GPUCT: A GPU-Accelerated CT Reconstruction System

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

Drew Maier

March 30, 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.

____

Approved	____________________________ *Kevin Skadron*	(Technical Advisor)
Approved	____________________________ Benjamin Cohen	(Science, Technology and Society Advisor)

PREFACE

This project has provided me with an incredibly rewarding (and at times frustrating) look at the world of computer science research. I first heard of general-purpose computation with graphics hardware in CS446: Real-time Rendering, a course taught by Prof. David Luebke last spring. Prof. Skadron guest lectured one class and talked about how he sought researchers to pursue this field. I approached Prof. Skadron last summer looking for one such thesis project, and he told me about an opportunity with the UVA Department of Interventional Radiology developing a CT reconstruction system.

I would like to thank Prof. Skadron for allowing me to participate in research with broader humanitarian implications. The other members of Prof. Skadron's multicore research group also deserve recognition, as they helped me comprehend GPU programming. I would also like to thank David Luebke, who has since begun working for NVIDIA research, for donating NVIDIA hardware to the CS Department and for giving several lectures on the new CUDA paradigm. Of course, I must also thank Prof. Cohen, who has helped me immensely in drafting this document, as well as developing my social and ethical awareness as an engineer. Finally, I thank my friends and family for supporting my expensive and time-consuming computer addiction.

Drew Maier

March 30, 2007

ABSTRACT

CT scanning is a medical imaging technique commonly used in hospitals, including the University of Virginia Hospital, to see inside the human body. Modern CT scanners can generate images of the body in three dimensions, a process called 3D reconstruction. This project illustrates the feasibility of using graphics hardware (GPUs) to process CT scans in a more efficient and inexpensive manner than current commercial reconstruction systems. Additionally, this research considers the ethical and social implications of an improved CT reconstruction system in terms of risks for hospitals and patients.

Other researchers have used GPUs to improve CT reconstruction processing, but none have done so with NVIDIA's new GPU programming Compute Unified Driver Architecture (CUDA) paradigm. This paradigm greatly simplifies GPU programming by providing transparency to programmers.

This project originally hoped to create two fully working reconstruction systems: a uniprocessor version and a GPU version. Problems with implementing the Feldkamp-Davis-Kress reconstruction algorithm led to the decision of implementing only part of this algorithm, called backprojection, which produced unintelligible, but most likely correct, output. To parallelize backprojection, this project used Foster's Design Methodology, a parallel design methodology.

The GPU backprojection program ran up to 56 times faster than a uniprocessor version, which shows promise for GPU-accelerated CT reconstruction processing. However, without valid output, these results are tentative. Future researchers can use this project as a basis for a complete GPU CT reconstruction system.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

GLOSSARY OF TERMS

Backprojection	The process of extrapolating an object's geometry based on many scans of that object [3]
CPU	Central processing unit; the component of a computer that interprets instructions and processes data, typically used as synonym for microprocessor [wikipedia.org]
CT	Computed Tomography; a type of medical imaging that uses x-rays to acquire images [1]
CUDA	Compute Unified Device Architecture; the underlying hardware in the newest NVIDIA chipset, designed for general applications [7]
Data Parallelism	A property in which elements in a set of data receive the same instructions [5]
GPU	Graphics Processing Unit; a dedicated graphics chip
GPGPU	General-Purpose computation on GPUs; using graphics chips for non-graphical applications and computations [6]
Voxel	A volume element, representing a value on a regular grid in three dimensional space [wikipedia.org]
SDK	Software Development Kit; a set of tools a programmer uses to create software for a specific platform [wikipedia.org]

CHAPTER 1: INTRODUCTION

Computed tomography (*CT*) is a type of medical imaging that generates images of the internals of an object based on scans of the object from several angles [1: 66]. Modern CT scanners can generate images in two or three dimensions. Doctors can rotate and zoom in 3D images, revealing higher levels of detail than static 2D images. This project aims to lessen the time to reconstruct a CT scan into 3D image by means of a new computational tool, the NVIDIA 8800 graphics chipset. This will increase efficiency in hospital radiology departments and will cost less than current reconstruction systems.

The University of Virginia Division of Angiography, Interventional Radiology, and Special Procedures uses CT scans to view blood flow through the human body. Currently, their CT scan 3D reconstruction system generates images with expensive specialized hardware. Depending on resolution, the time from scan start to onscreen interactive 3D model may be upward of ten minutes [2]. According to specialists from the University of Virginia Department of Interventional Radiology, many doctors reluctantly opt for 3D reconstruction of a CT scan because it is a lengthy process [2]. By not using reconstruction, however, doctors may not see important blood flow information – information that could reveal the presence of arterial disease, cardiovascular disease, or cancer, to name a few. Even when doctors order reconstruction, reconstruction processing takes enough time that they leave the room to attend to other patients.

Reconstruction is just one of several stages involved in a CT scan. First, a radiologist injects "contrast dye" into the patient. This dye is highly reactive with x-rays and shows up brightly in x-ray images. After injection of the dye, a large x-ray sweeps around the patient, capturing x-ray images. This process takes about twenty seconds, depending on how many images the scanner captures. At this point, the CT scanner sends scan data to a dedicated processing machine via computer network. This machine runs an algorithm on the scan data that extrapolates three-dimensional geometry based on two-dimensional images. This process, known as *backprojection*, takes approximately 97% of the total time to reconstruct an image [3]. The most widely-used backprojection algorithm is the Feldkamp-Davis-Kress algorithm (FDK). A typical scan captures over 100 images, each of which could cover over a million pixels in area. This means that the FDK algorithm must run on over 100 million values per CT scan. Thus, reconstruction systems must execute the FDK algorithm quickly and efficiently to perform ideally. This project accomplishes reconstruction with a graphics processing unit (*GPU*).

Since their breakthrough in the late 1990s, computer users have traditionally installed GPUs for accelerating graphically intensive applications such as games. To accelerate graphics, the GPU executes computationally intensive operations instead of the central processing unit (*CPU*) because the GPU is a superior math processor [4: 463]. One can think of this like an engineering firm. The boss (the CPU) is a capable engineer, but better at management, and therefore leaves intense number-crunching to workers who are more mathematically inclined (the GPU). In addition to performance gained by reducing CPU load, engineers designed GPUs specifically to execute tasks that express *data parallelism.* Data parallelism is a property in which many elements in a set of data receive the same operation [5: 10]. Graphics processing involves many data parallel instructions to display an image.

Graphics calculations are not, however, the only calculations that express data parallelism. Recognizing this, many researchers have begun to realize new ways with which to use GPUs. This movement is known as general-purpose computing on graphics processing units, or *GPGPU*. Some examples of GPGPU include signal processing, physical simulation, data mining, and image processing [6]. GPUs are an excellent medium with which to process CT scan data. CT scan processing is a data parallel operation since it runs the FDK algorithm on each pixel in a scan. As such, a GPU will most likely reconstruct scans faster than a CPU.

Until late 2006, programmers had two options for creating GPGPU applications. In the first scheme, programmers write GPGPU applications as if they are standard graphics applications. These applications display input data as a 2D image (known as a texture) and run necessary computations on the texture. In other words, the programmer tricks the computer into thinking that the texture is a picture to display, not data on which to run computations. Several projects use this method, but programmers have generally found this paradigm unintuitive. Other GPGPU projects use custom GPGPU languages designed by researchers, which allows for a more straightforward programming approach. With custom languages, however, GPGPU programmers must learn an entirely new language, which requires extra time. Also, these custom languages have limited functionality and are difficult for programmers to get working, so they have not gained much popularity.

NVIDIA Corporation's latest GPU, the 8800, is the first GPU created with explicit GPGPU functionality. This design, known as the Compute Unified Device Architecture (*CUDA*), allows programmers to make programs for the 8800 using extensions to the standard C language [7]. As such, CUDA avoids many of the pitfalls of the previous ways of creating GPGPU applications. By writing in C, programmers do not have to learn an entirely new language. Additionally, NVIDIA has designed this architecture such that programmers can install the software development kit *(SDK)* and begin coding immediately with a full set of features. The University of Virginia Department of Computer Science received an 8800 card in December that students can use for research purposes.

Researchers have used GPUs for 3D reconstruction in the past, though under the previous GPGPU paradigms. In particular, Fang Xu and Klaus Mueller from Stony Brook University first proposed CT reconstruction with a GPU nearly three years ago. They have published several papers since, each of which illustrates the benefits of using GPUs for reconstruction processing. Chapter 3: Review of Technical Literature further discusses these findings.

Though this project is not the first to use GPUs for 3D reconstruction, it is the first to use NVIDIA's new CUDA to do so. This research shows the strengths and weaknesses of CUDA by comparing its results to those of Xu and Mueller. This project also analyzes the backprojection parallelization process at a more critical level than many academic papers. Though still ongoing, this project hopes to culminate as a working CT reconstruction system.

Reducing 3D reconstruction processing time has several benefits for hospitals. First, doctors feel encouraged to prescribe 3D reconstruction more often, since it is less time consuming. Increased use of CT reconstruction helps lead to early diagnoses, and could mean the difference between life and death for a patient. Another benefit of this is that doctors are more likely to stay with their patients during reconstruction. Though CT scans are minimally invasive, they can frighten patients, especially if patients are alone.

A further advantage of decreasing this time is that doctors maintain their concentration on a given patient. Anyone, not just a doctor, will have a harder time focusing on problem solving if they must shift their focus for ten minutes. From an economic perspective, reducing scan time equates to better hospital efficiency because minimizing scan processing time will increase patient examination throughput. Each scan does have an associated cost, however, meaning that the hospitals spend more money. Fortunately, this extra cost is negligible since the largest associated cost of CT scans is dye, for which the patient typically pays.

Aside from reducing reconstruction time, processing data on commercial graphics hardware is significantly cheaper than using specialized hardware to process data. The existing processing system uses a graphics processing unit that costs somewhere in the five-figure range [8]. The NVIDIA 8800 GPU used in this project retails for $599. This project considers itself successful even if commodity hardware achieves the same performance as specialized hardware because the cost of commodity hardware is less by orders of magnitude.

The remainder of this report discusses the findings of this research project. First, in Chapter 2: Analysis of Social Context and Ethical Implications, this report discusses the historical, social, and ethical contexts in which this project lies. As this is a medical project, this report discusses ethical and social impacts of this research in depth. Chapter 3: Review of Technical Literature follows this with a critical review of literature related to 3D reconstruction, GPGPU, and the evolution of combining GPGPU with CT reconstruction. Moving into this particular project's research, Chapter 4: Materials & Methodology, summarizes the materials required for this project, as well its methodology for conducting research. Using this methodology to complete the research, Chapter 5: Results & Result Analysis provides results and analysis of this project, presenting the benefits and disadvantages of using GPUs for reconstruction processing. Based on these results, the final section, Chapter 6, presents suggestions for future work on this project and evaluates how well this project accomplished its goals. This chapter is the most important for the reader, as it highlights the main conclusions of this project.

CHAPTER 2: ANALYSIS OF SOCIAL CONTEXT AND ETHICAL IMPLICATIONS

This project benefits the University of Virginia hospital as an inexpensive mechanism for improved efficiency. However, it cannot only consider efficiency and cost; it must also operate ethically. As such, this chapter presents economic and social contexts in order to predict broader impacts of this research on both doctors and patients.

Efficiency is the core of this project; hardware and algorithmic efficiency in the CT system implies organizational efficiency for hospitals. There is often a paradox with efficiency in a workplace: greater efficiency allows an office to save time, yet all employees still work a full day, calling into question the purpose of efficiency. In the case of this project, however, this paradox applies in a positive manner because efficiency in a hospital results in doctors helping more patients. Moreover, doctor productivity increases since doctors have less idle time during their workday.

This organizational context, emphasizing reduced costs and efficiency, intertwines with a greater political context. The Department of Interventional Radiology would likely have faster CT processing equipment if the state provided them with more funds. At some point in time, a state appropriations committee agreed to give a certain amount of money to the hospital for equipment. In turn, the hospital allocated some of their total budget to the Department of Interventional Radiology. Thus, political decisions have set a limit on the level of quality possible with CT scans. A corporate partner produces the current reconstruction system, providing another example of the political context. Hospitals and corporations frequently partner so that hospitals can buy equipment for less in exchange for using one brand of products. Other corporations may make faster 3D reconstruction systems, but hospitals cannot purchase them due to constraints of partnership.

Without standardized health care in the United States, any advances in medicine and medical technology have social implications stemming from economics. Regardless of whether patients have health insurance, they must at least pay a deductible for any care they receive during a hospital visit. CT scans have further costs to hospitals, since hospitals must pay for equipment maintenance and employ a full-time operator. Thus, insured patients can more viably receive CT scans. Furthermore, patients with insurance are more likely to go to the hospital at all since they are not afraid of leaving with a large bill. In an even broader scope, this project may seem unfair as it helps to improve the nationally recognized UVa. Hospital [9], not one of the many hospitals in other states or countries that are more desperate. While other hospitals certainly have more basic needs than the UVa. Hospital, this project creates a generalized low-cost system that can work anywhere. Thus, more hospitals can adapt CT reconstruction, giving patients better care.

The contexts described above reveal ethical matters associated with promoting fast CT scans. In general, issues arise with improved hospital efficiency and increased use of scanning – issues that have both positive and negative effects for doctors and patients. Other ethical issues arise with this project's technical approach.

As stated previously, a speedup in the scanning process results in increased hospital efficiency. On the positive side, doctors help more people daily, which is hopefully their goal as a professional. A side effect of this, however, is that doctors may have fewer breaks during their workday. This could lead to higher levels of stress in an already stressful occupation, which could in turn affect the care that a patient receives. Therefore doctors must maintain a reasonable workload, even if technological advances allow them to work more than they currently can.

Since this project promotes frequent use of CT, it must ensure the safety of CT for patients. As with any form of radiology, interventional radiology poses some health risks. First, recall that before a CT scan, a radiologist injects the patient with dye. In rare cases, this dye can cause kidney damage or even a potentially fatal allergic reaction [10]. Also, during the scan, the x-ray capture device emits a “moderate to high” amount of radiation [11]. Research has linked radiation exposure to cancer, meaning that CT scans can contribute to the problem were designed to help mitigate [12, 13]. With faster CT scans, doctors can prescribe CT more frequently, thus increasing the number of patients exposed to these health risks. Presently, a doctor consults a patient before that patient undergoes a CT scan, which helps prevent scan complications.

The implementation of this reconstruction system presents additional ethical issues. A positive implication of this project is that it contributes to the promising field of GPGPU. GPGPU excites software developers because it allows programs to run faster without radical hardware changes. However, GPGPU developers do not necessarily create morally sound programs. Just as GPGPU can speed up calculations for medical imaging, so too can it speed up calculations for immoral purposes. GPGPU projects are not the only ones faced with such a dilemma; any software or hardware project can have similar ethical implications. Typically, software developers follow a code of ethics to ensure moral behavior. The next chapter, Chapter 3, moves from the broad social and ethical contexts just described to discussing the narrower technical context in which this project resides.

CHAPTER 3: REVIEW OF TECHNICAL LITERATURE

CT scanning is part of the broad field of medical imaging. It remains one of the most popular methods of medical imaging due to its ability to produce high contrast images at lower cost than other methods such as MRI. This section outlines the history of CT, with a focus on the evolution of image reconstruction, to illustrate the basis for undertaking this research.

Computed tomography research began in the 1950s with Allan Cormack, who devised the mathematical and experimental foundation for a CT scanner. Godfrey Hounsfield realized the first CT scanner in 1972, and received the 1979 Nobel Prize in Physiology or Medicine for his accomplishment. Since then, the most significant change in CT imaging has been the breakthrough of 3D reconstruction, which researchers discovered in the mid-eighties [1:89][14].

Around the same time period, a more socially significant technological breakthrough, the personal computer, changed how businesses, researchers, and individuals operated. Low-cost machines like the Apple II offered everyone – not just institutions – to the power of software. On another front, the PC birthed a new generation of amateur programmers due to easy and free high-level languages such as BASIC. The PC undoubtedly changed hospital operations, particularly through medical records and medical imaging. Indeed, economically feasible reconstruction systems would not exist without the PC.

Three-dimensional reconstruction techniques consist of three phases: weighting, filtering, and backprojection. Of these phases, backprojection is the most computationally expensive. As such, researchers have posed a variety of algorithms to efficiently project scan data from 2D to 3D space. These algorithms depend on the manner in which the CT scanner captures images. The University of Virginia's CT scanner captures data by rapidly rotating a 2D x-ray sensor around a patient, a process called cone-beam scanning.

Feldkamp et al. published the first cone-beam algorithm in 1984 [15]. Their algorithm extended existing algorithms for 2D reconstruction into 3D with excellent results. This algorithm approximates reconstruction, which means that it is fast, albeit with some error. Images produced by Feldkamp et. al. illustrate that this error is insignificant, and as a result, the FDK algorithm is still the basis of modern 3D reconstruction algorithms. While many researchers have optimized the FDK algorithm to achieve faster running times, hardware advances have provided the most significant performance boosts to 3D reconstruction.

Parallel computer architectures are particularly well-suited for backprojection since backprojection involves many independent operations. Reimann et al. achieved a speedup of 1.61 on a parallel system, as described in their paper "Parallel Computing Methods for X-Ray Cone Beam Tomography with Large Array Sizes" [16]. While Reimann et. al. decreased the overall running time of reconstruction, their solution is impractical since it requires an expensive cluster of computers. Sakamoto et al. researched the possibilities of medical imaging on a new parallel architecture, the Cell Broadband Engine Architecture (CBEA), developed by IBM, Sony, and Toshiba. With the CBEA, Sakamoto achieved a speedup of 20 running Feldkamp's algorithm [3]. Though this is promising, programmers have already expressed that programming CBEA systems is unintuitive. Additionally, the only commercially available CBEA systems are the Playstation 3 gaming console and high-performance server racks. Graphics processors have much in common architecturally with the aforementioned systems, though they have the added benefit of lower cost.

Graphics processors contain many parallel processors optimized for computationally intensive math operations. Realizing the potential of graphics hardware for CT reconstruction, Xu and Mueller of Stony Brook University published “Towards a Unified Framework for Rapid 3D Computed Tomography on Commodity GPUs” in 2003. This paper presents preliminary findings of using GPUs for 3D reconstruction using several algorithms, including the FDK algorithm. Xu and Mueller found the FDK algorithm to give the fastest reconstruction times, at the expense of some image quality [17].

A follow-up paper, “Ultra-Fast 3D Filtered Backprojection on Commodity Graphics Hardware,” published in 2004, concentrates explicitly on running the FDK algorithm on GPUs. This more technical paper explains details of how to implement the FDK algorithm on graphics hardware, as well as optimizations to allow the algorithm to run as fast as possible. Using these techniques, Xu and Mueller ran the FDK algorithm roughly 7.5 times faster on a GPU than on a CPU with comparable image quality [18]. Moreover, they achieved an enormous speedup of 37 on the GPU with slightly noisier images.

In their most recent paper, Xu and Mueller present additional optimizations present additional optimizations to their previous papers, as well as more comprehensive results [8]. Their goal in this research was to achieve speedups in a number of 3D reconstruction algorithms, including the FDK algorithm. While they achieved reasonable speedups using other algorithms, the highest-quality form of the FDK algorithm executed as quickly as the low-quality form in their 2004 paper. Thus, with negligible loss of image quality, they achieved a speedup of 37 by running the FDK algorithm on a GPU instead of a CPU.

This thesis project is a continuation of Xu and Mueller's research. While they have produced impressive results in their research papers, they reveal little about their methodology and possible flaws in their implementation. They give results only for two scan resolutions and do not explain expected performance for smaller or larger datasets. This project provides results for several output resolutions. The most important distinction between this project and those of Xu and Mueller is that this project operates under the new GPGPU paradigm. This paradigm is spiritually similar to the PC revolution in that it gives ordinary users control over their hardware. As the next two chapters show, this paradigm is not only more intuitive for the programmer, but is also faster than previous GPGPU methods.

Chapter 4: Materials & Methodology

The main tenet of the GPGPU paradigm is for programs to achieve better performance by fully utilizing standard graphics hardware. As such, this project required only hardware, software, and literature. This project's methodology encompassed both its software development cycle and its approach to parallelizing CT reconstruction. The following section describes in greater detail the materials required for this project's operation, as well as its approach to parallel software design.

Hardware accounted for a significant portion of this project's cost. This project pushed the boundaries of consumer-level computers, so it required a fast system with a large amount of memory, as well as GPU compatibility. In addition to the main system, this project used the newest GPU chipset, the NVIDIA 8800. As noted earlier, a programmer need not use this chipset for GPGPU programming, though it allows for the most intuitive GPGPU programming interface currently available. This project ran on a Dell XPS gaming computer, upgraded with the NVIDIA 8800 graphics card. The Department of Computer Science purchased the XPS system for Prof. Skadron's research team, but NVIDIA donated an 8800 GPU to the Department for academic research purposes. Refer to Listing 1 in Appendix D for more detailed specifications of the XPS system.

This project required several types of software: a programming environment, an output visualization tool, and an interface for running programs on the GPU. It used standard text editors and compilers, such as `Emacs` and `gcc`, in Fedora Core 4 (a freely available Linux distribution) to develop most of the software. Portions of this project ran in Windows in Microsoft's Visual Studio programming environment because of stability issues in Linux. Given its graphical nature, this project required a means of viewing output in a three dimensional space. While several commercially available programs do this, this project developed a custom *voxel* visualization tool to allow for more control of the output parameters. The most crucial piece of software in this project was the Software Development Kit (*SDK*) for the GPU. This software allows programs to communicate directly with the GPU, thus enabling GPGPU. NVIDIA provides the CUDA SDK for free on its website. A nice feature of this SDK is that it includes a GPU emulator so that programmers can write programs for CUDA even if they do not have CUDA graphics cards.

Literature for this project pertained to CT reconstruction and GPU programming. Given the novelty of the GPGPU paradigm (and even newer CUDA), few authors have published hard books on GPU programming for non-graphical applications. The best resource for working with CUDA is [19], which is included with the CUDA SDK. This document, created by NVIDIA, describes how to create GPU programs from scratch and also provides documented examples. However, much of the research for this project involves medical imaging techniques. Online databases such as the IEEE host many of the significant CT reconstruction papers for academic use. Finally, this project required scan data for input. The Department of Interventional Radiology provided this in the form of 133 scans of a torso phantom (an artificial torso). Figure 1 shows one such scan.

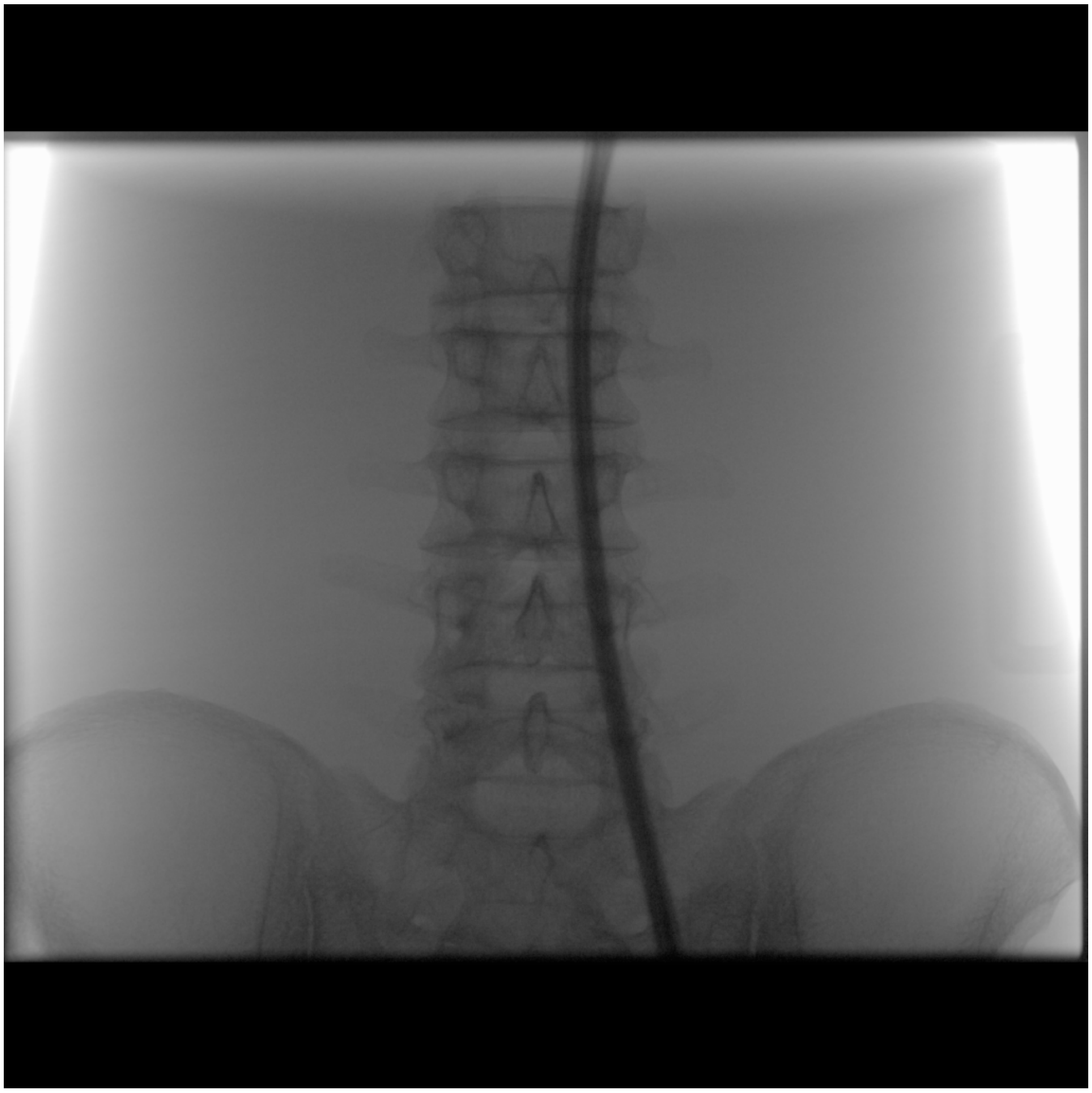


FIGURE 1 – CT scan of the phantom torso (data provided by UVA Department of Interventional Radiology, image generated by author)

This section now discusses this project's approach to a GPU-accelerated CT reconstruction system. Unforeseen problems with the first phase of this project resulted in changes to its methodology. Initially this project planned to create two reconstruction systems: a CPU (uniprocessor) version, and a GPU (parallel) version. First, this project would write the conceptually simple CPU code. Using this CPU code as a template, the project would then write a more complex GPU version. Both the CPU and GPU phases contained four typical software development phases: planning, prototyping, development, and testing. While this development model ensured accuracy of the GPU code, it created a dependency between the GPU code and the CPU code, since the GPU code could not begin until the project had completed the CPU code phase (Figure 3 in Appendix A). The dependency of the GPU code on the CPU code proved problematic when the CPU code halted in its verification stage; the longer the CPU phase took to implement, the less time this project could devote to implementing GPU code.

The verification stage of the CPU programming phase halted this project's development. To verify this program, this project viewed its output through the voxel visualization application. Figure 2 shows this application when given the output of the CPU backprojection program. This output is obviously incorrect, as it does not resemble a 3D model of the torso phantom as seen in Figure 1.

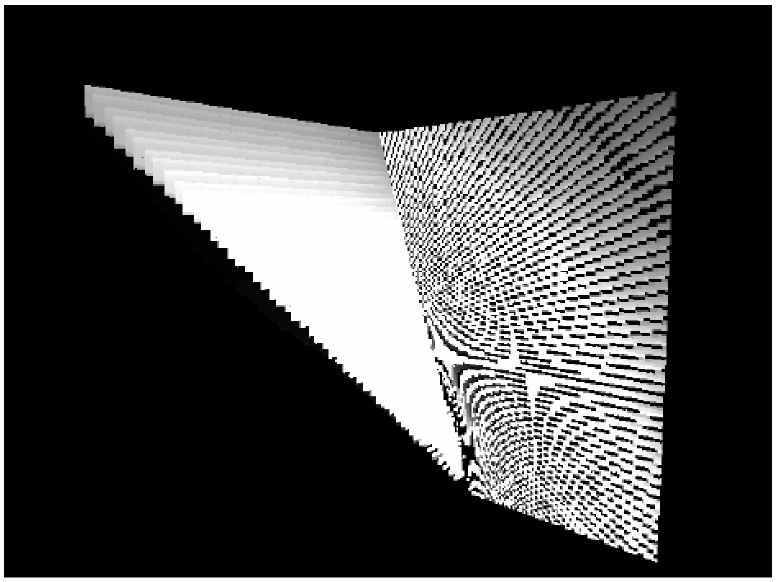


FIGURE 2 – Visualization of torso phantom from CPU program (created by author)

Three possibilities could explain the incorrect output:

1. The CT scans provided by the hospital were incompatible with the FDK algorithm

2. The C code translation of the FDK algorithm was not equivalent to the actual FDK algorithm
3. The C code correctly translated the FDK algorithm, but lacked some intermediate step

Experts from the Department of Interventional Radiology assured that professional reconstruction systems use similar data as input for their software, thus eliminating the first possibility. As for the second option, Figure 5 in Appendix A shows the correct FDK backprojection algorithm, obtained from [20]. Figure 6 in Appendix A shows this project's C code translation of the FDK algorithm, which appears equivalent to the original algorithm.

Research papers on CT reconstruction provide several approaches to the FDK algorithm. Implementing various versions of the FDK algorithm in the CPU program produced identical output, even if the intermediate math differed. This indicated that the code implementation was in fact correct, but that it lacked some sort of input data formatting. Xu and Mueller mention in [8][17] and [18] that the FDK algorithm requires filtered scan data as input, though they neglect to explain the filtering algorithm in depth. Due to such lack of emphasis on filtering in CT reconstruction literature, this project had erroneously interpreted that reconstruction programs could overlooked the filtering stage. Given that the CPU program's output was consistent with different algorithmic implementations, and that [3] states that backprojection is the most computationally significant stage, this project froze the CPU code without including a filtering component. The project then converted its software development cycle from a two-phase spiral approach to a single spiral containing both projects (Figure 4 in Appendix A). With this approach, the project had not completed the CPU program, but it could commence the GPU program to observe the parallelized performance of the backprojection program.

With a new software development cycle in place, this project transitioned to GPU programming. This phase used the CPU code as a basis for writing the new GPU code. To translate CPU code to GPU code, this project could not simply copy and paste; it had to analyze the algorithm to find parallelization opportunities. To successfully parallelize the FDK algorithm for the GPU, this project used Foster's Design Methodology, a four-step process for designing parallel algorithms. Foster's Design Methodology consists of four phases: partitioning, communication, agglomeration, and mapping [5].

In Foster's Design Methodology, a project's partitioning phase divides its major computations into smaller parallel pieces called primitive tasks. The FDK algorithm is "embarrassingly parallel," meaning that it is completely data parallel; no voxel intensities are dependent on one another. As such, the project partitioned the overall task of creating a three-dimensional voxel model into D^3 tasks, where D is the length of one side of the cubic space. Thus, the FDK algorithm's primitive task is computing the intensity of a single voxel.

This project did not consider the communication phase of Foster's Design Methodology, since the FDK algorithm requires no communication between voxels. The agglomeration phase involves combining small tasks into somewhat larger tasks so as to improve performance or make the program easier to code. Given low degree of communication overhead in the FDK algorithm and the intuitive CUDA programming model, this project also eliminated the agglomeration phase.

The final stage in Foster's Design Methodology, the mapping stage, assigns tasks to processors. To adequately explain this stage, this report must first describe the CUDA programming model. CUDA contains several layers of task organization, and represents primitive tasks as "threads." In this project, each thread computed a single voxel intensity. CUDA groups threads together into "blocks," and further groups blocks of threads together into a "grid."

Figure 7 in Appendix A illustrates this thread-grouping model.

For its mapping stage, this project combined fast performance with scalability. NVIDIA states that a GPU program performs ideally with 256 threads per block (a 16x16 block). With this in mind, this project created D groups of $\frac{D^2}{16^2}$ thread blocks. Figure 8 in Appendix A illustrates this thread model. To use a numerical example, suppose the output is a 64x64x64 voxel region (64^3 total voxels). Thus, $D = 64$, so the GPU will contain 64 groups of $\frac{64^2}{16^2} = 16$ thread blocks. Therefore the grid dimension is 16x64. Double-checking to ensure each thread maps to a voxel, observe that:

$$16\ blocks \times 64\ blocks \times 16^2 \left(\frac{threads}{block} \right)^2 = 262144\ threads = 64^3\ threads$$

The number of threads therefore equals is the total number of voxels. Figure 9 in Appendix A diagrams this example.

For a fixed dimension, the reader may find this mapping unintuitive. For example, a programmer could more easily code a grid of D x D blocks of D threads. However, a large output space makes this scheme problematic because the number of threads in a block may exceed the limit of threads per block. This project uses a superior approach because it allows the program to scale correctly without exceeding the maximum number of threads per block.

In addition to requiring a mapping of primitive tasks to processors, this project also demanded a one-to-one mapping of a thread's position within a block and grid to a three-dimensional coordinate in a cubic space. NVIDIA provides formulas to convert between a thread's position in a block to its global identity (called its thread ID). This project then created simple functions to map the thread ID to $\{x, y, z\}$ coordinates. Figure 10 in Appendix A shows these mapping functions. Having completed the four steps of Foster's Design Methodology, this project had successfully parallelized the CPU code for the GPU.

With a complete parallel design, this project began programming and testing the GPU implementation with the CUDA emulator. Verifying the GPU program was straightforward: its output was correct only if it matched the CPU program's output. This project verified the GPU program by writing the output values of both the CPU and GPU programs to file and then using the UNIX program `cmp` to display lines that differed in these files. When no values differed (or differed by a small threshold, due to error), this project considered the GPU program equivalent to the CPU program. Eventually, the GPU program completed its development cycle and passed all verification tests. The next chapter, Chapter 5, demonstrates the GPU program's timing results, as well as their significance.

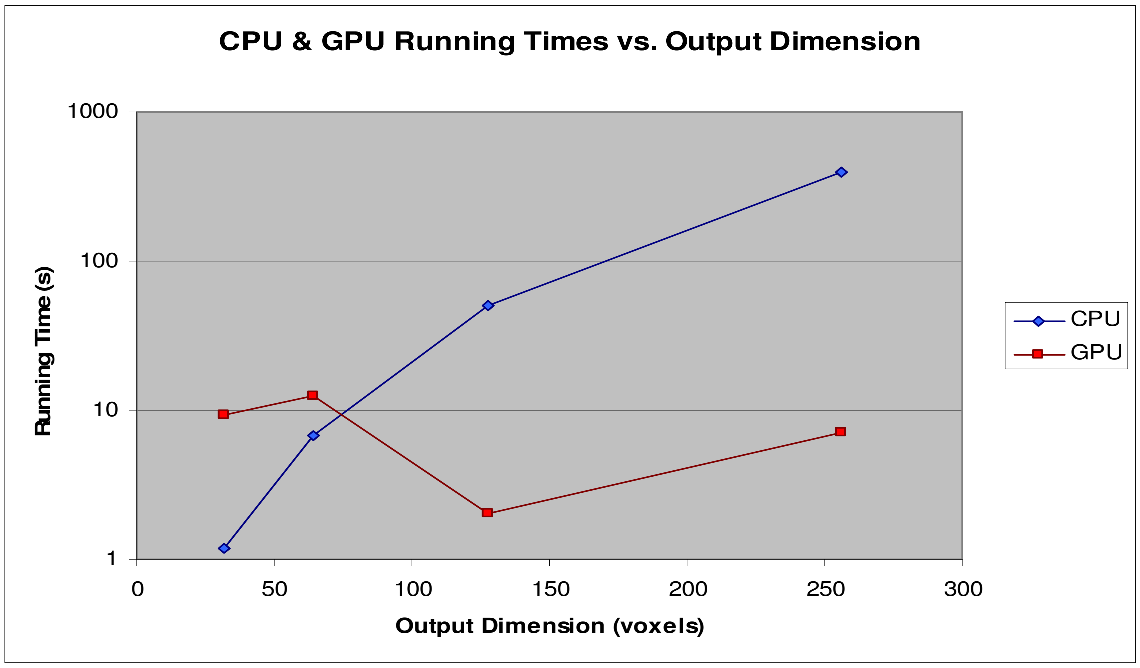
CHAPTER 5: RESULTS & RESULT ANALYSIS

After developing and verifying the GPU implementation in the emulator, this project executed the GPU program on physical hardware. This chapter shows running times for both the GPU and CPU programs with various output sizes, and also explains results both in terms of their significance and the conditions under which this project obtained them. Overall, the results illustrate the viability of GPU-accelerated reconstruction. However, these results are preliminary and require refinement to accurately reflect the degree of speedup for GPU reconstruction.

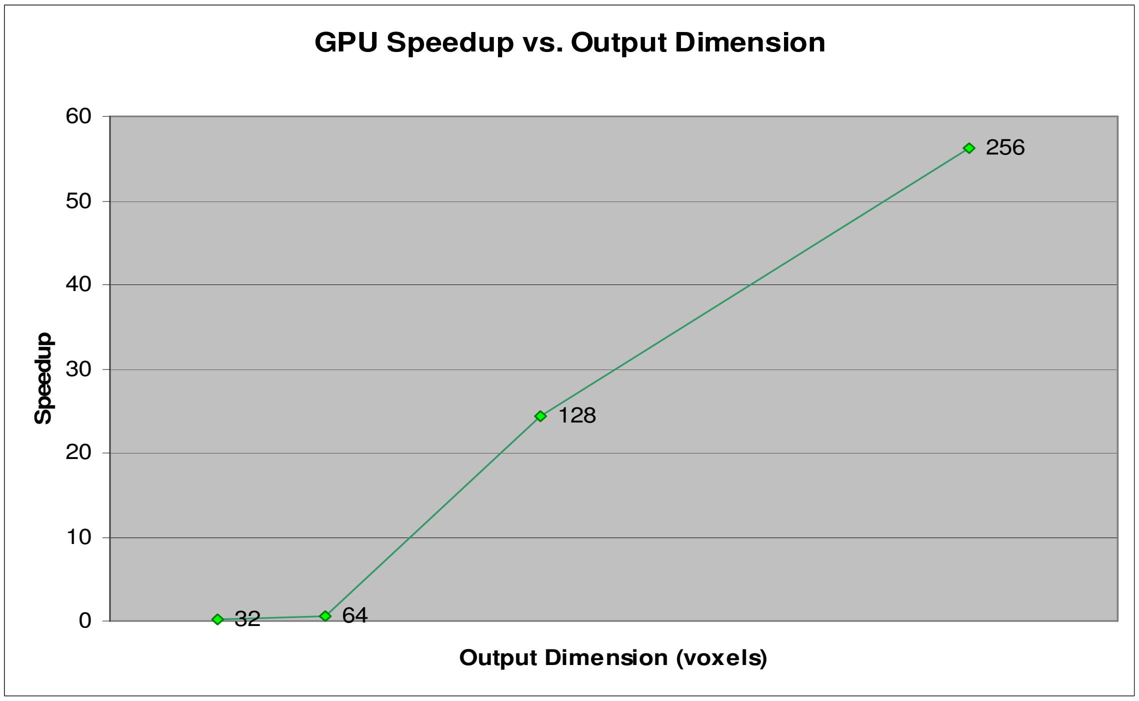
Before gathering timing data, this project ran the GPU program on the NVIDIA 8800 to detect any possible hardware compatibility issues (NVIDIA's emulator is still in its beta stage and does not always accurately simulate the card's behavior). This worked without any problems. Next, the project gathered timing results with several output resolutions. Table 1 shows the averaged timing data for four output resolutions, as well as the calculated speedup with the GPU. Graph 5 shows running times on the CPU and GPU speedup as a function of output resolution, while Graph 6 shows speedup on the GPU as a function of output resolution. Appendix B has a more detailed chart, while Appendix C contains more detailed versions of the graphs.

Dimension	Running Time (s)		Speedup
	CPU	GPU	
32	1.20	9.25	0.13
64	6.82	12.56	0.54
128	49.96	2.05	24.41
256	396.13	7.04	56.26

TABLE 1 – Timing data and speedup vs. dimension (created by author)



GRAPH 5 – CPU and GPU running time as a function of output resolution, logarithmic scale (created by author)



GRAPH 6 - GPU speedup as a function of output resolution (created by author)

Timing results varied widely for this project because of their dependence on I/O operations (i.e. disk or memory access). These times included time spent loading scans from disk. For the CPU program, this meant loading scan files and moving them into local memory. The GPU program incurred the additional overhead of moving scans from local memory to GPU memory, and intensity values from GPU memory to CPU memory. Furthermore, the first execution of the program after compiling (for both CPU and GPU versions) typically took much longer than subsequent executions. This occurred because the scans cached after the first execution and thus loaded quicker in future executions. The timing results in this report do not reflect the first run of either program since these times were outliers.

Running times on the GPU for small output resolutions seem inconsistent with the assertion that GPUs map well to backprojection. In fact, with a small output space, the GPU program took much longer to execute than the CPU program. However, this behavior is common in parallel programs because of the overhead of parallelization. With so few voxel intensities to calculate at low resolution, the overhead of memory setup and transfer dominates the amount of computation for each thread. This overhead diminishes as the output size increases, since the time to compute intensities exceeds the time to set up the GPU. The GPU program's running time hit a minimum with an output size around 128x128x128 voxels and then linearly increased.

The speedup figures demonstrate that the FDK algorithm strongly benefits from parallelization on the GPU. The GPU generated intensities for over two million voxels in roughly two seconds and intensities for sixteen million voxels in eight seconds. These times are orders of magnitude less than those from the CPU program, and are comparable to those achieved by Xu and Mueller. More importantly, the speedup increased as the output size grew, meaning this program scales well on the GPU. The final chapter, Chapter 6, illustrates the importance of these results in a broader scope.

Chapter 6: Conclusion & Recommendations for Future Research

This section serves to recapitulate the findings of this project. This includes both reiterating the results of this project and assessing how well these results accomplished the intended goals of this project. Lastly, this section proposes ideas for future work on this project, as well as guidelines for how others should use its findings.

This project hoped to improve processing times for CT scan reconstruction via graphics hardware. It failed in that it unsuccessfully reconstructed the torso phantom in three dimensions. However, it did successfully convert a slow uniprocessor backprojection program to a fast, parallelized program on the NVIDIA 8800 GPU. The GPU program ran 25 times faster than the CPU program for a small output size of 128x128x128 voxels and over 50 times faster for an average output size of 256x256x256 voxels. The duration of the backprojection process decreased from minutes on the CPU to seconds on the GPU. These results are tentative, though, since the program is unfinished.

This project cannot establish any absolute claims about the performance of GPUs for CT scan reconstruction since it produced invalid output. At a critical level, this project took a broken program, parallelized it, and sped it up to create a faster broken program. Thus, one could view this project simply as a parallelization exercise for the researcher. Yet, this view naïvely undervalues the process of parallelization.

The greatest strength of this project is that it produced identical CPU and GPU output. For most applications that can benefit from parallelization on a GPU, the programmer typically begins with a working CPU code base and must convert this code to GPU code. Therefore, the important lesson of this project was not learning medical imaging algorithms but learning how to take slow sequential code and convert it to fast parallel code on the GPU. This skill will prove useful with the increasing ubiquity of parallel computer architectures.

Other programmers and researchers should find this project useful because it provides a non-expert view of CUDA. CUDA has only existed for about six months and few, except NVIDIA employees, have documented their experiences with it. Overall, the new programming paradigm worked well for this application. This project did not have significant problems porting CPU code to fit the CUDA model, and had no problems moving from the CUDA simulator to the physical hardware. At the same time, this project used an easily parallelizable algorithm. CUDA is slower and less intuitive to program for algorithms requiring thread communication. Therefore, this project cannot assert that all parallel programs will run faster with CUDA, yet it can make stronger claims about the FDK algorithm's performance.

Despite its output, this project illustrates much about the potential of creating a GPU-accelerated reconstruction system with CUDA. The speedup results show that GPUs perform well with data parallel algorithms, especially those with large amounts of data to process. Even though this project cannot show correct output to prove this claim, its results, along with those of previous researchers, indicate that the GPU model fits well with the FDK algorithm. The FDK algorithm clearly expresses a high degree of data parallelism because it contains no inter-iterational dependencies. This project believes that its backprojection program is correct, but that a filtering program must first process its input. Even if this project's reconstruction algorithm is not the correct FDK algorithm, it closely resembles FDK algorithms described in medical imaging literature. As such, a complete GPU reconstruction system may not achieve the same speedup as this project, but should receive a speedup nonetheless.

Given that the reconstruction program is still incorrect, this project has several suggestions for anyone continuing this research. This includes ways in which hospitals should use this project, particularly to maintain an ethically and socially responsible setting. Though the GPU sped up the backprojection code by orders of magnitude, further code optimizations could make this speedup even greater. First, CUDA cards have varying memory latencies depending on data types. In particular, CUDA chips store constant values in their fastest memory. This project denoted most of the constant values as such, but the compiler complained that the card could not put some of the values in constant memory. The program compiled when it allocated these values in shared memory, which is still fast, but not as fast as constant memory. Apparently NVIDIA documents a workaround for this problem in their latest CUDA documentation. Using constant memory for all constants would decrease the overall running time, but probably not by a significant amount.

A future researcher could also optimize the task mapping strategy in the GPU code. Currently, the methods for calculating the thread ID, the $\{x,y,z\}$ coordinates and output array location use fairly expensive operations such as modulus, division, and several multiplications. Another mapping of tasks to processors, keeping in mind the thread per block limit, could reduce the number of calculations required to find the ID, coordinates, and array location.

In addition to hardware optimizations, the GPU program could further benefit from algorithmic optimization. This project implemented the most straightforward form of the FDK algorithm. This algorithm is $O(D^3 \bullet S)$, where D is the length of one side of the output space and S is the total number of input scans. However, other researchers have created less expensive versions of the FDK algorithm, such as Basu and Bresler's $O(N^2 \log_2 N)$ implementation [21], that could lessen the overall running time even further. These algorithms may or may not map easily to the CUDA programming model.

This project's GPU implementation worked up to an output size of 484x484x484, at which point the GPU ran out of memory. To get results for greater output spaces, a researcher would have to create a multi-pass system. This means that the program would stream some number of voxels within the GPU's memory limit to the GPU for processing, process them, stream them back into CPU memory, and then repeat this process until the all voxels had an intensity value. This allows for a potentially infinite output size, but would run slower than the current program due to memory transfers.

Clearly, this project should generate correct 3D output. Medical imaging papers indicate that backprojection only works with filtered scan data. A future researcher has two ways of obtaining filtered scan data: filtering the data manually, or finding pre-filtered data from some other source. This project searched for filtered scan data on the internet with no results, but perhaps other resources, such as hospitals or other universities, could provide such data. If not, a future researcher must create a filtering stage in the backprojection program. According to other members of Prof. Skadron's research group, the best way to filter data involves performing a Fourier transform on the data, thus moving it to the frequency domain, then multiplying the transformed data by some filtering kernel, then performing an inverse Fourier transform to move the filtered data back to the spatial domain. A programmer can use the FFT (Fast Fourier Transform) library in the CUDA SDK to perform the transformation steps required for filtering. If the incorrect output does not stem from filtering, a future researcher should contact a medical imaging expert, perhaps Xu or Mueller, to get advice on how to correctly implement reconstruction algorithms.

If a future researcher ever completes a fully operational reconstruction system, hospitals must use this system judiciously. First, doctors should ensure that they screen patients to avoid complications with exposure to contrast dye or radiation. Secondly, hospitals must secure all computers running their reconstruction software so that outsiders cannot access confidential patient information or maliciously attack the system. Most importantly, doctors should recall that, while 3D CT reconstructions provide greater detail than 2D images, they are not a panacea. Doctors should still use all resources available to treat a patient and not rely solely on scans.

Nevertheless, this project provides great benefits for hospital radiology departments everywhere. Using GPUs to reconstruct CT scans provides hospitals with two distinct advantages over current uniprocessor reconstruction systems: faster system performance and lower system cost. Faster reconstructions translate to more efficient hospital operation. Faster reconstructions should also encourage the use of CT. For patients, increased use of CT should equate to more frequent and earlier diagnoses, which potentially saves lives.

WORKS CITED

[1] Suetens, Paul. Fundamentals of Medical Imaging. New York: Cambridge University Press, 2002.

[2] Goode, Allen and Eric Huffman. "3D Reconstruction Project Initial Scope Presentation." Olsson Hall, University of Virginia. 31 August 2006.

[3] Sakamoto, Masaharu, et al. "An Implementation of the Feldkamp Algorithm for Medical Imaging on Cell." IBM Corporation Systems and Technology Group. 2005.

[4] Owens, John. "Streaming Architectures and Technology Trends." GPU Gems 2. Ed. Matt Pharr. Upper Saddle River, NJ: Addison-Wesley, 2005.

[5] Quinn, Michael J. Parallel Programming in C with MPI and OpenMP. New York: McGraw-Hill, 2004.

[6] GPGPU. 2006. 28 September 2006. Available HTTP: http://www.gpgpu.org

[7] "NVIDIA CUDA Homepage." 7 November 2006. NVIDIA Corporation. 6 February 2007. Available HTTP: http://developer.nvidia.com/object/cuda.html

[8] Xu, Fang and Klaus Mueller. "Accelerating Popular Tomographic Reconstruction Algorithms on Commodity PC Graphics Hardware." IEEE Transaction of Nuclear Science. 2005.

[9] "UVA Medical Center Ranks as One of the Nation's Top 100 Hospitals." University of Virginia Health System: Public Relations. 6 March 2006. 29 September 2006. Available HTTP: http://www.healthsystem.virginia.edu/internet/news/archives06/top_100_hospital_2006.cfm

[10] "Computed Tomography Angiography." 12 May 2005. Radiological Association of North America. 29 September 2006. Available HTTP: http://radiologyinfo.org/en/info.cfm?pg=angioct&bhcp=1

[11] "Multislice CT Angiogram." Angioplasty.org. 29 September 2006. Available HTTP: http://www.ptca.org/imaging/msct.html

[12] "UNSCEAR 2000 Report to the General Assembly." United Nations Scientific Committee on the Effects of Atomic Radiation. 30 October 2006. Available HTTP: http://www.unscear.org/docs/reports/gareport.pdf

[13] Wolbarst, Anthony Brinton. Looking Within: How X-Ray, CT, MRI, Ultrasound, and Other Medical Images Are Created, and How They Help Physicians Save Lives. Berkeley: University of California Press, 1999.

[14] Computed Tomography: Its History and Technology. Siemens AG, Medical Solutions. 14 September 2004. 1 March 2007. Available HTTP: http://www.medical.siemens.com/siemens/zh_CN/gg_ct_FBAs/files/brochures/CT_History_and_Technology.pdf

[15] Feldkamp, L.A., L.C. Davis, and J.W. Kress. "Practical Cone-Beam Algorithm." Journal of the Optical Society of America. 1.6 (1984): 612.

[16] Reimann, David A., et al. "Parallel Computing Methods for X-Ray Cone Beam Tomography with Large Array Sizes." IEEE Nuclear Science Symposium. 1997.

[17] Xu, Fang and Klaus Mueller. "Towards a Unified Framework for Rapid 3D Computed Tomography on Commodity GPUs." IEEE Medical Imaging Conference, Portland, OR. 2003.

[18] Xu, Fang and Klaus Mueller. "Ultra-Fast 3D Filtered Backprojection on Commodity Graphics Hardware." IEEE International Symposium on Biomedical Imaging, Washington D.C. 2004.

[19] NVIDIA CUDA Compute Unified Device Architecture Programming Guide. NVIDIA Corporation. Version 0.8. 12 February 2007. 1 March 2007. Available HTTP:http://developer.download.nvidia.com/compute/cuda/0_8/NVIDIA_CUDA_Programming_Guide_0.8.pdf

[20] Goddard, Iain, and Marc Trepanier. "High-speed cone-beam reconstruction: an embedded systems approach." Medical Imaging 2002: Visualization, Image-Guided Procedures, and Display, (2002): pp. 483-491. Available HTTP: http://spiedl.aip.org/getpdf/servlet/GetPDFServlet?filetype=pdf&id=PSISDG004681000001000483000001&idtype=cvips&prog=normal

[21] Basu, Samit and Yoram Bresler. "O($N^2 \log_2 N$) Filtered Backprojection Reconstruction Algorithm for Tomography." IEEE Transactions on Image Processing, vol. 9, no. 10. October 2000. Available HTTP: http://ieeexplore.ieee.org/iel5/83/18833/00869187.pdf

APPENDIX A – FIGURES

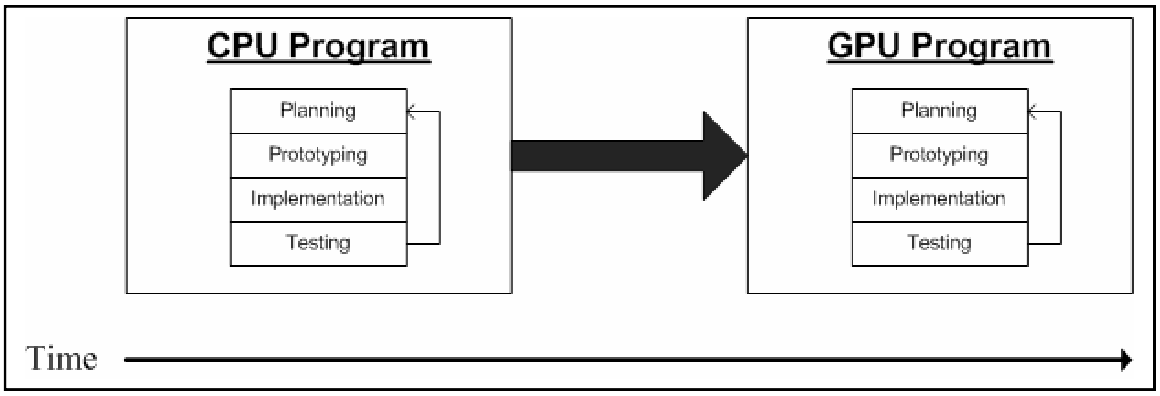


FIGURE 3 – Initial software development cycle (drawn by author)

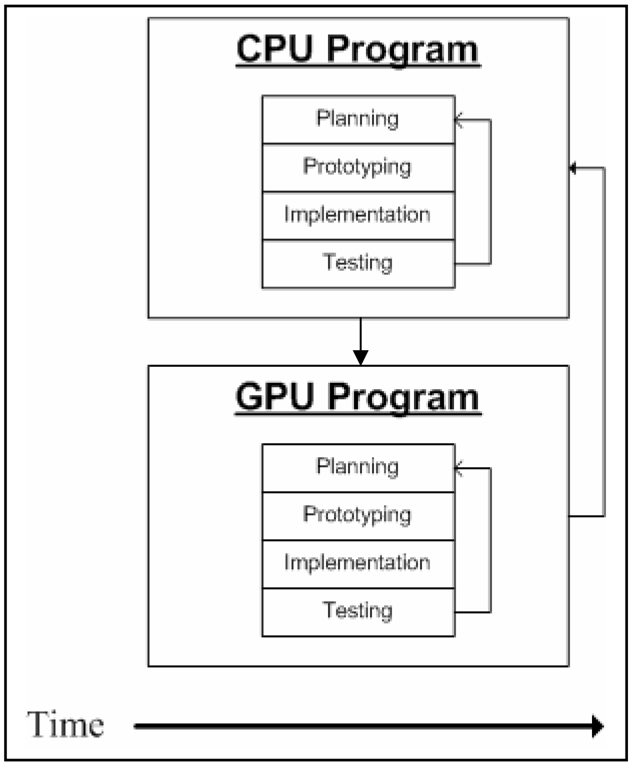


FIGURE 4 – Modified software development cycle (drawn by author)

2.4 Algorithm

We selected a standard Feldkamp CBR algorithm[12] as a general benchmark baseline. The value of the desired output individual image voxels I(x,y,z) is the sum over all projection angles β of the filtered projection pixels P(u,v,β) times a weighting factor w(x,y,β). The geometry (Fig 5.) and image voxels can be calculated as shown below:

$$I[x,y,z] = \sum_{\beta} P[u(x,y,\beta), v(x,y,\beta), \beta] \cdot w(x,y,\beta) \tag{1}$$

$$s = x\cos\beta + y\sin\beta \tag{2}$$

$$t = -x\sin\beta + y\cos\beta \tag{3}$$

$$w = \left[\frac{SOD}{(SOD - s)}\right]^2 \tag{4}$$

$$u = \frac{t * SID}{(SOD - s)} \tag{5}$$

$$v = \frac{z * SID}{(SOD - s)} \tag{6}$$

where:
I = 3-Dimensional output image array with x, y, and z-axes.
P = 2-Dimensional filtered projection data with u and v axes taken at all the various projection angles (β). This is the planar array of sensor projection data taken at angle β.
β = Projection angle
s = Projected image voxel location onto the s-axis
t = Projected image voxel location onto the t-axis
w = Weighting factor based on source distance to image voxel location.
SID = Source-to-image (detector) distance.
SOD = Source-to-object (center of rotation) distance.
u = u-axis value for the P data for a given voxel location and projection angle.
v = v-axis value for the P data for a given voxel location and projection angle.

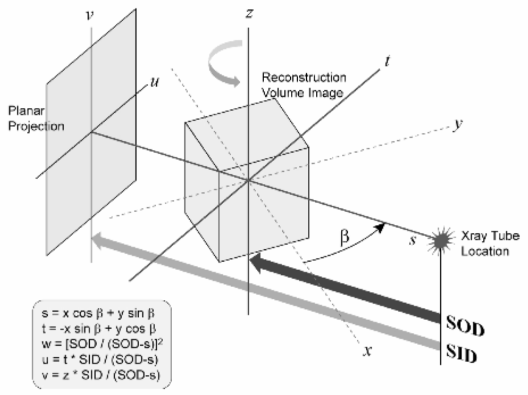


Fig. 5: Diagram of Geometry.

FIGURE 5 – Complete FDK algorithm [20]

```
for(x = 0; x < OUT_DIM; x++) {
  for (y = 0; y < OUT_DIM; y++) {
    for (z = 0; z < OUT_DIM; z++) {
      intensity = 0.0f;
      for (scan_num = start; scan_num < start+RANGE; scan_num++) {
        beta = scan_num*1.5f*DEGTORAD;
        s =  x * cosf(beta) + y * sinf(beta);
        t = -x * sinf(beta) + y * cosf(beta);
        d = D / (D - s);
        u = t * d;
        v = z * d;
        w = powf(d, 2);
        data_u = (int)u;
        data_v = (int)v;

        if ((data_u >= 0 && data_u < OUT_DIM) &&
            (data_v >= 0 && data_v < OUT_DIM)) {
          intensity += w*scan_data[scan_num][data_u][data_v];
        }
      }
      intensity_data[x][y][z] = intensity;
    }
  }
}
```

FIGURE 6 – C implementation of FDK algorithm (created by author)

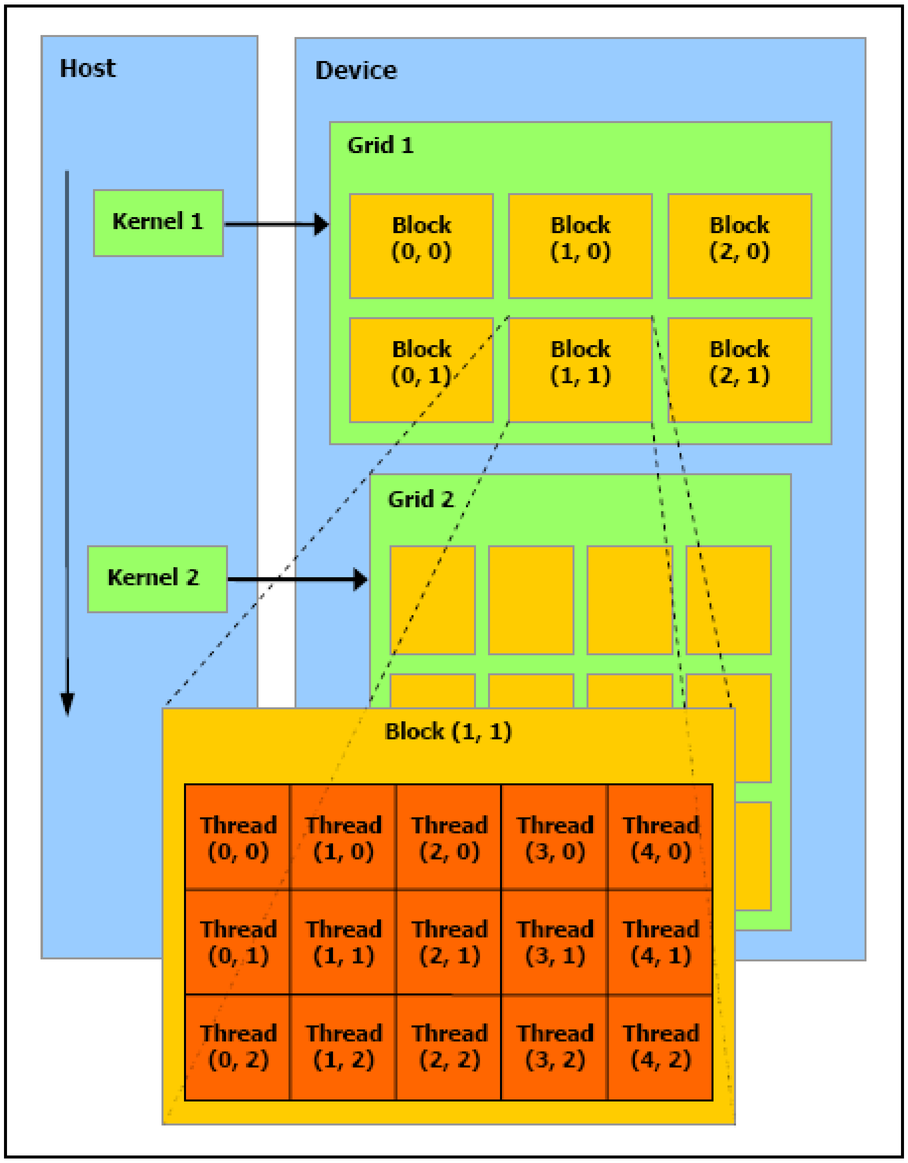


FIGURE 7 – NVIDIA CUDA task organization model [19]

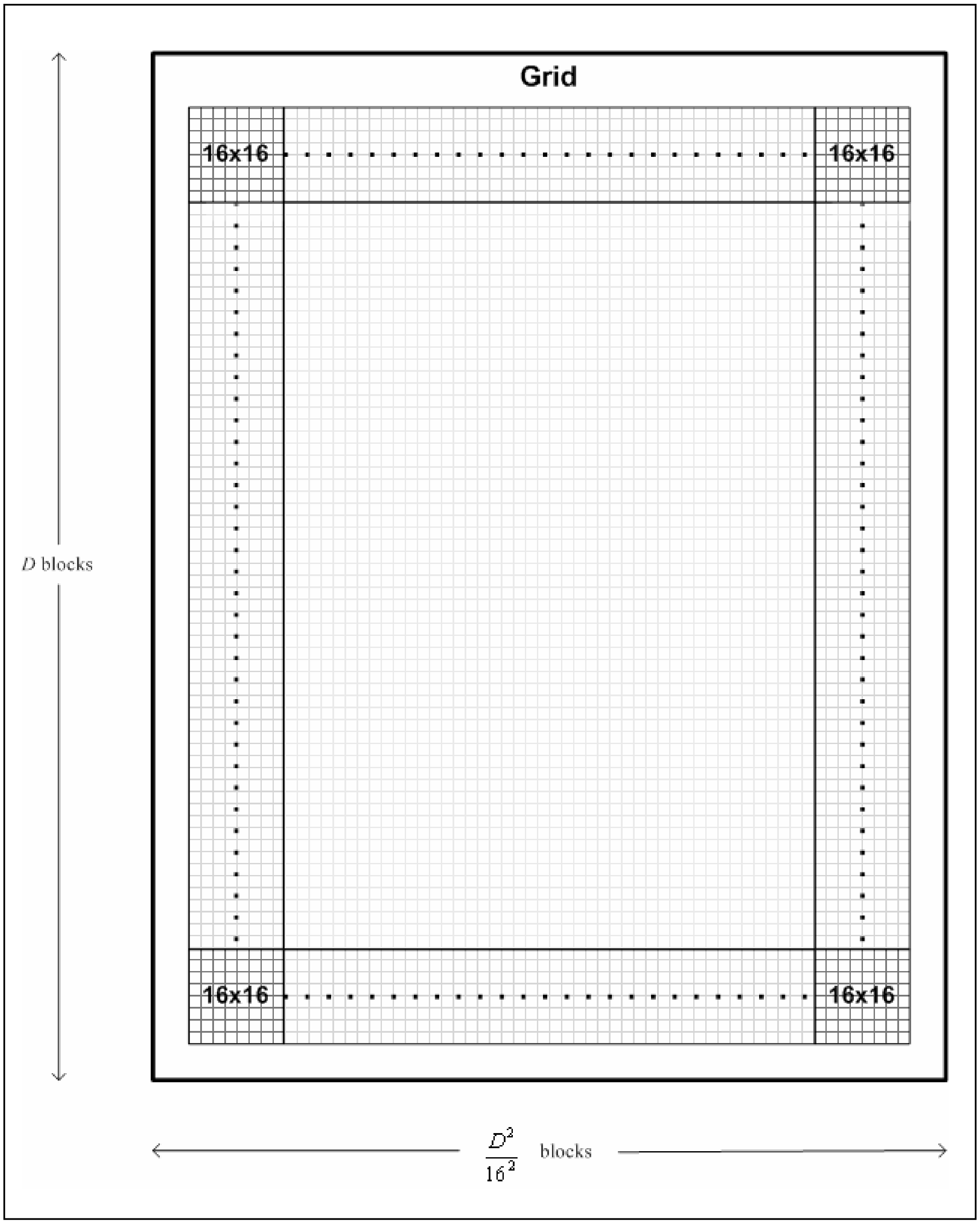


FIGURE 8 – General CUDA thread grouping model for the FDK algorithm (drawn by author)

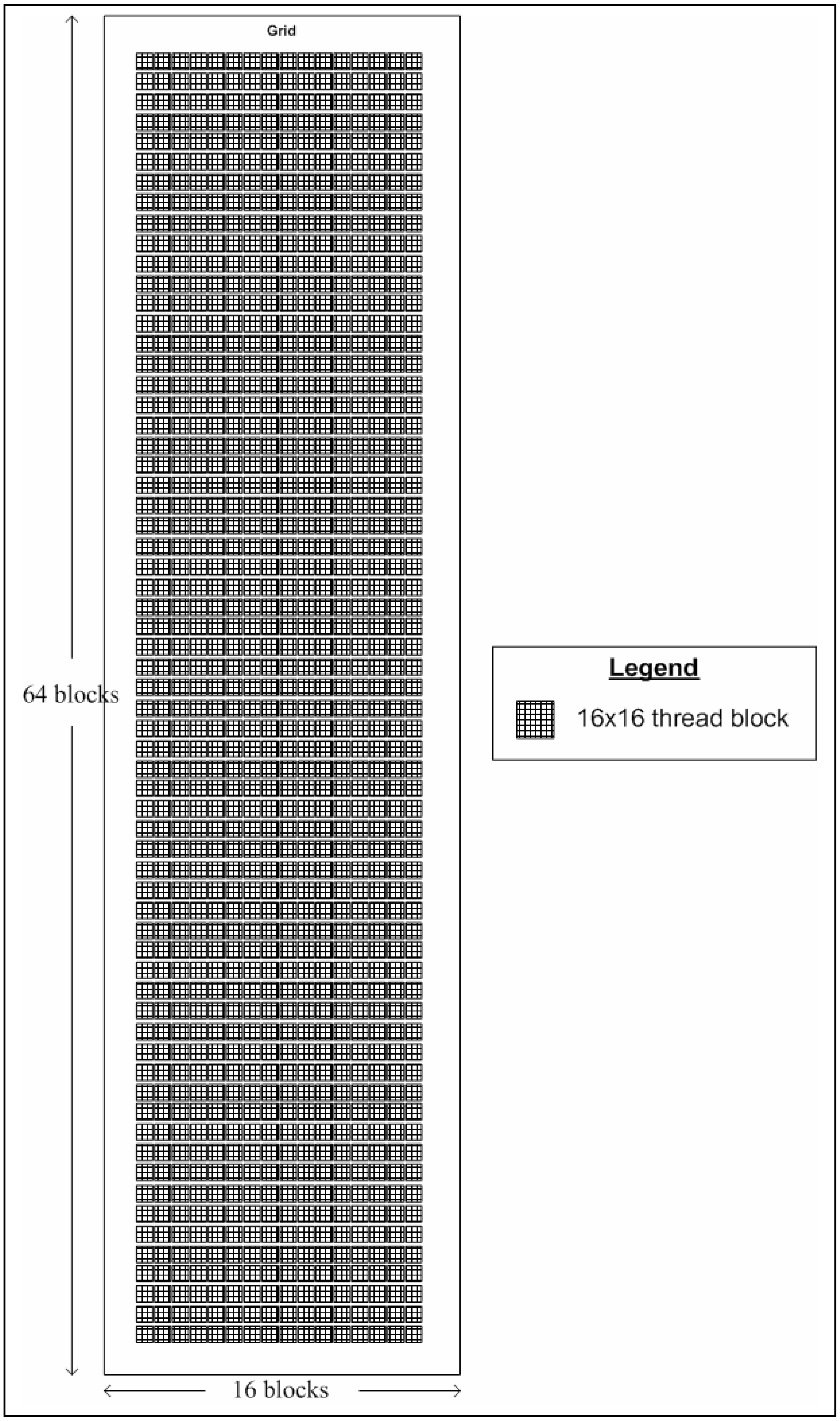


FIGURE 9 – CUDA thread grouping model for the FDK algorithm for 64x64x64 output resolution (drawn by author)

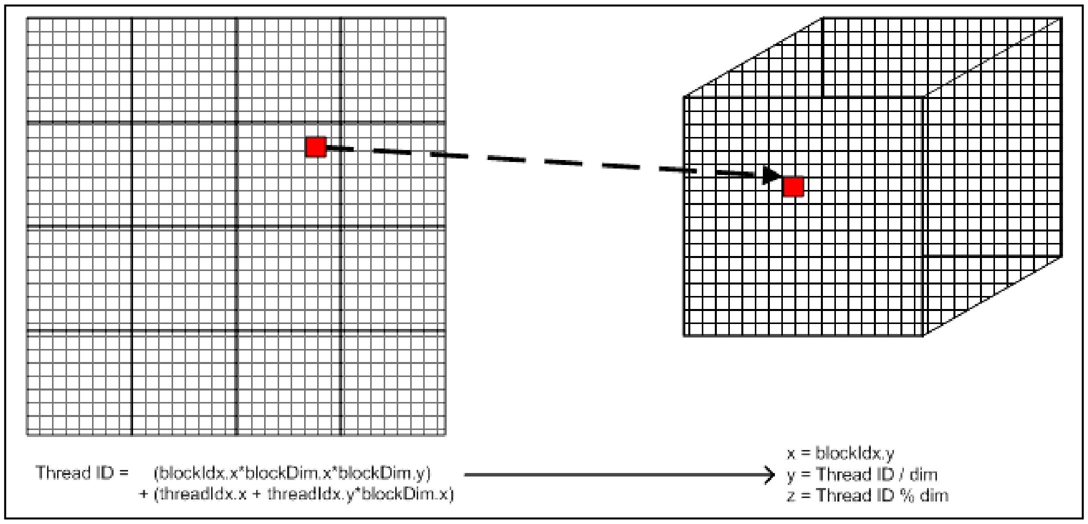

Thread ID = (blockIdx.x*blockDim.x*blockDim.y)
+ (threadIdx.x + threadIdx.y*blockDim.x)
x = blockIdx.y
y = Thread ID / dim
z = Thread ID % dim

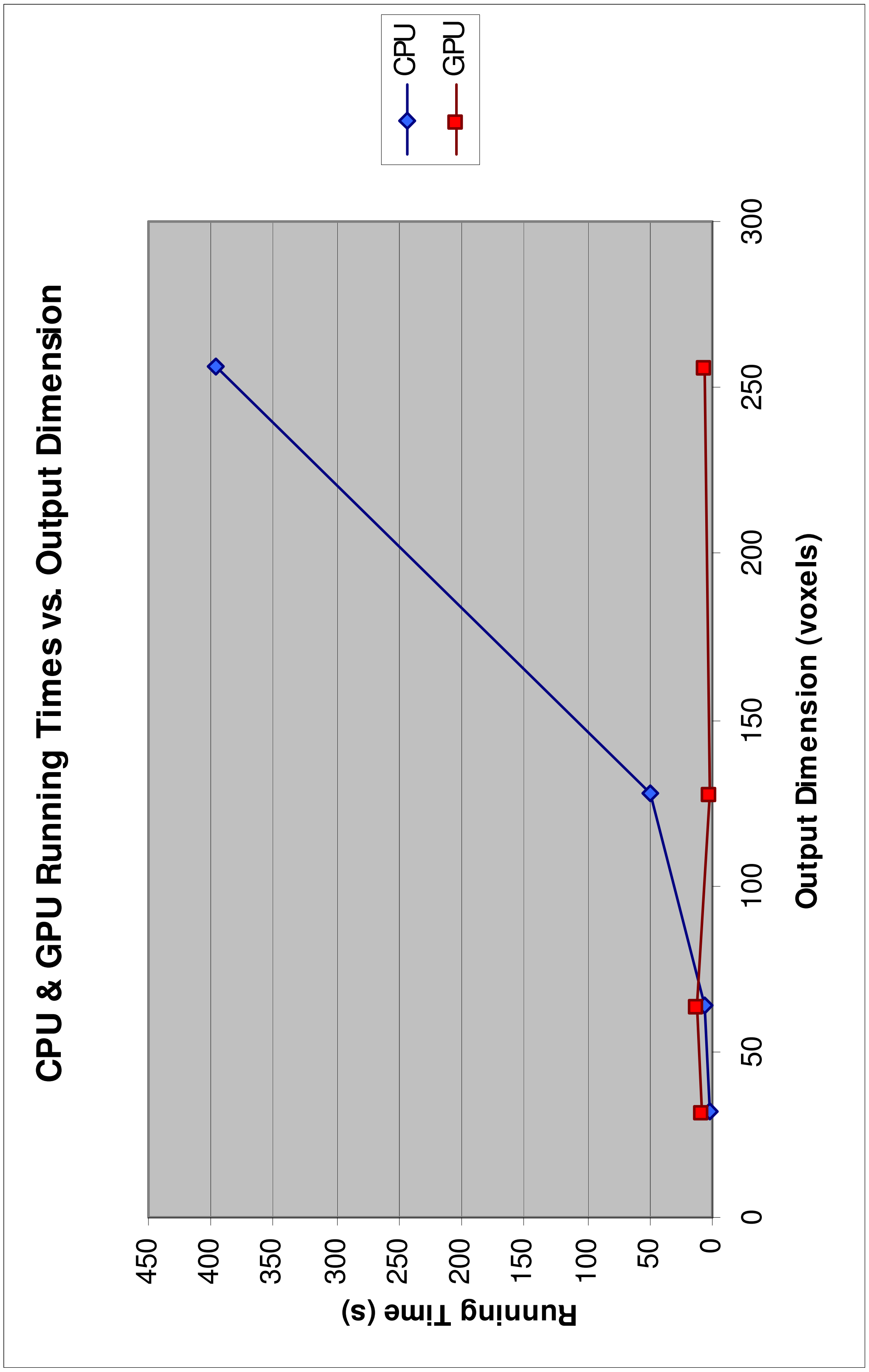
FIGURE 10 – Formula for thread ID and mapping of thread ID to $\{x,y,z\}$ coordinates (drawn by author)

APPENDIX B – TABLES

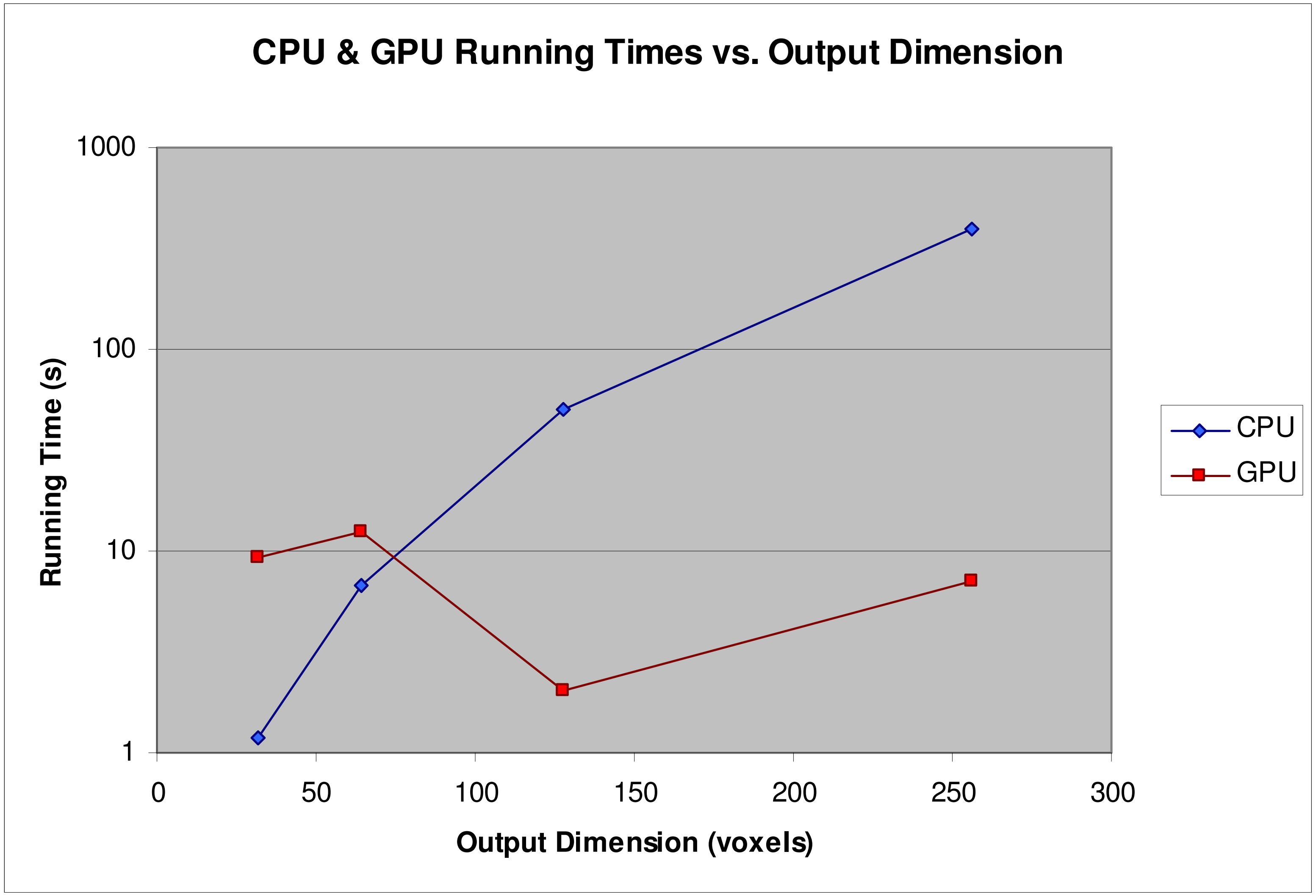
Dimension	Running Time (s) CPU	GPU	Speedup
32	1.20	8.99	0.13
	1.19	7.70	0.15
	1.20	11.05	0.11
64	7.17	13.00	0.55
	6.74	12.27	0.55
	6.55	12.42	0.53
128	51.41	2.05	25.11
	49.30	2.02	24.47
	49.17	2.08	23.66
256	394.34	7.03	56.09
	393.97	7.02	56.16
	400.07	7.08	56.52

TABLE 2 – Running times and speedup for CPU and GPU programs at four output resolutions (created by author)

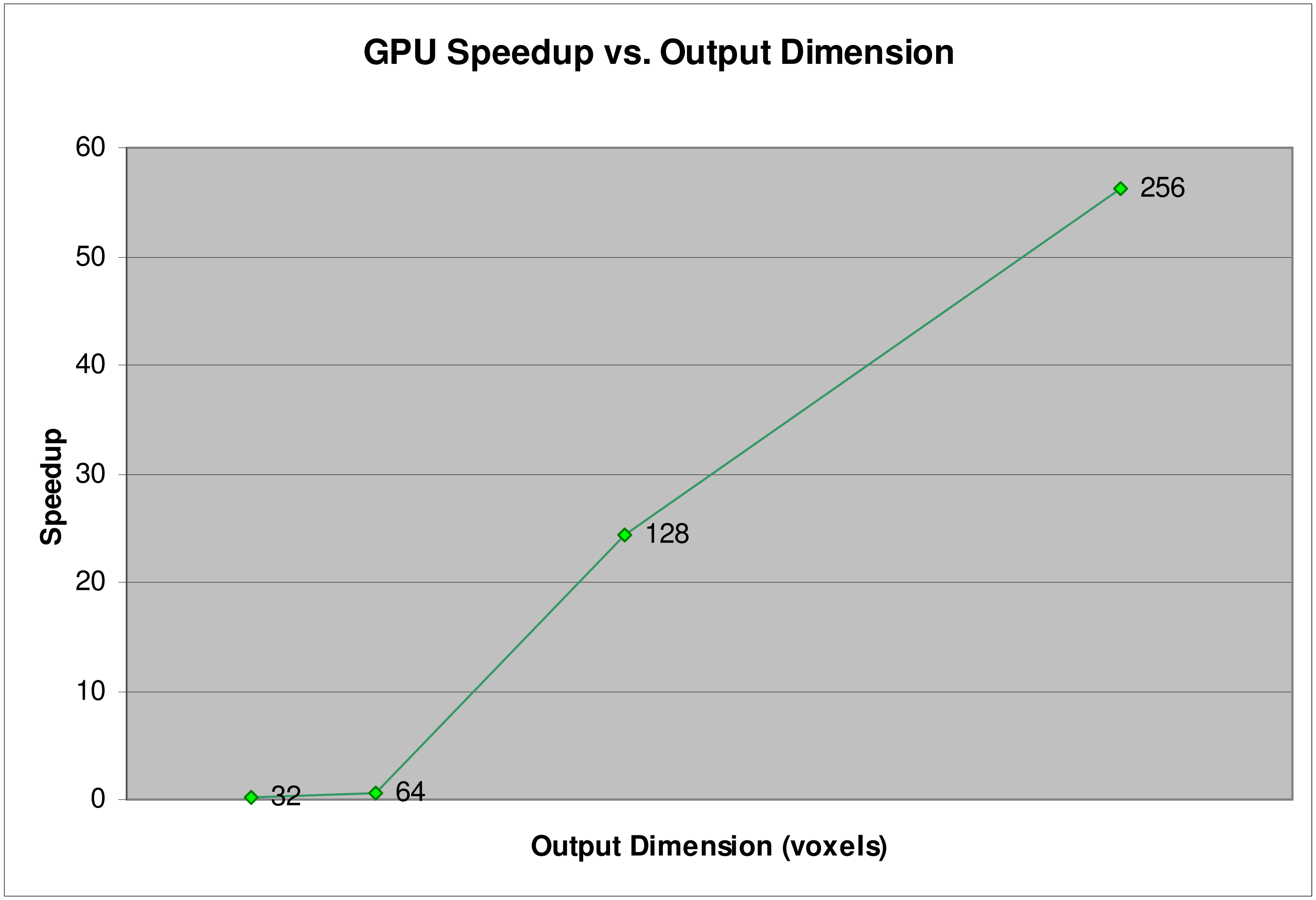
APPENDIX C – GRAPHS



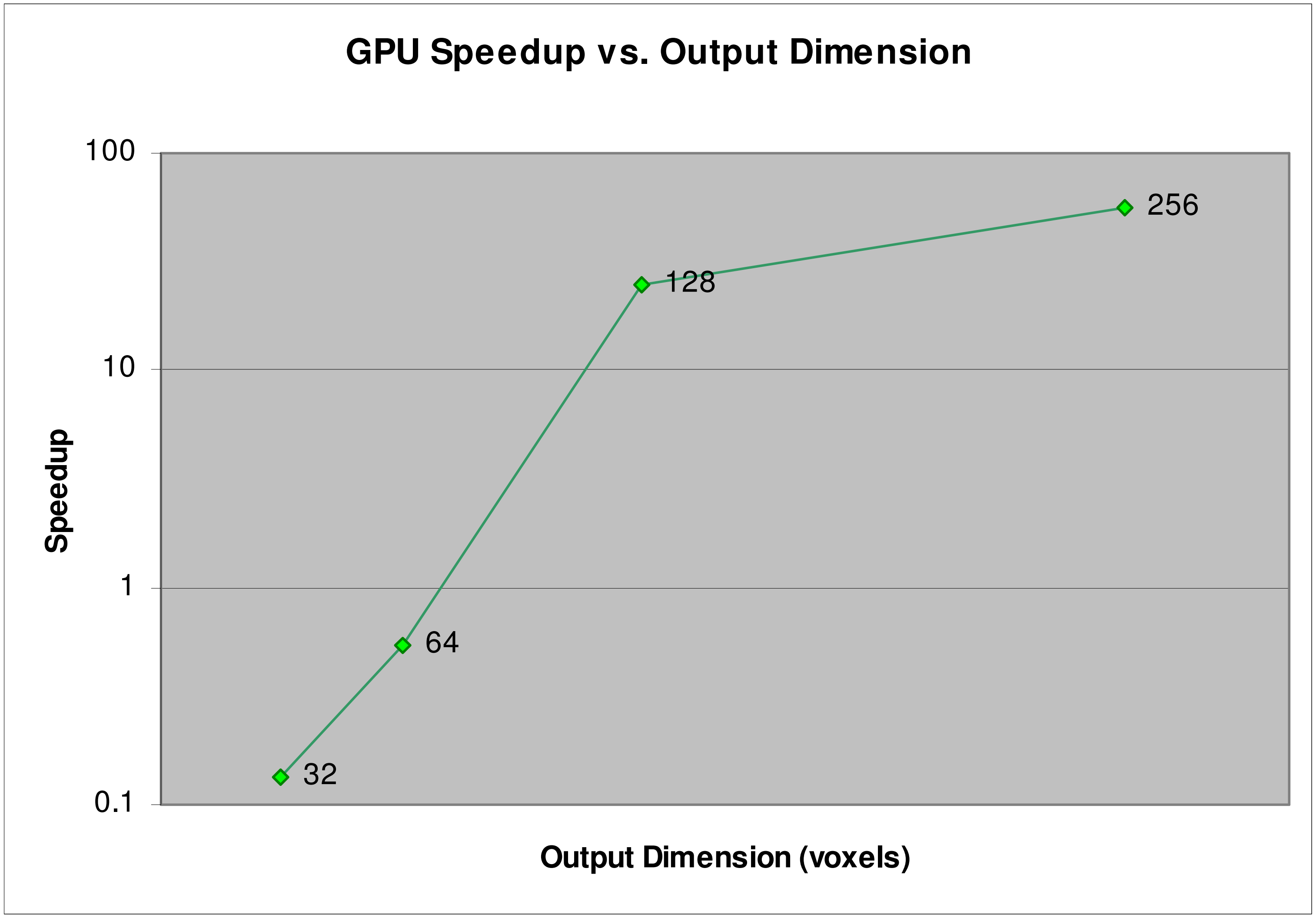
GRAPH 7 - GPU and CPU running times as a function of output resolution, linear scale (created by author)



GRAPH 8 - GPU and CPU running times as a function of output resolution, logarithmic scale (created by author)



GRAPH 9 - GPU speedup as a function of output resolution, linear scale (created by author)



GRAPH 10 - GPU speedup as a function of output resolution, logarithmic scale (created by author)

APPENDIX D – LISTINGS

```
------------------
System Information
------------------
Time of this report: 3/24/2007, 14:41:38
       Machine name: SKADRONDELL4
   Operating System: Windows XP Professional (5.1, Build 2600) Service Pack 2 (2600.xpsp.061012-0254)
           Language: English (Regional Setting: English)
System Manufacturer: Dell Inc.
       System Model: Dell DXG061
               BIOS: Phoenix ROM BIOS PLUS Version 1.10 1.1.3
          Processor: Intel(R) Core(TM)2 CPU          6300  @ 1.86GHz (2 CPUs)
             Memory: 1022MB RAM
          Page File: 411MB used, 2048MB available
        Windows Dir: C:\WINDOWS
    DirectX Version: DirectX 9.0c (4.09.0000.0904)
DX Setup Parameters: Not found
     DxDiag Version: 5.03.2600.2180 32bit Unicode

---------------
Display Devices
---------------
        Card name: NVIDIA GeForce 8800 GTX
     Manufacturer: NVIDIA
        Chip type: GeForce 8800 GTX
         DAC type: Integrated RAMDAC
       Device Key: Enum\PCI\VEN_10DE&DEV_0191&SUBSYS_039C10DE&REV_A2
   Display Memory: 768.0 MB
     Current Mode: 1280 x 1024 (32 bit) (75Hz)
          Monitor: Plug and Play Monitor
  Monitor Max Res: 1600,1200
      Driver Name: nv4_disp.dll
   Driver Version: 6.14.0010.9773 (English)
      DDI Version: 9 (or higher)
Driver Attributes: Final Retail
 Driver Date/Size: 2/2/2007 15:25:00, 5365504 bytes
      WHQL Logo'd: No
  WHQL Date Stamp: None
              VDD: n/a
         Mini VDD: nv4_mini.sys
    Mini VDD Date: 2/2/2007 15:25:00, 5957024 bytes
Device Identifier: {D7B71E3E-42D1-11CF-4355-962303C2CB35}
        Vendor ID: 0x10DE
        Device ID: 0x0191
        SubSys ID: 0x039C10DE
      Revision ID: 0x00A2
      Revision ID: 0x00A2

------------------------
Disk & DVD/CD-ROM Drives
------------------------
      Drive: C:
 Free Space: 65.5 GB
Total Space: 76.0 GB
File System: NTFS
      Model: n/a

      Drive: D:
      Model: HL-DT-ST DVD-ROM GDRH10N
     Driver: c:\windows\system32\drivers\cdrom.sys, 5.01.2600.2180 (English), 8/4/2004 08:00:00, 49536 bytes
```

LISTING 1 – System specifications for Dell XPS system with NVIDIA 8800 (generated by dxdiag.exe)

LISTING 2 – Complete code listing for sequential (CPU) backprojection code (created by author)

```
//////////////////////////////////////////////////////////////////
//                                                              //
// Drew Maier                                                   //
// 4th Year Thesis Project                                      //
//                                                              //
// feldkamp.c                                                   //
// Sequential (CPU) version of the FDK backprojection algorithm //
//                                                              //
//////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

const int SCAN_DIM = 1024;
const int AREA = 1048576;
const int MAX_INTENSITY = 4096;
const float DEGTORAD = 0.01745f;
const float RADTODEG = 57.29577f;
const int BYTES_PER_PIXEL = 2;
const float D = 5.0f;

int main(int argc, char **argv)
{
  int i, j, x, y, z, start, RANGE, scan_num, OUT_DIM, base, data_u, data_v,
      index, io = 0;
  float intensity, percent_done;
  char currentfile[256], scannumbuf[20], *filepath;
  FILE *in_file;
  // scan data matrix
  short int ***scan_data, *scan_mem;
  // result matrix
  float ***intensity_data, *intensity_mem;
  float s, t, d, u, v, w, beta;

  // Args: starting file path, start #, range, out dimension, I/O boolean
  if (argc == 6) {
    filepath = argv[1];
    start = atoi(argv[2]);
    RANGE = atoi(argv[3]);
    OUT_DIM = atoi(argv[4]);
    io = atoi(argv[5]);
  }
  else {
    printf("Argument error: Need path, start, range, and output resolution\n");
    exit(1);
  }

  // Set up scan data matrix
  scan_mem = (short int*)malloc(RANGE*AREA*sizeof(short int));
  scan_data = (short int***)malloc(RANGE*sizeof(short int**));

  for (i = 0; i < RANGE; i++) {
    scan_data[i] = (short int**)malloc(SCAN_DIM*sizeof(short int*));
  }

  // make memory contiguous
  for (i = 0; i < RANGE; i++) {
    for (j = 0; j < SCAN_DIM; j++) {
      scan_data[i][j] = &scan_mem[i*RANGE*SCAN_DIM+j*SCAN_DIM];
    }
  }

  // Set up intensity matrix
  intensity_mem = (float*)malloc(OUT_DIM*OUT_DIM*OUT_DIM*sizeof(float));
  intensity_data = (float***)malloc(OUT_DIM*sizeof(float**));

  for (i = 0; i < OUT_DIM; i++) {
    intensity_data[i] = (float**)malloc(OUT_DIM*sizeof(float*));
  }

  // make memory contiguous
  for (i = 0; i < OUT_DIM; i++) {
    for (j = 0; j < OUT_DIM; j++) {
      intensity_data[i][j] = &intensity_mem[i*OUT_DIM*OUT_DIM+j*OUT_DIM];
    }
  }

  // read in all scans from disk
  for (scan_num = start; scan_num < start+RANGE; scan_num++) {
    // create filenames to load
    strcpy(currentfile, filepath);
    strcat(currentfile, ".");
    sprintf(scannumbuf, "%d", scan_num);
    strcat(currentfile, scannumbuf);

    in_file = fopen(currentfile, "rb");
    if (!in_file) {
      perror("File Error");
    }

    base = scan_num - start;

       // load scans to memory
    fread(scan_data[base][0], BYTES_PER_PIXEL, AREA, in_file);
    fclose(in_file);
  }

  // Feldkamp algorithm
  for(x = 0; x < OUT_DIM; x++) {
    for (y = 0; y < OUT_DIM; y++) {
      for (z = 0; z < OUT_DIM; z++) {
             intensity = 0.0f;
             for (scan_num = start; scan_num < start+RANGE; scan_num++) {
               beta = scan_num*1.5f*DEGTORAD;
               s =  x * cosf(beta) + y * sinf(beta);
               t = -x * sinf(beta) + y * cosf(beta);
               d = D / (D - s);
               u = t * d;
               v = z * d;
               w = powf(d, 2);
               data_u = (int)u;
               data_v = (int)v;

               if ((data_u >= 0 && data_u < OUT_DIM) &&
                      (data_v >= 0 && data_v < OUT_DIM)) {
                    intensity += w*scan_data[scan_num][data_u][data_v];
               }
             }
             intensity_data[x][y][z] = intensity;
        }
    }
  }
  if (io) {
    for(x = 0; x < OUT_DIM; x++) {
      for (y = 0; y < OUT_DIM; y++) {
       for (z = 0; z < OUT_DIM; z++) {
         printf("%f\n", intensity_data[x][y][z]);
       }
      }
    }
  }

  // deallocate memory
  free(scan_mem);
  free(scan_data);
  free(intensity_mem);
  free(intensity_data);

  return 0;
}
```

Listing 3 – Complete C++ code listing for sequential (CPU) backprojection code integrated with visualization tool. This includes each `.h` and `.cpp` file (created by author)

```
//////////////////////////////////////////////////////////////////
//                                                              //
// Drew Maier                                                   //
// 4th Year Thesis Project                                      //
//                                                              //
// camera.h                                                     //
// Defines a camera class.                                      //
//                                                              //
//////////////////////////////////////////////////////////////////

#ifndef CAMERA_H
#define CAMERA_H

#include "geometry.h"
#include "timer.h"

class Camera {
      public:
            Camera();
            Vector pos;         // position vector
            Vector angle;       // angles (heading, pitch, yaw)
            Vector size;        // size
            Vector velocity;
            float speed;
};

#endif


//////////////////////////////////////////////////////////////////
//                                                              //
// Drew Maier                                                   //
// 4th Year Thesis Project                                      //
//                                                              //
// camera.cpp                                                   //
// Implements camera functions                                  //
//                                                              //
//////////////////////////////////////////////////////////////////

#include "camera.h"
#include <math.h>
#include <iostream>
using namespace std;

// default constructor
Camera::Camera() {
      angle = Vector(45.f, 45.0f, 0.0f);
      size = Vector(0.3f, 1.8f, .1f);
      pos = Vector(0.0f, 128.0f, 0.0f);
      speed = 100.0f;
}
```

```
/////////////////////////////////////////////////////////////////////
//                                                                 //
// Drew Maier                                                      //
// 4th Year Thesis Project                                         //
//                                                                 //
// geometry.h                                                      //
// Defines a geometric classes and constants.  This includes       //
// points and vectors, as well pi and angle conversion             //
//                                                                 //
/////////////////////////////////////////////////////////////////////

#ifndef GEOMETRY_H
#define GEOMETRY_H
#include <math.h>

const float PI = 3.141592f;
const float DEGTORAD = PI / 180.0f;

class Point2D {
	public:
		Point2D(): x(0), y(0) {}
		Point2D(int a, int b): x(a), y(b) {}
		int x, y;
};

class Point3D {
	public:
		Point3D(): x(0), y(0), z(0) {}
		Point3D(int a, int b, int c): x(a), y(b), z(c) {}
		int x, y, z;
};

class Vector {
	public:
		Vector(): x(0.0f), y(0.0f), z(0.0f) {}
		Vector(float a, float b, float c): x(a), y(b), z(c) {}
		Vector scalMult(float a) { return Vector(x*a, y*a, z*a); }
		float magnitude() { return sqrt(x*x+y*y+z*z)*1.f; }
		Vector normalize() { return Vector(x/magnitude(),
y/magnitude(), z/magnitude()); }
		float x, y, z;
};

#endif
```

```
//////////////////////////////////////////////////////////////////////
//                                                                  //
// Drew Maier                                                       //
// 4th Year Thesis Project                                          //
//                                                                  //
// timer.h                                                          //
// Defines a reconstruction object. This object represents a       //
// 3D CT reconstruction                                             //
//                                                                  //
//////////////////////////////////////////////////////////////////////

#ifndef Reconstruction_H
#define Reconstruction_H
#include <iostream>
#include <GL/freeglut.h>
#include "geometry.h"
using namespace std;

#define DIM 1024
#define OUT_DIM 512
#define MAX_INTENSITY 4096
#define RADTODEG 57.29577f;
#define BYTES_PER_PIXEL 2
#define D 128.0f
#define EPSILON 0.001f

class Reconstruction {
 public:
  Reconstruction(char *file);
  void draw(bool wire);
  Point2D pStart;
 private:
  float intensity[OUT_DIM][OUT_DIM][OUT_DIM];
  short int ***raw_data, *mem;
};

#endif
```

```
/////////////////////////////////////////////////////////////////
//                                                             //
// Drew Maier                                                  //
// 4th Year Thesis Project                                     //
//                                                             //
// reconstruction.cpp                                          //
// Implements a reconstruction object. This is where           //
// backprojection and display occur                            //
//                                                             //
/////////////////////////////////////////////////////////////////

#include "reconstruction.h"
#include "timer.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
using namespace std;

// constructor
Reconstruction::Reconstruction(char *filepath) {
  int x, y, z, start = 0, RANGE = 133, scan_num;
  FILE *in_file;
  float i;
  char currentfile[256];
  char scannumbuf[20];

  float s, t, u, v, w, beta;

  // Set up scan data matrix
  mem = (short int*)malloc(RANGE*DIM*DIM*sizeof(short int));
  raw_data = (short int***)malloc(RANGE*sizeof(short int**));

  for (x = 0; x < RANGE; x++) {
    raw_data[x] = (short int**)malloc(DIM*sizeof(short int*));
  }

  for (x = 0; x < RANGE; x++) {
    for (y = 0; y < DIM; y++) {
      raw_data[x][y] = &mem[x*DIM*DIM+y*DIM];
    }
  }

  // read in all scans from disk
  for (scan_num = start; scan_num < start+RANGE; scan_num++) {
    // fun C string business
    strcpy(currentfile, filepath);
    strcat(currentfile, ".");
    sprintf(scannumbuf, "%d", scan_num);
    strcat(currentfile, scannumbuf);

    in_file = fopen(currentfile, "rb");
    if (!in_file) {
      printf("File Error, can't find %s", currentfile);
      exit(1);
    }
    printf("Loading %s\n", currentfile);

    int base = scan_num - start;
    fread(raw_data[base][0], BYTES_PER_PIXEL, DIM*DIM, in_file);
    fclose(in_file);
  }

  Timer clk = Timer();
  clk.Start();

  int count = 0;
  int total_voxels = (int)powf(OUT_DIM, 3);
  int prev_percent = 0, percent_done = 0;

  // Feldkamp algorithm
  for(x = 0; x < OUT_DIM; x++) {
    for (y = 0; y < OUT_DIM; y++) {
      for (z = 0; z < OUT_DIM; z++, count++) {
      i = 0.0f;
      for (scan_num = start; scan_num < start+RANGE; scan_num++) {
        beta = scan_num*1.5f*DEGTORAD;
        s =  x * cos(beta) + y * sin(beta);
        t = -x * sin(beta) + y * cos(beta);
        u = (t * D) / (D - s);
        v = (z * D) / (D - s);
        w = pow(D / (D - s), 2);

        int data_u = (int)u;
        int data_v = (int)v;
        if ((data_u >= 0 && data_u < OUT_DIM) &&
            (data_v >= 0 && data_v < OUT_DIM)) {
          i += w*raw_data[scan_num][data_u][data_v];
        }
      }
      intensity[x][y][z] = i;

      // display progress
      prev_percent = percent_done;
      percent_done = (int)((count/(total_voxels*1.0f))*101);
      if (percent_done != prev_percent)
        printf("%3d%% complete\n", percent_done);
      }
    }
  }

  clk.Stop();
  printf("Elapsed time: %f\n", clk.Time());
}

// draw the reconstructed scan model
void Reconstruction::draw(bool wire) {
  for (int x = 0; x < OUT_DIM; x++) {
    for (int y = 0; y < OUT_DIM; y++) {
      for (int z = 0; z < OUT_DIM; z++) {
            float c = (1.0f*intensity[x][y][z]) / MAX_INTENSITY;
            if (c > EPSILON) {
              glColor3f(c, c, c);
              // points are much faster to draw
              if (wire) {
                // draw a point
                  glBegin(GL_POINTS);
                  glVertex3f(x, y, z);
                  glEnd();
              }
              else {
                // draw a cube
                  glBegin(GL_QUADS);


                  glVertex3f(x      , y         , z-1.0f);
                  glVertex3f(x-1.0f , y         , z-1.0f);
                  glVertex3f(x-1.0f , y         , z         );
                  glVertex3f(x      , y         , z         );

                  glVertex3f(x      , y-1.0f,   z           );
                  glVertex3f(x-1.0f , y-1.0f,   z           );
                  glVertex3f(x-1.0f , y-1.0f,   z-1.0f      );
                  glVertex3f(x      , y-1.0f,   z-1.0f      );

                  glVertex3f(x,           y,          z);
                  glVertex3f(x-1.0f,      y,          z);
                  glVertex3f(x-1.0f,      y-1.0f,     z);
                  glVertex3f(x,           y-1.0f,     z);

                  glVertex3f(x,           y-1.0f,     z-1.0f);
                  glVertex3f(x-1.0f,      y-1.0f,     z-1.0f);
                  glVertex3f(x-1.0f,      y,          z-1.0f);
                  glVertex3f(x,           y,          z-1.0f);

                  glVertex3f(x-1.0f,      y,          z           );
                  glVertex3f(x-1.0f,      y,          z-1.0f      );
                  glVertex3f(x-1.0f,      y-1.0f,     z-1.0f      );
                  glVertex3f(x-1.0f,      y-1.0f,     z           );

                  glVertex3f(x,     y,          z-1.0f      );
                  glVertex3f(x,     y,          z           );
                  glVertex3f(x,     y-1.0f,     z           );
                  glVertex3f(x,     y-1.0f,     z-1.0f      );

                  glEnd();
              }
            }
      }
    }
  }
}
```

```
////////////////////////////////////////////////////////////////////
//                                                                //
// Drew Maier                                                     //
// 4th Year Thesis Project                                        //
//                                                                //
// timer.h                                                        //
// Defines a timer object. This code was taken from CS445,        //
// Intro. to Computer Graphics with Greg Humphreys.               //
//                                                                //
////////////////////////////////////////////////////////////////////

#ifndef TIMER_H
#define TIMER_H

#if defined( _unix )
#include <sys/time.h>
#elif defined( _WIN32 )
#include <windows.h>
#endif

      class Timer {
      public:
            Timer();
            ~Timer();

            void Start();
            void Stop();
            void Reset();

            double Time();

      private:
            double time0, elapsed;
            bool running;
            double GetTime();

#if defined( __sgi )
            int fd;
            unsigned long long counter64;
            unsigned int counter32;
            unsigned int cycleval;

            typedef unsigned long long iotimer64_t;
            typedef unsigned int iotimer32_t;
            volatile iotimer64_t *iotimer_addr64;
            volatile iotimer32_t *iotimer_addr32;

            void *unmapLocation;
            int unmapSize;
#elif defined( _WIN32 )
            LARGE_INTEGER performance_counter, performance_frequency;
            double one_over_frequency;
#elif defined( _unix )
            struct timeval timeofday;
#endif
      };
#endif
```

```
//////////////////////////////////////////////////////////////////////
//                                                                  //
// Drew Maier                                                       //
// 4th Year Thesis Project                                          //
//                                                                  //
// timer_win.cpp                                                    //
// Implements a timer object. This code was taken from CS445,      //
// Intro. to Computer Graphics with Greg Humphreys.                 //
//                                                                  //
//////////////////////////////////////////////////////////////////////

#include <iostream>

using namespace std;

#if defined( __sgi )
#include <stddef.h>
#include <fcntl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/syssgi.h>
#include <sys/errno.h>
#include <unistd.h>
#elif defined( _WIN32 )
#include <windows.h>
#elif defined( _unix )
#include <sys/time.h>
#elif defined( powerc )
#include <ctime>
#else
#error TIMER ARCHITECTURE
#endif
#include "timer.h"

Timer::Timer()
{
#if defined( __sgi )
      __psunsigned_t phys_addr, raddr;
      int poffmask = getpagesize() - 1;
      int counterSize = syssgi(SGI_CYCLECNTR_SIZE);

      phys_addr = syssgi(SGI_QUERY_CYCLECNTR, &(cycleval));
      if (phys_addr == ENODEV) {
            cerr << "Sorry, this SGI doesn't support timers." << endl;
            exit(0);
      }

      raddr = phys_addr & ~poffmask;
      fd = open("/dev/mmem", O_RDONLY);

      if (counterSize == 64) {
            iotimer_addr64 =
                  (volatile iotimer64_t *)mmap(0, poffmask, PROT_READ,

                               MAP_PRIVATE, fd, (off_t)raddr);
            unmapLocation = (void *)iotimer_addr64;
            unmapSize = poffmask;
            iotimer_addr64 = (iotimer64_t
*)((__psunsigned_t)iotimer_addr64 +
                        (phys_addr & poffmask));
      }
      else if (counterSize == 32) {
            iotimer_addr32 = (volatile iotimer32_t *)mmap(0, poffmask,
PROT_READ,
                        MAP_PRIVATE, fd,
                        (off_t)raddr);
            unmapLocation = (void *)iotimer_addr32;
            unmapSize = poffmask;
            iotimer_addr32 = (iotimer32_t
*)((__psunsigned_t)iotimer_addr32 +
                        (phys_addr & poffmask));
      }
      else {
            cerr << "Fatal timer init error" << endl;
            exit(0);
      }
#elif defined( _WIN32 )
      QueryPerformanceFrequency( &performance_frequency );
      one_over_frequency = 1.0/((double)performance_frequency.QuadPart);
#endif
      time0 = elapsed = 0;
      running = 0;
}


double Timer::GetTime()
{
#if defined( __sgi )
      if (iotimer_addr64) {
            volatile iotimer64_t counter_value;
            counter_value = *(iotimer_addr64);
            return ((double) counter_value * .000000000001) * (double)
cycleval;
      }
      else {
            volatile iotimer32_t counter_value;
            counter_value = *(iotimer_addr32);
            return ((double) counter_value * .000000000001) * (double)
cycleval;
      }
#elif defined( _WIN32 )
      QueryPerformanceCounter( &performance_counter );
      return (double) performance_counter.QuadPart * one_over_frequency;
#elif defined( _unix )
      gettimeofday( &timeofday, NULL );
      return timeofday.tv_sec + timeofday.tv_usec / 1000000.0;
#elif defined( powerc )
      return (double) clock() / CLOCKS_PER_SEC;
#else
#error TIMER ARCH
#endif
}
```

```
Timer::~Timer()
{
#if defined( __sgi )
      close( fd );
#endif
}

void Timer::Start()
{
    running = 1;
    time0 = GetTime();
}

void Timer::Stop()
{
    running = 0;

    elapsed += GetTime() - time0;
}

void Timer::Reset()
{
    running = 0;
    elapsed = 0;
}

double Timer::Time()
{
      if (running) {
            Stop();
            Start();
      }
      return elapsed;
}
```

```
/////////////////////////////////////////////////////////////////
//                                                             //
// Drew Maier                                                  //
// 4th Year Thesis Project                                     //
//                                                             //
// main.cpp                                                    //
// Application entry point for visualization tool              //
//                                                             //
/////////////////////////////////////////////////////////////////

#include "reconstruction.h"
#include "timer.h"
#include "camera.h"
#include "timer.h"
#include <iostream>
#include <math.h>
#include <time.h>
using namespace std;

#define NEAR_PLANE 0.1f
#define FAR_PLANE 10000.0f
#define ASPECT_RATIO 1.6f
#define X_SENSITIVITY .25f
#define Y_SENSITIVITY .08f
#define WIDTH 640
#define HEIGHT 480

// flags
bool wPressed, sPressed, aPressed, dPressed, rPressed;
bool fullScreen = true, recticle = false, wire = false;
// keep frames synched
Timer frameTimer;
float lastTime;
// mouse positions
Point2D oldMouse;
Vector oldPos;
// instanciations of Reconstruction and Camera
Reconstruction *model;
Camera *cam;

void clearMatrices() {
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
}

void display( void )
{
  glutWarpPointer(WIDTH / 2, HEIGHT / 2);
  // clear out matrices and buffers
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  clearMatrices();
  gluPerspective(45.0, ASPECT_RATIO, NEAR_PLANE, FAR_PLANE);

  glRotatef(-cam->angle.y, 1.0f, 0.0f, 0.0f);
  glRotatef(-cam->angle.x, 0.0f, 1.0f, 0.0f);
  glTranslatef(-cam->pos.x, -cam->pos.y, -cam->pos.z);

  // draw the scene
  model->draw(wire);

  glColor3f(1.0f, 1.0f, 1.0f);

  // draw recticle (easier to aim camera)
  if (recticle) {
    glDisable(GL_LIGHTING);
    glDisable(GL_TEXTURE_2D);
    clearMatrices();
    gluOrtho2D(-4.0f, 4.0f, -3.0f, 3.0f);
    glLineWidth(2.0f);
    glColor3f(1.0f, 1.0f, 0.0f);
    glBegin(GL_LINES);
    glVertex2f(-0.1250f, 0.0f);
    glVertex2f(-0.0625f, 0.0f);
    glVertex2f( 0.0625f, 0.0f);
    glVertex2f( 0.1250f, 0.0f);
    glVertex2f(0.0f,-0.1250f);
    glVertex2f(0.0f,-0.0625f);
    glVertex2f(0.0f, 0.0625f);
    glVertex2f(0.0f, 0.1250f);
    glEnd();
  }
  glColor3f(1.0f, 1.0f, 1.0f);
  glutSwapBuffers();
}

void initgl(void)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glClearDepth(1.0);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_CULL_FACE);
  glFrontFace(GL_CW);
  glShadeModel(GL_SMOOTH);
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
  glutFullScreen();
  glutSetCursor(GLUT_CURSOR_NONE);
  // init flags
  wPressed = sPressed = aPressed = dPressed = rPressed = false;
  frameTimer.Start();
  cam->pos.x = model->pStart.x;
  cam->pos.z = model->pStart.y;

  glPolygonMode(GL_FRONT, GL_FILL);
}

void reshape( int w, int h )
{
  static int first = 1;
  if (first)
    {
      first = 0;
      initgl();
    }
  glViewport( 0, 0, w, h );
  clearMatrices();
  gluPerspective(45.0, ASPECT_RATIO, NEAR_PLANE, FAR_PLANE);
  glutPostRedisplay();
}

void keyboard( unsigned char key, int x, int y )
{
  // ESC to quit
  if (key == 27)
    {
      exit(1);
    }
  // w to walk forward
  if (key == 87 || key == 119)
    {
      wPressed = true;
    }
  // s to walk backward
  if (key == 83 || key == 115)
    {
      sPressed = true;
    }
  // a to walk left
  if (key == 65 || key == 97)
    {
      aPressed = true;
    }
  // d to walk right
  if (key == 68 || key == 100)
    {
      dPressed = true;
    }
  // r for recticle (aimer)
  if (key == 82 || key == 114) {
    recticle = !recticle;
  }
  // for full screen
  if (key == 70 || key == 102) {
    if (fullScreen) {
      glutReshapeWindow(WIDTH, HEIGHT);
      glutPositionWindow(0, 0);
    }
    else glutFullScreen();

    fullScreen = !fullScreen;
  }
  // q for wireframe
  if (key == 81 || key == 113)
    {
      wire = !wire;
    }
  glutPostRedisplay();
}
```

```
void keyup( unsigned char key, int x, int y )
{
  // w to walk forward
  if (key == 87 || key == 119)
    {
      wPressed = false;
    }
  // s to walk backward
  if (key == 83 || key == 115)
    {
      sPressed = false;
    }
  // a to walk left
  if (key == 65 || key == 97)
    {
      aPressed = false;
    }
  // d to walk right
  if (key == 68 || key == 100)
    {
      dPressed = false;
    }
  glutPostRedisplay();
}

// idle function
void idle() {
  float xMove, yMove, zMove;
  float timeElapsed = frameTimer.Time() - lastTime;
  float distance = timeElapsed * cam->speed;

  // adjust pitch and yaw
  xMove = sinf(cam->angle.x* DEGTORAD) * distance;
  yMove = sinf(cam->angle.y* DEGTORAD) * distance;
  zMove = cosf(cam->angle.x* DEGTORAD) * distance;

  float delta = 10.f;
  // move the camera based on input
  if (wPressed) {
    cam->velocity.x = xMove;
    cam->velocity.y = yMove;
    cam->velocity.z = zMove;
    cam->pos.x -= cam->velocity.x;
    cam->pos.y += cam->velocity.y;
    cam->pos.z -= cam->velocity.z;
  }
  if (sPressed) {
    cam->velocity.x = xMove;
    cam->velocity.y = yMove;
    cam->velocity.z = zMove;
    cam->pos.x += cam->velocity.x;
    cam->pos.y -= cam->velocity.y;
    cam->pos.z += cam->velocity.z;
  }
  if (aPressed) {
    cam->velocity.x = -zMove;
    cam->velocity.z = xMove;
    cam->pos.x += cam->velocity.x;
    cam->pos.z += cam->velocity.z;
  }
  if (dPressed) {
    cam->velocity.x = -zMove;
    cam->velocity.z = xMove;
    cam->pos.x -= cam->velocity.x;
    cam->pos.z -= cam->velocity.z;
  }
  // reset timers and repaint
  if (cam->angle.z >= 2*PI) cam->angle.z = 0.0f;
  if (frameTimer.Time() > 1.0) {
    frameTimer.Reset();
    frameTimer.Start();
  }
  lastTime = frameTimer.Time();
  oldPos.x = cam->pos.x;
  oldPos.z = cam->pos.z;
  glutPostRedisplay();
}

static bool doing_it = false;

void mouse( int button, int state, int x, int y )
{
  if (state == GLUT_UP)
    {
      doing_it = true;
    }
  else doing_it = false;
  glutPostRedisplay();
}

void look( int x, int y )
{
  int tempX = x - WIDTH/2;
  int tempY = y - HEIGHT/2;
  cam->angle.x -= tempX * X_SENSITIVITY;
  cam->angle.y -= tempY * Y_SENSITIVITY;
  glutPostRedisplay();
}

int main( int argc, char *argv[] )
{
  char *filename = "../../CTdata/3Dtest";

  if (argc == 2) {
    filename = argv[1];
  }

  glutInit( &argc, argv );
  glutInitWindowSize(WIDTH, HEIGHT);
  glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE );

  glutCreateWindow( "Reconstruction Viewer 1.0" );
  glutDisplayFunc( display );
  glutReshapeFunc( reshape );
  glutKeyboardFunc( keyboard );
  glutKeyboardUpFunc( keyup );
  glutMouseFunc( mouse );
  glutPassiveMotionFunc( look );
  glutIdleFunc( idle );

  cam = new Camera();
  model = new Reconstruction(filename);

  glutMainLoop();
  delete cam;
  delete model;
  return 0;
}
```

LISTING 4 - Complete code listing for parallel (GPU) backprojection code, including two `.cu` files (created by author)

```
//////////////////////////////////////////////////////////////////
//                                                              //
// Drew Maier                                                   //
// 4th Year Thesis Project                                       //
//                                                              //
// gpuct.cu                                                     //
// Application entry point for GPU backprojection program. This //
// loads the CT data from disk and calls the GPU processing     //
// function.                                                    //
//////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#ifdef _WIN32
#include <gpuct_kernel.cu>
#endif

const int SCAN_DIM = 1024;
const int AREA = 1048576;
const int BYTES_PER_PIXEL = 2;

// forward declaration
extern "C" void CUDA_backproject_scans(float *out_intensity, short int
       *scan_mem, const int &range, const int &out_dim, const int
       &intensity_mem_size, const int &scan_mem_size, const int
       &total_voxels);

int main(int argc, char **argv)
{
      int i, j, scan_num, start, range, out_dim, scan_mem_size,
          intensity_mem_size, total_voxels, io = 0;
      char *filepath;
      FILE *in_file;
      char currentfile[256];
      char scannumbuf[20];

      // scan data matrix
      short int* scan_mem;
      short int ***scan_data;

      // Args: starting file path, start #, RANGE
      if (argc == 6) {
            filepath = argv[1];
            start = atoi(argv[2]);
            range = atoi(argv[3]);
            out_dim = atoi(argv[4]);
            io = atoi(argv[5]);
      }
```

```
else {
      printf("Argument error: Need path, start, range, and out
              file resolution\n");
      exit(1);
}

// pre-compute memory usage
scan_mem_size = range*AREA*sizeof(short int);
total_voxels = (int)powf(out_dim, 3);
intensity_mem_size = total_voxels*sizeof(float);

// Set up scan data matrix
scan_mem = (short int*)malloc(scan_mem_size);
scan_data = (short int***)malloc(range*sizeof(short int**));

for (i = 0; i < range; i++) {
      scan_data[i] = (short int**)malloc(SCAN_DIM
                     *sizeof(short int*));
}

// make memory contiguous
for (i = 0; i < range; i++) {
      for (j = 0; j < SCAN_DIM; j++) {
            scan_data[i][j] = &scan_mem[SCAN_DIM*(i*range+j)];
      }
}

// read in all scans from disk
for (scan_num = start; scan_num < start + range; scan_num++) {
      // get file names to load
      strcpy(currentfile, filepath);
      strcat(currentfile, ".");
      sprintf(scannumbuf, "%d", scan_num);
      strcat(currentfile, scannumbuf);

      in_file = fopen(currentfile, "rb");
      if (!in_file)
            perror("File Error");

      int base = scan_num - start;

      fread(scan_data[base][0], BYTES_PER_PIXEL, AREA, in_file);
      fclose(in_file);
}

float *intensity = (float*)malloc(intensity_mem_size);

// GPU call
CUDA_backproject_scans(intensity, scan_mem, range, out_dim,
           intensity_mem_size, scan_mem_size, total_voxels);

if (io) {
      for(i = 0; i < powf(out_dim, 3); i++) {
            printf("%f\n", intensity[i]);
      }
}
```

```
      free(scan_mem);
      free(scan_data);
      free(intensity);

      return 0;
}



//////////////////////////////////////////////////////////////////
//                                                              //
// Drew Maier                                                   //
// 4th Year Thesis Project                                      //
//                                                              //
// gpuct_kernel.cu                                              //
// Definition of GPU program. This transfers scans to GPU and   //
// computes each voxel intensity.                               //
//                                                              //
//////////////////////////////////////////////////////////////////

#ifndef _GPUCT_KERNEL_H_
#define _GPUCT_KERNEL_H_
#include <cutil.h>
#include <stdio.h>

// primitive task function
__global__ void CUDA_func(short int* scan, float* intensity,
                          int *out_dim, int *scan_range) {

  __constant__ int SCAN_DIM = 1024;
  __constant__ float DEGTORAD = 0.01745f;
  __constant__ float D = 5.0f;
  __constant__ float ANGLE_INC = 1.5f;

  __shared__ int dim;
  __shared__ int range;
  __shared__ short int* scan_data;

  dim = *out_dim;
  range = *scan_range;
  scan_data = scan;

  // calculate the thread ID
  int tID = (blockIdx.x*blockDim.x*blockDim.y)
            + (threadIdx.x + threadIdx.y*blockDim.x);

  // translate the thread ID into x,y,z coordinates
  int x = blockIdx.y;
  int y = tID / dim;
  int z = tID % dim;

  float beta, d, s, t, u, v, w, vox_intensity = 0.0f;
  int scan_num, data_u, data_v, index;

  // Feldkamp algorithm
  for (scan_num = 0; scan_num < range; scan_num++) {
    beta = scan_num*ANGLE_INC*DEGTORAD;
    s =  x * cosf(beta) + y * sinf(beta);
    t = -x * sinf(beta) + y * cosf(beta);
    d = D / (D - s);
    u = t * d;
    v = z * d;
    w = pow(d, 2);
    data_u = (int)u;
    data_v = (int)v;

    if ((data_u >= 0 && data_u < dim) &&
      (data_v >= 0 && data_v < dim)) {
      // translate 3D coordinates to 1D array
      index = data_v + SCAN_DIM*(data_u + scan_num * range);
      vox_intensity += w*scan_data[index];
    }
  }
  int intensity_index = (blockIdx.y*gridDim.x*blockDim.x*blockDim.y)
                        + tID;
  intensity[intensity_index] = vox_intensity;
}

// sets up GPU for computation
extern "C" void CUDA_backproject_scans(float *out_intensity, short int
            *scan_mem, const int &range, const int &out_dim, const int
            &intensity_mem_size, const int &scan_mem_size, const int
            &total_voxels) {

  // allocate scan memory on GPU
  short int *scan_GPU;
  cudaMalloc((void**)&scan_GPU, scan_mem_size);
  // move scans onto GPU
  cudaMemcpy(scan_GPU, scan_mem, scan_mem_size, cudaMemcpyHostToDevice);

  // allocate memory for results from GPU
  float *intensity_GPU;
  cudaMalloc((void**)&intensity_GPU, intensity_mem_size);

  // allocate memory for dimension on GPU
  int *dim_GPU;
  cudaMalloc((void**)&dim_GPU, sizeof(int));
  // put output dimension on GPU
  cudaMemcpy(dim_GPU, &out_dim, sizeof(int), cudaMemcpyHostToDevice);

  // allocate memory for range on GPU
  int *range_GPU;
  cudaMalloc((void**)&range_GPU, sizeof(int));
  // put range on GPU
  cudaMemcpy(range_GPU, &range, sizeof(int), cudaMemcpyHostToDevice);

  // 16x16 is the sweet spot
  int BSIZE = 16;

  // set up thread blocks
  dim3 dimBlock(BSIZE, BSIZE);

  // create grid of blocks
  int gx = powf(out_dim/BSIZE,2);
  int gy = out_dim;
  dim3 dimGrid(gx, gy);

  // run the function
  CUDA_func<<<dimGrid, dimBlock>>>(scan_GPU, intensity_GPU, dim_GPU,
                                   range_GPU);

  // copy computed answer back to host
  cudaMemcpy(out_intensity, intensity_GPU, intensity_mem_size,
             cudaMemcpyDeviceToHost);

  cudaFree(scan_GPU);
  cudaFree(intensity_GPU);
  cudaFree(dim_GPU);
  cudaFree(range_GPU);
}

#endif
```

Appendix E – Project Update

Since completing this document over a month ago, I have continued research for GPUCT. Prof. Skadron has kindly allowed me to work on this project full-time this summer until I move in August. We hope to make significant progress with the program's output and hope to accomplish some of the goals I set out in the last chapter of this thesis report. Our ultimate goal is to publish a paper on using CUDA for CT reconstruction.

I recently met with a post-doctorate who has experience with the FDK algorithm. Seeing my output, he believes that I am implementing the algorithm incorrectly. He stated that filtering would only enhance the quality of otherwise correct output, but would not explain incorrect surfaces. Thus, I must experiment with my implementation of the FDK algorithm, making sure that I am manipulating the input data correctly. With more time to devote to this research, I should be able to fix this quickly.

Drew Maier

April 27, 2007