. Kevin Skadron with his research. Parallel computing certainly is not a new development in computer science, having been used for high-performance research for decades, but is just recently become a major issue in mainstream computing. Current CPU architectures are beginning to reach their maximum heat limits, so major manufacturers, such as Intel, have been scrambling to find other design solutions such as parallel, multi-core architectures for their mainstream products. As such, computer architecture design and parallel programming have become very exciting areas within computer science to study.

Additionally, I have done cellular image processing research with Prof. Ammasi Periasamy, which has shown me the critical importance of computing in biomedical research. Some extremely essential medical research about cancer and other diseases require much effort by computer scientists to provide the fastest, most robust tools to analyze research data. I chose this project because it attempts to create better performing algorithms that can be used for medical imaging on a revolutionary, new multi-core platform.

I would like to acknowledge and thank the following persons for their help and contribution to this project:

- Kevin Skadron
- Edmund Russell
- Brian Pfaffenberger

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Deformable model image segmentation is the process of mathematically modeling three-dimensional flexible, or non-static, objects, such as human organs, onto a computer through the analysis of multiple images of the subject from a variety of different angles and directions. This type of image segmentation is used widely in the medicine, commerce, and defense fields. Some examples of applications include surgical head-up displays providing an enhanced view of a patient's body, intelligent character recognition systems for digitizing documents, and military missile and aircraft tracking techniques. The deformable model image segmentation process can be incredibly intensive and time-consuming on current, traditional, single-core computer micro-architectures and generally require expensive high-end mainframes or distributed systems. Aside from the obvious price deterrent, these types of computers are generally quite cumbersome and immobile making them impractical for many applications. Multi-core processors such as NVIDIA's GeForce series, however, show a great deal of promise by providing high-end computational power while remaining as portable as personal computers. This project concentrated on implementing a deformable model segmentation algorithm for one of NVIDIA's processors, attempting to achieve real-time processing capability. Although this goal was not reached, a 1.24x speedup was achieved showing a great deal of promise for parallel programming and multi-core processors. Finally, as more mainstream processors begin to use multi-core designs, making parallel programming take on a larger role in general computing, new tools will have to be created to help software designers create parallel software as efficiently as in the past.

CHAPTER 1: INTRODUCTION

Many important applications used for health care, defense, and many other industries require algorithms that are able to match, track, and model objects by analyzing photographs or images. This chapter introduces the type of algorithm that will be used in this project, discusses the need for improved implementations, and describes the scope and organization of the project and this report.

1.1: Introduction to Deformable Model Image Segmentation

Image segmentation is a process used to distinguish objects within images, such as photographs, radar outputs, or x-rays, from their background. For example, given a microscopic image of a blood sample, an image segmentation process could be used to locate and identify all blood cells in the image, recording each cell's position in the sample and even categorizing its blood cell type (Asano, Chen, Katoh, & Tokuyama, 1996). Image segmentation has advanced to a point where it can even match highly textured images as shown in Figure 1. The cells would be stored on a computer as a mathematical model, keeping track of their size and shape. Image segmentation methods can also be used to find and store three-dimensional objects by analyzing multiple images of the same objects from different angles. A challenge, however, arises when the objects being scanned for are not static—in other words, objects that are not always the same shape. For instance, finding and modeling a human liver can be difficult because it has a different shape depending on the specific subject and their age. Deformable model image segmentation is the technique that enables the matching of non-static three-dimensional objects from two-dimensional images (Baker, 2005) as shown in Figure 2.

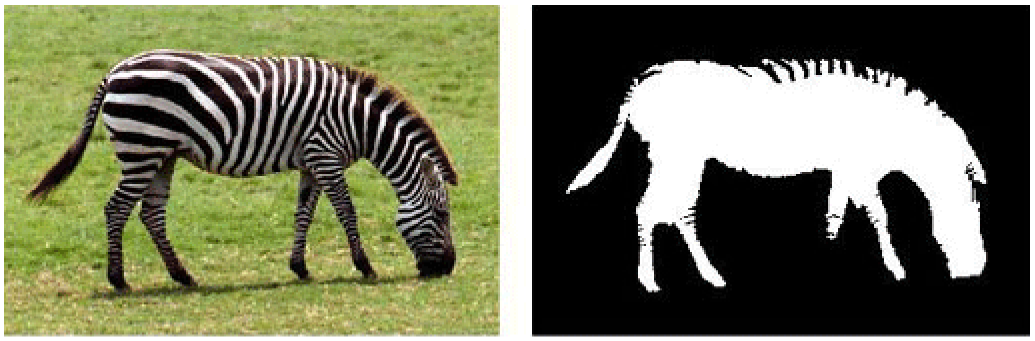
Figure 1: Image Segmentation to match object (Kompatsiaris, 2004).

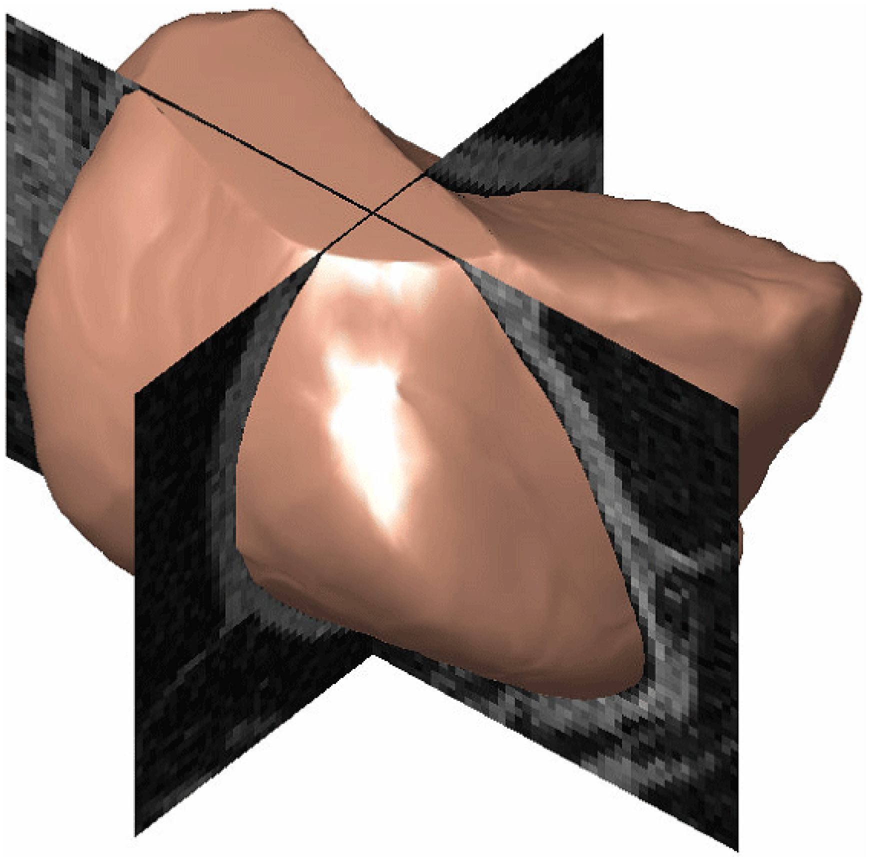
Figure 2: 3D model created from two 2D cross-sections (Li & Acton, 2006).

1.2: Needs for a Better Performing Implementation

Current deformable model algorithms implemented for single processor architectures are quite processor-intensive and can take many billions of cycles to execute accurately (Lachaud & Taton, 2005). As a result, the run time of these algorithms can be too long to be used for real-time segmentation applications, where the algorithm must run multiple times per second. Many different fields, including health care and defense, require these real-time image segmentation implementations to match and model objects (Grinstead, Koschan, Page, & Abidi, 2006). New advances in computer processing architectures create opportunities to implement much faster algorithms achieving or coming close to real-time applicability, such as an implementation of a complex Feldkamp algorithm by IBM researchers (Sakamoto et al., 2005). Due to their parallel processing architectures, both graphics processing units (GPU) and IBM’s Cell Broadband Engine Architecture (Cell) are of particular interest for image processing use. Current GPUs feature over a hundred processing units, every one able to analyze data concurrently with each other; this makes them extremely well suited for image processing (Owens et al., 2005). The goal of this project is to implement a current deformable model image segmentation algorithm for a GPU that can calculate high-quality computer models of objects from detailed images fast enough to allow for real-time applications.

Deformable model image segmentation has a long history of research and development and has been used in many different applications (Gibson & Mirtich, 1997). However, real-time applications generally remain extremely costly and immobile, requiring expensive high-end mainframes or a powerful distributed system. This is a major roadblock for many segmentation applications such as surgical display aids and onboard aircraft tracking on military vehicles, which cannot meet the space requirements for a mainframe. For these types of developments to be feasible, segmentation time must be reduced to real-time speeds on a portable micro-architecture such as a GPU. To accomplish this, the segmentation algorithms must either become more efficient, or processing throughput must increase (Lorts, 2000). Much work has gone into these issues; current algorithms are highly optimized for traditional single-core processors. This project will attempt to create an optimized segmentation algorithm for a GPU that fully utilizes the architecture's performance enhancing features such as high-speed vector processing and parallel data partitioning.

1.3: Organization and Scope of the Thesis Project

The segmentation algorithm used for this project was written by Bing Li and Scott T. Acton of the University of Virginia (UVA) (Li & Acton, 2006). Adam Banda carried out the rest of the project, converting the algorithm for use in a multi-core environment and optimizing the application to achieve greater performance. The implementation was created at UVA's Computer Science department starting in the summer of 2006 and continuing until this project document's publication date.

This project report will disclose the multi-core implementation's creation process, analyze its results, and give recommendations for future research. First, it will discuss the use and impact of the algorithm in society and the motivation for an improved solution. Next, it will talk about the technology behind the project and some current, related research. Then, the report will precisely describe the entire process of creating the new implementation, detailing the problems encountered and lessons learned. The report will then reveal, analyze, and discuss the test results including the implementation's speed and accuracy relative to the original. Finally, it will disclose the conclusions of the project and make recommendations for future research.

CHAPTER 2: SOCIETAL CONTEXT AND ETHICAL IMPLICATIONS

First, this chapter will discuss deformable model segmentation algorithms' use in object matching, modeling, and tracking applications in today's society. These types of computer-aided object managers are essential in modern society and are used in numerous applications. They benefit numerous areas including culture, economy, health care, and defense. This chapter will then go on discuss some of the ethical issues of this project including contracts with NVIDIA, the manufacturer of the hardware used in this project, and uses of related algorithms for military purposes.

2.1: Object matching, modeling, and tracking applications in society

The first use of object tracking is for general computing. Image and video retrieval are important as computers gain more storage and processing capacity allowing for vast collections of media files. New object tracking systems are being developed to be able to track, organize, and compare large collections of media objects (Aslandogan & Yu, 1999). Also of importance is character and handwriting matching. As computers become more powerful, the ability to transform writing to digital media is becoming easier (Belongie, Malik & Puzicha, 2002). More accurate methods of character recognition are now being used, such as object matching systems using deformable models (Cheung, Yeung & Chin, 2002).

Another use of object tracking is in the defense industry. A strong and secure homeland requires the military and intelligence communities to have the ability to track and analyze objects across the globe such as aircraft and vehicles. Computers can be used to aid the defense industry by analyzing photography and radar imagery for objects.

Object matching algorithms can be used to find and map roads from radar images. Furthermore, the military can use this ability to automatically control and steer vehicles, even through darkness, fog or snow (Kaliyaperumal, Lakshmanan & Kluge, 2001). Defense communities can also use these algorithms for detecting the location and bearing of actual vehicles and aircraft. For example, infrared images of the sky can be taken, either from the ground or from mounted cameras on commercial or military planes. From these two-dimensional images, objects can be recognized and modeled. Additionally, these methods can determine the aircraft’s type, velocity, direction, and most importantly, whether or not it is a target (Kamgar-Parsi, Jain & Dayhoff, 2001).

In addition to its defensive uses, object matching can be used to translate hand and face gestures into digital form for analysis. The modeling and analysis of human motion can be used for many different applications. Examples include gesture-driven user interfaces, motion analysis in sports and medicine, psycholinguistics, surveillance, and entertainment (Bryll, Rose & Quek, 2005). In addition, accurate translation of the hand and lip signs of the deaf can be transferred to digital form through object modeling techniques.

Deformable object modeling is also very important in a variety of other fields as well. It can be used, for example, in computer-aided apparel design where computer models can simulate fabric folding and draping. In the entertainment industry, these models can be used for the animation of complex objects like clothing or facial expressions (Gibson & Mirtich, 1997). Deformable object modeling can also be used to expand culture. For instance, researchers at the University of North Carolina have developed a system utilizing deformable modeling that lets artists paint directly to a digital medium. The system uses the artists force and tactile information along with a physically-based deformable brush model to created digital images very similar to their paint-and-canvas analogs (Mahoney, 2006).

Finally, deformable object modeling is of tremendous importance in the health care industry. Medical imaging is revolutionizing the medical industry. It enables doctors to look inside the human body noninvasively. It can also be used to plan surgery or help eliminate tumors with minimal collateral damage (McInerney & Terzopoulos, 1996). Some examples of deformable object modeling applications include medical diagnostics, preoperative planning, intraoperative navigation, surgical robotics, training, and telesurgery (Shadidi, Tombropoulos & Grzeszczuk, 1998). Additionally, using deformable models, an image-enhanced endoscopy system was created for surgeons. The system used a transparent display overlaying the patient to be operated upon. The monitor displays three-dimensional models of the organs and area of the body to be operated on and uses deformable models to calculate and calibrate the images. This system enables the surgeon to streamline the procedure by constantly giving him all the necessary information directly in his line of sight (Shahidi, et al., 2002). A similar system has also been used to target lesions for surgeons, helping them navigate around critical points. These systems help to achieve a more effective, safe, and streamlined surgery (Liao, et al., 2004).

2.2: Ethical implications of the project

My research involves developing a deformable model algorithm for one of the latest computer architectures. As such, no research with humans will be conducted. Nor will my application be used to directly create a system that is used on humans. My project involves only the manipulation of data in a computer. Therefore no real safety concerns are relevant, especially if proper ergonomic techniques and equipment are used. The equipment to be operated is owned by the UVA's Computer Science department. A variety of sources, including GPU documentation from NVIDIA, will be used for this project. Originally, NVIDIA's development aid being used, CUDA, was unreleased and bound by a nondisclosure agreement (NDA). As such, no information concerning CUDA could be discussed with people not under the NDA, including its feature set, programming model, and even its very existence. However, at the time of this report's publication, the NDA has been dissolved as NVIDIA has publicly released its newest product line featuring CUDA support. All sources used in this project were acknowledged and properly cited. This research was conducted primarily by Banda with some collaboration and discussion with Kevin Skadron.

Since NVIDIA's NDA was upheld, it is doubtful that many other ethical problems will come out of this project. Even though very similar algorithms have been used for military purposes, they are mainly for surveillance and intelligence gathering, aiding in the defense of the nation. Since designing improved body armor and other military hardware is not unethical, neither is creating better algorithms that could be used in a military setting. Deformable model research has been conducted for the past two decades with few problems for any groups of people. Instead, the research has created more accurate and efficient applications in everything from surgical procedures to assembly line optimization. Deformable model research can provide many benefits to a wide range of interests and applications while harming very few.

CHAPTER 3: REVIEW OF TECHNICAL LITERATURE

This chapter first discusses the development of parallel processors and their important role in current and future mainstream microprocessor design. Then it will talk about the particular algorithm used in this project, including the reasons it was chosen.

3.1: Importance of Parallel Microprocessor Architectures

Since I will be developing the segmentation algorithm on a GPU, a multi-core processor, parallel programming will be vital. Parallel processing has long been at the forefront of high-performance computing. The world's fastest computers all use multiple processors along with various parallel programming techniques to be able to compute the most complex and intense algorithms. Traditionally, these machines were restricted to only the largest, most prestigious institutions such as the Lawrence Livermore National Laboratory, which uses IBM's Blue Gene, one of the fastest supercomputers in the world (Zheng, Singla, Unger & Kale, 2002). Advances, however, in both the design and manufacture of chipsets could bring high-performance computing to the commercial world (Smith, Hsu & Hsuing, 1990).

In 2003, Virginia Tech contributed to the evolution of supercomputing when they purchased 1,100 Power Mac G5 desktop computers from Apple Computer. Networked together, these personal computers formed the third best performing computer in the world. Even more impressive was the fact that the system was relatively inexpensive, $5.2-million instead of the tens of millions of dollars usually spent on more traditional supercomputing designs (Olsen, 2003).

In addition to being great for high-performance computing, parallel processing will be essential in the future personal computing market. Processors have doubled in performance about every two years (Lundstrom, 2003, p. 210). Traditionally this was done by increasing the processor's clock speed, the number of operations it can perform per second, and reducing transistor size thereby increasing its performance. Currently, however, some major limitations are being reached (Lundstrom, 2003). Author Patrick Gelsinger has said that "as [microprocessor] designs become more complex, technology scaling more difficult, and power issues more pressing, 'business as usual' [in the microprocessor industry] no longer suffices" (2001). Increases in processor clock-speeds have stalled recently, mostly due to heat issues (Puri, Karnik & Joshi, 2006). To continue the performance increase trend the industry is used to, microprocessor manufacturers must make improvements in other areas.

One way hardware manufacturers, such as Intel and AMD, have continued to increase computing performance without increasing the clock-speed is by increasing the number of cores on a single processor. For example, one of Intel's newest high-end microprocessors, the Core Duo, has two processing cores on a single chip. This allows two processes to run concurrently on a single microprocessor, for instance, the playback of a DVD and the encoding of an email simultaneously (Intel, 2006). With only two cores, full processor utilization is simple because most systems do have two or more processes running at the same time. However, this multi-core trend suggests that future microprocessors will have to house many cores to increase performance, creating an enormous challenge for software engineers in utilizing all of the processing units efficiently.

Many scientific and medical algorithms require high-performance computers to process them in a manageable amount of time. With the introduction of multi-core processors to the consumer market, it could become feasible for researchers, scientists, and doctors to afford portable computers powerful enough to compute their intensive algorithms instead of relying on cumbersome and expensive high-performance mainframes. New multi-core architectures, such as IBM's Cell and NVIDIA's GPUs, could prove very useful for these processor-intensive algorithms. In fact, implementations for various medical imaging algorithms have already been created for both the Cell processor and GPUs. For instance, the Fast Fourier Transform, an essential algorithm in medical imaging, has been implemented successfully for the GPU (Moreland & Angel, 2003). Additionally, J. Greene and R. Cooper have ported the same algorithm to a Cell processor with significant performance increases (2005).

3.2: Algorithm Used in the Project

In this project, Banda has implemented a recent image segmentation algorithm on NVIDIA's GeForce 8 Series (G80) GPU architecture. He used the "Poisson inverse gradient for automatic initialization of deformable model segmentation" method by Bing Li and Scott T. Acton (2006). This algorithm aimed to make automatic modeling of objects more efficient. Much effort has gone into this area, and various different methods for initializing the object model have been attempted and tested (Duan, Yang, Qin & Samaras, 1997). Eduardo Tejada and Thomas Ertl were able to create an implementation of a similar deformable model algorithm for a GPU that was able to run as fast as or faster than traditional, single-core implementations (2005). Based on these results, it was deemed that Li and Acton’s algorithm should be able exhibit similar performance increases when translated to a multi-core processor.

Implementing Li and Acton’s algorithm for use on a GPU was ideal for this project for two reasons. First, the algorithm itself is revolutionary; the Poisson inverse gradient method is completely novel and makes improvements on previous approaches including Neuenschwander, Fua, Szekely, and Kubler’s “Velcro surfaces” deformable modeling method (1997). Second, GPUs have exhibited performance increases at a rate much higher than tradition processing units made by Intel and AMD (Owens et al., 2005) as shown in Figure 3. In addition, the two largest graphics manufacturers, NVIDIA and ATI, have begun to support general purpose programming in their GPUs (GPGPU) enabling researchers and software engineers to more easily implement algorithms on the units.

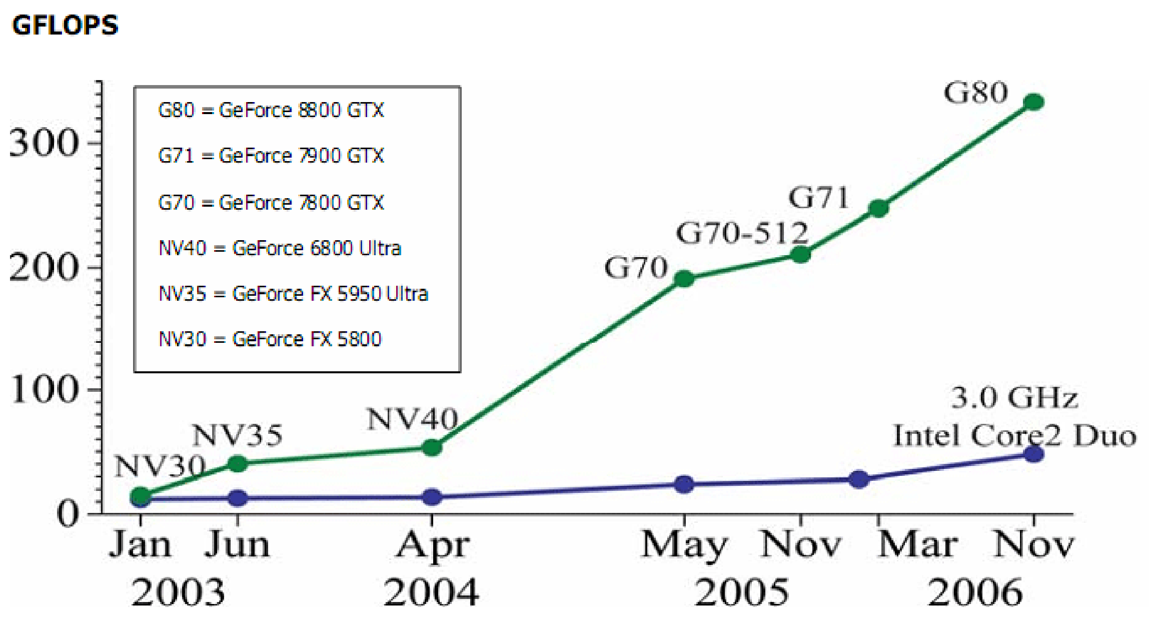

Figure 3: NVIDIA GPU vs. Intel performance over time (NVIDIA, 2007).

CHAPTER 4: MATERIALS AND METHODS

This chapter will focus on the steps carried out to complete the multi-core implementation of Li and Acton's algorithm. It will first describe the research carried out to gain the knowledge necessary to create a well-made implementation. Next, it will discuss the materials obtained and steps taken to set up equipment necessary to complete the project. Finally, it will describe the actual implementation process, including code translation and optimization.

4.1: Preliminary Research

The first step in the project was to research current deformable model image segmentation techniques. To do so, Banda has used reference searching services and library systems to research the latest developments and gain knowledge in overall segmentation methods. Using this information, he was able to study a variety of related algorithms, searching for one that was both extremely intensive and highly parallelizable. At this stage Banda had collaborated directly with Skadron to find the best choice of algorithm for the project. After consultation with the "Poisson inverse gradient" method's creators, Li and Acton, they determined that this algorithm was best suited for the project.

Along with researching segmentation methods, Banda began gaining more expertise in parallel programming techniques. All parallel architectures have some similar attributes, even across completely different systems. These include data partitioning, parallelization, and load balancing. Each was extremely important when finally implementing the algorithm. To gain detailed understanding of these techniques, Banda studied them through a variety of textbooks and articles on the subject. In addition, he consulted with Skadron, who had advanced knowledge and expertise in parallel programming and advanced microprocessor architectures.

4.2: Materials and setup

The next step was to gain access to a workstation and install NVIDIA’s software development kit (SDK). The SDK was used to aid with writing, testing, and emulating programs, even before the necessary NVIDIA hardware was obtained. To find a suitable workstation, Banda and Skadron applied for office space from UVA’s Computer Science department. The SDK software itself was available for free from NVIDIA and was installed on the workstation. The process was aided in part by UVA’s Computer Science System Staff.

After setting up the SDK, Banda became comfortable with the GPU’s programming models. First, he studied the documentation installed with the SDK (NVIDIA, 2007). These files included a wealth of information involving the system customization, program simulation, GPU programming models, and software testing. Then, he further improved his knowledge by writing some sample programs and becoming familiar with the SDK’s components and interfaces. Next, he practiced debugging methods on NVIDIA’s CUDA emulator. Full comprehension of the debugging procedure would be necessary to fully implement and optimize the deformable model segmentation algorithm.

The next step required the profiling of the selected deformable model segmentation algorithm. First, Banda created a flowchart of the algorithm. This tracked the execution trajectory of the program and helped in the overall understanding of the program. By mapping out the algorithm visually, trends in its execution can be found allowing him to pinpoint sections of the algorithm that could be most effectively parallelized for use by the many individual processing cores of NVIDIA’s G80. It will also help to reveal data dependencies between different sections of the program that could cause problems when parallelizing. Next, Banda used profiling tools to measure the runtime of different sections of the algorithm in order to determine where speed enhancements would most drastically improve its overall performance. Li and Acton’s original algorithm was written for MatLab, a numerical computing environment allowing for easy manipulation of matrices such as image arrays (Goering, 2004), so Banda used MatLab’s built in profiling tools to find which parts of the program had the longest runtime. These critical parts of the program would need to be optimized to create the fastest final implementation.

4.3: Project Implementation Process

Now that Banda had obtained and setup all the programs and equipment, he began to translate the algorithm from the MatLab language to the C programming language. This involved the line-by-line conversion of instructions from their original command to the fastest available instruction compatible with C. This step proved quite time consuming, so the decision was made to only translate a subsection of the original MatLab code to C in order to complete the project on time. To determine which parts to translate, Banda again used the MatLab profiler to obtain the results shown in Appendix A. One subsection of the code was found to use over 54.9% of the programs runtime. This part of the algorithm was used to sort a given matrix by its rows. This section made extensive use of sorting functions, and since sorting functions can generally be effectively parallelized for performance gains (Rajasekaran, 2000), these sections of the program were translated to C.

After transforming parts of the algorithm from MatLab to C, Banda began to parallelize the code for the GPU. One of the reasons that GPUs from NVIDIA and ATI have been receiving increased use by researchers is that they have been adding more support for general purpose (GP) programming with each new release. NVIDIA's GPGPU solution, CUDA, whose uses are shown in Figure 4, makes programming GP software much easier by handling much of the internal management such as direct memory transfers itself, letting the programmer concentrate on his implementation. To parallelize, the programmer tells the system how many partitions he needs to process the data; in other words, how many threads he needs. Then each thread runs the same code on different data. The trade-off is that the programmer has less control over the code possibly resulting in less than perfect performance. Since direct memory transfers were handled behind the scenes by CUDA, Banda concentrated on parallelizing the code so that it made use of all 128 individual processors that NVIDIA's G80 Series features.

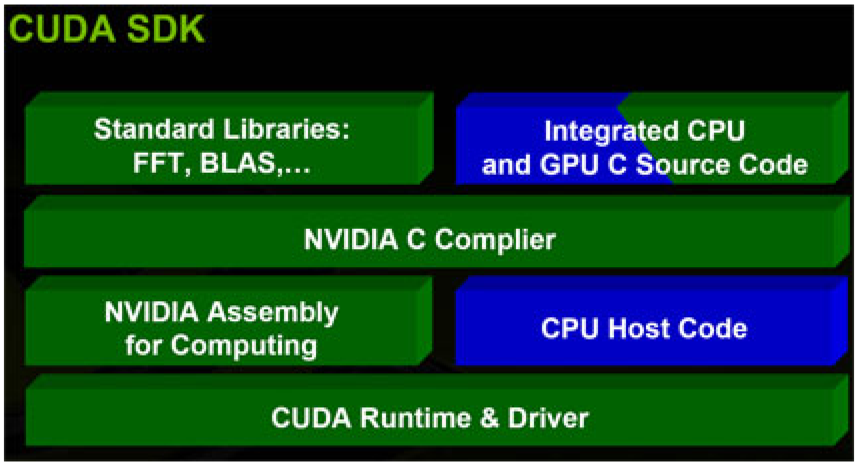
CUDA SDK
Standard Libraries: FFT, BLAS,...
Integrated CPU and GPU C Source Code
NVIDIA C Complier
NVIDIA Assembly for Computing
CPU Host Code
CUDA Runtime & Driver

Figure 4: Uses of NVIDIA's CUDA (NVIDIA, 2007).

Since the sections of the algorithm that he converted to C frequently used a sort function to arrange hundreds—or sometimes thousands—of numbers in order, a robust sort function that took full advantage of all of the GPU's processing elements was needed. In the end, the bitonic sort method was selected. This particular algorithm was chosen for two major reasons. First, it is very easily parallelized. At each stage in the sort process, *n* pairs of data elements are compared, where 2*n* represents the total number of data elements. Each pair is only compared once and the data read or written to is only from those two elements, meaning there is no data dependencies between different pairs within a stage. This simplifies the algorithm allowing separate threads to handle each pair, and no thread needs to communicate with each other. Second, the bitonic search performs very well on multi-core processors with many processing elements. Since this project uses the NVIDIA GeForce 8800 GTX, which has 128 such processing elements, bitonic search was the most ideal algorithm.

Normally bitonic search takes an array of numbers and return a sorted version of the array. However, for this project, a slight modification was needed. Instead of the actual sorted values being returned by the sort method, this application needed the final indices of the sorted array to be returned. This modification was fairly simple to implement. First, an additional array was created, its size being equal to the input array. Then, each element in this new array was initialized to value of its index, i.e. the first was given the value '0', the second the value '1', and so on. Next, whenever a pair of data elements was swapped, the corresponding pair of index elements was swapped too. Finally, instead of returning the data array, the modified bitonic sort would just return the index array. While this modification was simple to implement, it had to use twice as much memory. Fortunately, since the initial index array could be computed on-the-fly and since only the final index array was returned, the modified bitonic sort algorithm only needed the same amount of data to be transferred to and from the algorithm as the original.

One of the main reasons bitonic sort was chosen for this project is that it is very straightforward to translate it from a single-core to a multi-core version. Bitonic sort consists of $\log(n) \cdot (\log(n) + 1) / 2$ stages each stage making $n / 2$ comparisons, where n is the number of elements. The advantage of bitonic sort is that each of these stages has no data-dependencies. Thus, independent threads can be used to process each index, never having to communicate with each other if they are synchronized after each stage. In the original, single-core version of the bitonic sort algorithm, at each stage it would loop through every index of the array serially, comparing and swapping certain values. For the multi-core implementation, each thread would carry out the same method as the single core version with the exception of the looping through each index. Instead, each thread only compared and swapped the two elements of its pair. After every thread completed a stage, or synchronized, they would begin the next stage. The CUDA system provides this functionality with its `__syncthreads()` operation, so between each stage in a thread's code, the `__syncthreads()` function was placed (NVIDIA, 2007).

The final step in the implementation process was to get the multi-core bitonic sort to fully run on the NVIDIA CUDA platform. The NVIDIA GPU hardware places restrictions on the number of threads that can run in the same block, or the number of threads that can communicate by sharing data. However, as previous mentioned, for bitonic sort, each stage requires no communication between threads. Instead a thread must simply wait until all other threads have finished the stage before moving on to the next. Therefore, any thread could be placed into any block of threads and still function the same. The only change needed to be made in a thread’s code was to add the product of the block size and its current block identification number to the thread’s index value allowing it to access the same array index independent of its block. Finally, the implementation was modified to actually send and receive the array data to and from the CPU and GPU and to initialize each thread on the GPU. This was implemented through NVIDIA’s special CUDA functions (NVIDIA, 2007). After some preliminary testing to make sure the new sort algorithm worked properly, Banda could now test his code’s accuracy and speed on the NVIDIA G80 Series hardware.

CHAPTER 5: RESULTS AND DISCUSSION

This chapter will first present the test results of the original and new implementations, namely comparing their runtimes. Then it will discuss the implications of the results and whether or not the project succeeded.

5.1: Test Results

Since only a portion of the original MatLab code was translated and optimized for the GPU, to make testing easier, the plan was to link the GPU sections from the MatLab program so that the entire algorithm could be tested at once. MatLab has a feature that allows it to use outside compilers to convert code written in C to a package that MatLab can run just like any other of its internal functions. Unfortunately, MatLab only supports a few specific compilers while the NVIDIA GPU uses its own special compiler, so Banda could not link all of the code into one package. Instead he had to test the two components separately—first the MatLab section and then the GPU section using data outputted from the MatLab program as the input in the GPU code. Some problems were presented; the sortrows function in MatLab that was converted to the GPU was to be called 26 times by the MatLab program as shown in Appendix A. This would mean that the output from MatLab to be used as input in the sortrows function would have to be calculated and executed 26 times. A better solution was found; since each call of the function used similar input data, and since the MatLab profiler was able to calculate exactly the amount of time spent within the sortrows function, this time could be replaced with the total runtime of 26 calls of the GPU code. This total time would have to include the time it takes to transfer memory from the CPU to the GPU at runtime, since this would have been included in the total runtime of a linked total package.

Figure 5: NVIDIA GeForce 8800GTX (NVIDIA, 2007).

The tests were all carried out on a Dell XPS workstation with an Intel Core 2 Duo 6300 CPU, 1 GB of RAM, and the NVIDIA GeForce 8800GTX GPU, shown in Figure 5. First, the entire MatLab algorithm was tested; the results are shown in Table 1. Next, the total time spent in the <u>sortrows</u> function was tested as shown in Table 2.

Trial #	Runtime (s)
1	1.064
2	1.048
3	1.082
4	1.061
5	1.053
6	1.058
Mean	1.061

Table 1: Matlab full algorithm runtimes.

Trial #	Runtime (s)
1	0.635
2	0.623
3	0.606
4	0.612
5	0.610
6	0.617
Mean	0.617

Table 2: Matlab sortrows function runtimes.

Next, the GPU code was tested. First the actual setup and memory transfer time of the GPU program was tested without doing the actual data processing. To get the most accurate results, the transfer time test was carried out 6 times, as shown in Table 3. Next the full GPU code was tested, as shown in Table 4.

Trial #	Runtime (s)
1	0.015672
2	0.015744
3	0.015744
4	0.015640
5	0.015642
6	0.015628
Mean	0.015672

Table 3: GPU memory transfer runtimes.

Trial #	Runtime (s)
1	0.015869
2	0.015819
3	0.015816
4	0.015942
5	0.015844
6	0.015840
Mean	0.015855

Table 4: GPU code runtimes.

As one can see, the memory transfer time takes up the vast majority of the runtime; in fact, the actual data processing only adds an addition 0.000183-s to the total runtime. Since sortrows in the original MatLab code was called 26 times, the total GPU code runtime must be multiplied by 26. This means the total runtime of the GPU code is

0.412230-s, while the total runtime of its analogue, sortrows, in the MatLab code is 0.617-s. In other words the GPU code performed the same data operations 0.20477 seconds faster for a speedup of 1.50x. Substituting the GPU runtime into the sortrows runtime of the original algorithm, the new overall runtime of the algorithm is 0.85623 seconds compared to the original runtime of 1.061 seconds. Therefore, the GPU implementation takes 80.07% of the time of the original, or a 1.24x speedup.

5.2: Discussion of the Results

The goal of the project was to create an implementation of the deformable model segmentation algorithm that could run at real-time speeds. Although the new implementation did reduce the execution time from 1.06 to 0.86 seconds, it is still not fast enough to run multiple times per second, as would be necessary for real-time applications. However, as was shown in Tables 3 and 4, the majority, 98.8% in fact, of the GPU runtime was spent simply transferring the data from the CPU to the GPU, not actually operating on the data. The reason for this is that each time the GPU code was to be run, an entirely new set of data on had to be sent from the CPU to the GPU and back again. Almost all of this memory transfer overhead could have been avoided if the entire algorithm had been translated to C and run completely on the GPU. Data would only need to be transferred to the GPU and back once, at the beginning and end of the algorithm. Currently, the memory transfer overhead is 0.015672-s × 26 = 0.407472 seconds, almost half of the total runtime; translating the entire algorithm to the GPU would have achieved a much faster runtime, possibly even one fast enough for real-time use.

CHAPTER 6: CONCLUSIONS

New multi-core microprocessor architectures such as those found in the Cell processor and GPUs offer the possibility of large processing speedup, but they come at the cost of more difficult, longer software development time—at least until better tools are created to streamline the parallelization and optimization process. This cost was certainly felt in the undertaking of this project; the parallel programming involved was far harder and required more time than normal application development would have.

NVIDIA’s CUDA platform, even though it is currently in an early stage of development, proved to be quite efficient and robust. Additionally, in terms of ease of programming, CUDA was far less daunting than the Cell processor architecture, mainly since much of the memory management was handled by the system, not the programmer. On the other hand, CUDA suffered performance-wise since large data arrays had to be transferred from the CPU to the video hardware each time the GPU was called. Unlike the Cell processor, the GPU has no GP processor meaning it must solely be controlled by the computer’s CPU. The addition of a single, traditional processor onto the actual GPU hardware would have allowed the entire algorithm to run on the device and would have eliminated the data transfer performance reduction that prevented the algorithm from achieving real-time capability.

The goal of creating a real-time applicable deformable model segmentation algorithm was not realized in the project; in these terms the project failed. Nevertheless, the 1.24x speedup achieved does show serious potential for multi-core architectures such as NVIDIA’s GPU, especially since much of the execution time was spent transferring data which could have mostly been avoided by translating more of the algorithm to the

GPU. Regrettably time constraints prevented this and the genuine possibility of creating a real-time deformable model segmentation algorithm.

Certainly, researchers should continue to experiment with parallel programming on multi-core architectures to achieve better performance. Especially considering the multi-core trends in the mainstream microprocessor industry, parallel programming will become more and more indispensable to take full advantage of ones computing hardware. However, it is critical that improved micro-architectures and compilers be created to ease the software engineer’s burden. Currently, parallel programming is truly harder than writing for tradition single-core processors, and better tools and methods will be necessary to create parallel applications as efficiently as before. Otherwise, fewer programs will be able to be developed, impairing all industries that rely on computing—which, today, is nearly all of them.

BIBLIOGRAPHY

Asano, T., Chen, D. Z., Katoh, N., & Tokuyama, T. (1996). Polynomial-time solutions to image segmentation. New Orleans, LA: Society of Industrial and Applied Mathematics.

Aslandogan, Y. A., & Yu, C. T. (1999). Techniques and systems for image and video retrieval. Knowledge and Data Engineering, 11(1), 56-63.

Baker, Monya. (2005). Searching for squishy shapes: vision algorithm models deformable objects. Technology Review, 108, 83.

Belongie, S., Malik, J., & Puzicha, J. (2002). Shape matching and object recognition using shape contexts. Pattern Analysis and Machine Intelligence, 24, 509-522.

Bryll, R., Rose, R. T., & Quek, F. (2005). Agent-based gesture tracking. Systems, Man. and Cybernetics, 35, 795-810.

Cheung, K. W., Yeung, D. Y., & Chin, R. T. (2002). Bidirectional deformable matching with application to handwritten character extraction. Pattern Analysis and Machine Intelligence, 24, 1133-1139.

Duan, Y., Yang, L., Qin, H., & Samaras, D. (2004). Shape reconstruction from 3d and 2d data using pde-based deformable surfaces. European Conference European Conference on Computer Vision, 3, 238–251.

Gelsinger, P. P. (2001). Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. In IEEE International Solid-State Circuits Conference, Digest of Technical Papers (pp. 22-25). Piscataway, NJ: IEEE.

Gibson, S. F., & Mirtich, B. (1997). A survey of deformable models in computer graphics (Technical Report TR-97-19). Cambridge, MA: Mitsubishi Electric Research Laboratories.

Goering, R. (2004, October, 4). Matlab edges closer to electronic design automation world. *EE Times*. Retrieved March 25, 2007, from http://www.eetimes.com/showArticle.jhtml?articleID=49400117.

Greene, J., & Cooper, R. (2005). A parallel 64K complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine processor. Paper presented at the Global Signal Processing Exposition, Santa Clara, CA.

Grinstead, B., Koschan, A., Page, D., & Abidi, M. (2006, April). Model building for simulation and testing under uncertain conditions. *Proc. SPIE Modeling and Simulation for Military Applications, 6228*, 98-109.

Intel. (2006). Intel Core Duo Processor and Intel Core Solo Processor on 65 nm process. Santa Clara, CA: Intel.

Kaliyaperumal, K., Lakshmanan, S., & Kluge, K. (2001). An algorithm for detecting roads and obstacles in radar images. Vehicular Technology, 50(1), 170-182.

Kamgar-Parsi, B., Jain, A. K., & Dayhoff, J. E. (2001). Aircraft detection: a case study in using human similarity measure. Pattern Analysis and Machine Intelligence, 23, 1404-1414.

Lachaud, J. O., & Taton, B. (2005). Deformable model with a complexity independent from image resolution. *Computer Vision and Image Understanding, 99,* 453-475.

Li, B., & Acton, S. T. (2006). On the Poisson inverse gradient for automatic initialization of deformable model image segmentation. Unpublished manuscript, University of Virginia, Charlottesville, VA.

Liao, H., Hata, N., Nakajima, S., Iwahara, M., Sakuma, I., & Dohi, T. (2004). Surgical navigation by autostereoscopic image overlay of integral videography. Information Technology in Biomedicine, 8(2), 114-121.

Lorts, D. (2000). Combining parallelization techniques to increase adaptability and efficiency of multiprocessing DSP systems. In Proceedings of Ninth DSP Workshop: *First Signal Processing Education Workshop*, Hunt, TX.

Lundstrom, M. (2003). Moore’s Law forever? Science, 299(5604), 210-211.

Mahoney, D. P. (2001). Painting with feeling. Computer Graphics World, 24, 15.

McInerney, T., & Terzopoulos, D. (1996). Deformable models in medical image analysis. Medical Image Analysis, 1(2), 91-108.

Moreland, K., & Angel, E. (2003). The FFT on a GPU. In M. Doggett, W. Heidrich, W. Mark, A. Schilling (Eds.), Graphics Hardware 2003: San Diego, California, July 26-27, 2003. New York: Association for Computing Machinery.

Neuenschwaner, W., Fua, P., Szekely, G., & Kubler, O. (1997). Velcro surfaces: fast initialization of deformable models. Computer Vision and Image Understanding, 65, 237-245.

NVIDIA. (2007). *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. Santa Clara, CA: NVIDIA.

Olsen, F. (2003). Virginia Tech takes 1,100 Macs and turns them into a supercomputer. Chronicle of Higher Education, 50(11), 38.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. 2005. A survey of general-purpose computation on graphics hardware. Proceedings of Eurographics 2005, *State of the Art Reports*, 21-51.

Puri, R., Karnik, T., & Joshi, R. (2006). Technology impacts on sub-90nm CMOS circuit design and design methodologies. Paper presented at the IEEE International Conference on VLSI Design, Piscataway, NJ.

Rajasekaran, S. (2000, January). A framework for simple sorting algorithms on parallel disk systems. *Theory of Computing Systems, 34*(2), 101-114.

Sakamoto, M., Nishiyama, H., Satoh, H., Shimizu, S., Sanuki, T., Kamijoh, K., Watanabe, A., & Asahara, A. (2005). An implementation of the Feldkamp algorithm for medical imaging on cell. New York: IBM Corporation.

Shahidi, R., Bax, M. R., Maurer, C. R., Jr., Johnson, J. A., Wilkinson, E. P., Wang, B., West, J. B., Citardi, M. J., Manwaring, K. H., & Khadem, R. (2002). Implementation, calibration and accuracy testing of an image-enhanced endoscopy system. Medical Imaging, 21, 1524-1535.

Smith, J. E., Hsu, W.C., Hsiung, C. (1990). Future general purpose supercomputer architectures. Intl. Journal of Supercomputer Applications, 90, 796-804.

Tejada, E., & Ertl, T. (2005). Large steps in GPU-based deformable bodies simulation. Simulation Modelling Practice and Theory, 13, 703-715.

Zheng, G., Singla, A. K., Unger, J. M., & Kale, L. V. (2002). A parallel-object programming model for petaflops machines and blue gene/cyclops. Paper presented at the NSF Next Generation Systems Program Workshop, 16th

International Parallel and Distributed Processing Symposium, Fort Lauderdale, FL.

Kompatsiaris, Y. (2004). Segmentation. Retrieved March 25, 2007, from http://www.iti.gr/~ikom/research_segmentation.htm.

ADDITIONAL REFERENCES

Bailey, D. H. (1988). A high-performance FFT algorithm for vector supercomputers. *Intl. Journal of Supercomputer Applications, 2*(1), 82-87.

Bechini, A., Prete, C. A. (2002). Performance-steered design of software architectures for embedded multicore systems. *Software - Practice and Experience, 32*, 1155-1173.

Chow, A. C., Fossum, G. C., & Brokenshire, D. A. (2005). *A programming example: large FFT on the Cell Broadband Engine*. New York: IBM.

Duncan, J. S., & Ayache, N. (2000) Medical image analysis: progress over two decades and the challenges ahead. Pattern Analysis and Machine Intelligence, 22(1), 85-106.

Owens, J. D., Shubhabrata, S., Horn, D., (2005). *Assessment of graphic processing units for Department of Defense digital signal processing applications* (Technical Report ECE-CE-2005-3). Davis, CA: University of California.

Shahidi, R., Tombropoulos, R., & Grzeszczuk, R. P. (1998). Clinical applications of three-dimensional rendering of medical data sets. Proceedings of the IEEE, 86(3), 555-568.

Terzopoulos, D., Witkin, A. (1988). Physically based models with rigid and deformable components. *IEEE Computer Graphics and Applications, 8*(6), 41-51.

APPENDIX A: MATLAB PROFILER RESULTS

Tables reveal runtime and percent of runtime of each function with the algorithm.

Functions with less than 1% of the total runtime were omitted.

PoissonSolver (1 call, 1.185 sec) – Main algorithm

Function Name	Calls	Total Time	% Time
ismember	8	0.929 s	78.5%
ind2sub	2	0.024 s	2.0%
bwperim	1	0.019 s	1.6%
Self time (built-ins, overhead, etc.)		0.202 s	17.0%
Totals		1.185 s	100%

ismember (22 calls, 1.009 sec) – Single function taking the vast majority of runtime

Function Name	Calls	Total Time	% Time
unique	18	0.458 s	45.4%
sortrows	9	0.317 s	31.4%
ismembc2	8	0.029 s	2.9%
Self time (built-ins, overhead, etc.)		0.202 s	20.1%
Totals		1.185 s	100%

unique (20 calls, 0.467 sec) – One of two of the major functions of ismember

Function Name	Calls	Total Time	% Time
sortrows	17	0.334 s	71.5%
Self time (built-ins, overhead, etc.)		0.133 s	28.5%
Totals		0.167 s	100%

sortrows (26 calls, 0.651 sec) – Notice, this function is called 9 times by ismember and 17 times by unique

Function Name	Calls	Total Time	% Time
sort_back_to_front	26	0.576 s	88.5%
Self time (built-ins, overhead, etc.)		0.075	11.5.%
Totals		0.651 s	100%

As a result, the function sortrows takes up 0.651 seconds of the algorithm's 1.185 second runtime, or 54.9% of the runtime, and was targeted for improvement and optimization.