2009

Parallel cloth simulation using OpenMp and CUDA

Gillian David Sims
Louisiana State University and Agricultural and Mechanical College

PARALLEL CLOTH SIMULATION USING OPENMP AND CUDA

A Dissertation
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
In partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The School of Human Ecology

by
Gillian Sims
B.S., Louisiana State University, 1999
M.S., Louisiana State University, 2002
December, 2009

Acknowledgements

I would like to express my appreciation for everyone who has contributed to this work. I would like to thank Dr. Jonathan Chen for serving as my major professor for the vast majority of my Ph.D. career in Human Ecology. You allowed me to progress at my own, albeit glacial, pace; and for that I will always be grateful. I would like to thank Dr. Jenna Kuttruff for serving as my major professor for the final stages of this process. You guided me to completion despite myself. I would also like to thank Dr. Bonnie Belleau, Michael Book, Dr. Jeffrey Gillespie, and Dr. Carol O'Neil for serving on my committee. I hope to one day be as gracious with my time as each of you were with yours.

I could not have completed this process without my extended families at CJ Brown Realtors and Southern University. The hours of life experience gained at each institution during this process are as valuable to me as any course that appears on my program of study. I would also like to thank Elva Bourgeois, Yvonne Leak, Pamela Vinci, and Dr. Teresa Summers for being instrumental in my growth as a person. I would like to thank Monica Santaella for being my cellmate while I served my sentence. You escaped first, but in so doing, showed me that it was indeed possible. I would like to thank Georgette Keaton for being my anchor to humanity and dearest friend. Your importance to me cannot be overstated.

Finally I would like to thank my mother Eugenia McManus. Without you, I would not be possible. The extent to which this achievement brings you joy is the primary satisfaction that I derive from it.

Table of Contents

List of Tables

List of Figures

Abstract

The widespread availability of parallel computing architectures has lead to research regarding algorithms and techniques that best exploit available parallelism. In addition to the CPU parallelism available; the GPU has emerged as a parallel computational device. The goal of this study was to explore the combined use of CPU and GPU parallelism by developing a hybrid parallel CPU/GPU cloth simulation application. In order to evaluate the benefits of the hybrid approach, the application was first developed in sequential CPU form, followed by a parallel CPU form.

The application uses Backward Euler implicit time integration to solve the differential equations of motion associated with the physical system. The Conjugate Gradient (CG) algorithm is used to determine the solution vector for the system of equations formed by the Backward Euler approach. The matrix/vector, vector/vector, and vector/scalar operations required by CG are handled by calls to BLAS level 1 and level 2 functions. In the sequential CPU and parallel CPU versions, the Intel Math Kernel Library implementation of BLAS is used. In the hybrid parallel CPU/GPU version, the Nvidia CUDA based BLAS implementation (CUBLAS) is used. In the parallel CPU and hybrid implementations, OpenMP directives are used to parallelize the force application loop that traverses the list of forces acting on the system. Runtimes were collected for each version of the application while simulating cloth meshes with particle resolutions of 20x20, 40x40, and 60x60.

The performance of each version was compared at each mesh resolution. The level of performance degradation experienced when transitioning to the larger mesh sizes was also determined. The hybrid parallel CPU/GPU implementation yielded the highest frame rate for the 40x40 and 60x60 meshes. The parallel CPU implementation yielded the highest frame rate for

the 20x20 mesh. The performance of the hybrid parallel CPU/GPU implementation degraded the least as it transitioned to the two larger mesh sizes. The results of this study will potentially lead to further research regarding the use of GPUs to perform the matrix/vector operations associated with the CG algorithm under more complex cloth simulation scenarios.

Chapter 1 Introduction

Cloth simulations can be found in a variety of contexts today. From Hollywood blockbusters to garment prototyping software, the presence of virtual clothing and cloth objects unites a wide range of visual output. Researchers and practitioners have been working for over 20 years on algorithms and techniques to convincingly render cloth objects using computers (House & Breen, 2000). These techniques vary from simple geometric approximations to complex physical systems.

Whether the cloth simulation will operate as part of an interactive system or as part of a pre-rendered animation, dictates not only the sophistication of the simulation but the time constraints imposed on processing the simulation. Time constraints for pre-rendered animations, such as those that appear in Hollywood films, are far less stringent than those imposed upon interactive systems, such as video games. How long it takes to create a pre-rendered animation is irrelevant to the viewer of the animation. As a viewer of an animated sequence, one has no reference to the amount of time required to create the animated sequence. The time taken to create the sequence is only relevant to the producer of the animation to the extent that it affects the animation's deadline and the animation's total cost.

The interactive case is very different. The experience of the individual interacting with the system is dependent upon the speed of the animation. An interactive system would ideally produce animation at a rate of 30 FPS (frames per second). If the frame rate is too slow, animated objects move in a disjointed manner, lacking the smoothness of natural motion. Producing multiple frames of animation per second differs greatly from the pre-rendered world where minutes and even hours are acceptable times for the generation of a single frame. The desire to push simulation sophistication to higher levels while maintaining interactive frame rates

motivates research involving interactive systems. The performance requirements of such systems are extremely high and implementing such systems continues to be a major challenge facing cloth simulation researchers. The techniques used in this study are, however, more applicable to pre-rendered animation.

1.1 Cloth Simulation and Computing Hardware

Early cloth simulation research sought to establish methods for simulating cloth. Once those methods were established, subsequent work sought to improve the fidelity of the simulations or the speed in which the simulations execute. Both the overall fidelity of the simulation and the speed with which it executes are linked to the computing hardware on which the simulation is run. This relationship between model (the particular abstraction of cloth the simulation utilizes) and machine has always affected cloth simulation research. Recent research efforts have focused intently on this relationship in order to achieve improved performance by utilizing various parallel hardware architectures (Lario, García, Prieto, & Tirado, 2001) (Dencker, 2006) (Thomaszewski & Blochinger, 2006) (Thomaszewski, Pabst, & Blochinger, 2007) (Selle, Su, Irving, & Fedkiw, 2009).

Improvement in the sequential performance of computers has occurred throughout cloth simulation's history. These improvements benefited cloth simulations without necessitating a reformulation of the implementation algorithms or the underlying model. The simulation simply ran faster because the computer calculated faster. In essence, it came for "free." Once improvements to sequential performance began to stagnate, CPU manufacturers had to pursue parallelism in order to continue increasing the performance of hardware. It was at this point that improved simulation performance stopped being "free." At the very least, a reformulation of the implementation algorithm was required and possibly the underlying model.

Beyond the need to reformulate implementation algorithms and underlying models, the fundamental question of whether cloth simulations could be modified to exploit parallelism for improved performance had to be answered. Enough research has been produced to say that it can, but the boundary at which more parallelism ceases to yield greater performance is still uncertain. This is an important consideration because higher degrees of parallelism always increase cost, power consumption, and heat generation, but may not always improve performance. In addition to CPU parallelism, the GPU has emerged as a massively parallel architecture suitable for general purpose computation. Nvidia corporation developed CUDA (Compute Unified Device Architecture) in order to facilitate the use of Nvidia GPUs for general purpose computation. OpenMP is a framework for easily adding CPU parallelism to software. The major benefit of OpenMP is its ability to add parallelism to regions of code in an iterative fashion, rather than doing a full parallel code rewrite.

Cloth simulation can be divided into three stages. Stage one implements the cloth abstraction model that will be used. This stage is primarily concerned with the data types associated with the model and populating those data types with the initial conditions/values of the simulation. Stage two implements the numerical solver that will calculate the positions of the components of the simulation by solving the equations of motion that result from the application of forces within the simulation. Stage three implements the collision handling system, which is comprised of collision detection and collision response. Collision detection is the method by which the simulation determines whether two or more components of the simulation are attempting to occupy the same point in space. Collision response is how the simulation handles the detected collisions. Each of these stages represents an opportunity in which parallelism can be exploited.

1.2 Statement of Problem

In order to continue to improve the performance of cloth simulations, one must exploit the parallelism made available by hardware architectures. Performance gains in hardware going forward will be brought about by higher levels of parallelism. Ideally not only should cloth simulation algorithms be designed to exploit the parallelism available today, they should be able to scale to the parallelism of tomorrow. The extent to which cloth simulation algorithms are able to do this is a question researchers will continue to explore. This study focuses on the display of monolithic, as opposed to sub-divided, cloth meshes of increasing particle density. Increased particle density is valuable in its ability to render minute cloth details such as wrinkles and folds. Selle et al. (2009, p. 339) were able to achieve the impressive results depicted in Figure 1-1 by using high resolution meshes. The fundamental goal of cloth models of increased complexity is to more accurately simulate the behaviors of actual cloth. Many cloth simulations lack a fine

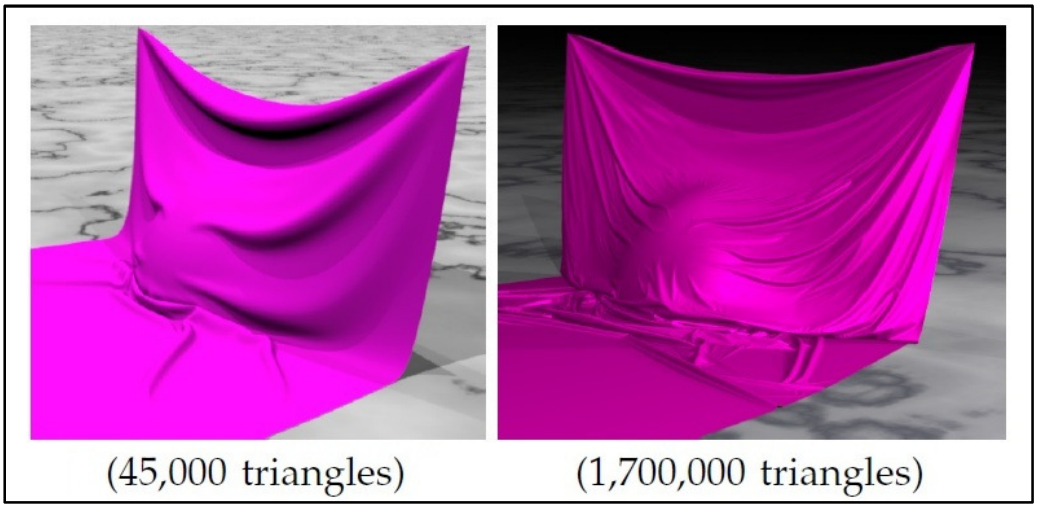

Figure 1-1. Wrinkles and folds in a high resolution mesh (Selle, Su, Irving, & Fedkiw, 2009)

grain representation of cloth suitable to represent deformations such as wrinkles and folds. This lack of granularity in regards to deformation stems from how sparse the underlying geometric abstraction of the cloth is. Sparse cloth models are implemented in order to maintain simulation

performance. If simulation performance can be improved, then increasingly fine geometric abstractions can be used in order to simulate fine cloth detail. More complex physical modeling, such as a sophisticated treatment of friction and drag, can also benefit from improved simulation performance.

1.3 Purpose

The purpose of this study is to determine whether a cloth simulation algorithm implemented using combined CPU and GPU parallelism will be able to offer higher animation frame rates than that same algorithm implemented using only one CPU or multiple CPU cores. This purpose will be achieved through the following objectives:

Objective 1: To develop a cloth simulation implementation that relies exclusively on sequential CPU computation in order to establish a base performance level on a given hardware platform, with a given simulation algorithm.

Objective 2: To modify the above cloth implementation so that CPU parallelism is exploited in the force calculation and numerical integration phases.

Objective 3: To modify the cloth simulation implementation produced in fulfillment of Objective 2 so that GPU parallelism is exploited in the numerical integration phase of the simulation. Force calculations will continue to be implemented using CPU parallelism.

Objective 4: To compare the frame rates produced by each of the above implementations on cloth meshes with the following particle densities: 20x20, 40x40, and 60x60. This objective seeks to determine the suitability of each implementation for a given cloth resolution.

1.4 Hypothesis

$H1_0$: There will be no frame rate difference between software implementations featuring no parallelism, CPU parallelism, or combined CPU/GPU parallelism.

$$FrameRate(Sequential) = FrameRate(Parallel\ CPU) = FrameRate(CPU/GPU)$$

$H1_A$: Software implementations featuring combined CPU/GPU parallelism will produce higher frame rates than software implementations featuring CPU parallelism alone. Software implementations featuring CPU parallelism will produce higher frame rates than software implementations featuring no parallelism.

$$FrameRate(CPU/GPU) > FrameRate(Parallel\ CPU) > FrameRate(Sequential)$$

$H2_0$: There will be no difference between the amount of frame rate decrease for software implementations featuring no parallelism, CPU parallelism, or combined CPU/GPU parallelism when applied to cloth meshes of the following densities: 20x20, 40x40, and 60x60.

$$FPS\text{-}Drop(Sequential) = FPS\text{-}Drop(Parallel\ CPU) = FPS\text{-}Drop(CPU/GPU)$$

$H2_A$: Software implementations featuring combined CPU/GPU parallelism will experience less frame rate decrease when applied to cloth meshes of increasing resolution than software implementations featuring CPU parallelism alone. Software implementations featuring CPU parallelism will experience less frame rate decrease when applied to cloth meshes of increasing resolution than software implementations that feature no parallelism.

$$FPS\text{-}Drop(CPU/GPU) < FPS\text{-}Drop(Parallel\ CPU) < FPS\text{-}Drop(Sequential)$$

1.5 Assumptions

For the purposes of this study, it is assumed that higher animation frame rates are a desirable application characteristic. Higher frame rates lead to smoother animation in interactive animations. Higher frame rates lead to faster job completion in pre-rendered animations.

1.6 Limitations

The implementations produced in this study simulate cloth in an application vacuum. No other application function exists apart from the rendering of the cloth mesh. It is understood that devotion of computational resources to the simulation of cloth diminishes the availability of those resources for other application functions. The extent to which it is desirable to allocate application resources to the simulation of cloth is a function of the level of importance assigned to cloth simulation within the application. The implementations produced in this study are also limited by the programming skill of the researcher. Since this limitation spans each implementation variant, it should not diminish the validity of the results observed.

Chapter 2 Review of Literature

For the purposes of this research, it is appropriate to differentiate between the parallelism gained by employing the use of clusters (multiple computers or computing nodes) and the parallelism that is available within a single workstation. This study focuses on the latter.

2.1 CPU Parallelism

Within a single workstation, there are two parallel computation architectures. There is parallelism at the CPU level and parallelism at the GPU level. CPU level parallelism can arise from the use of SMT (simultaneous multi-threading), multiprocessing, and/or CMP (chip-level multiprocessor). GPUs are inherently parallel, and in their current implementations contain tens to hundreds of stream processors/processing cores.

Simultaneous multithreading (SMT) is a method by which a single physical processor presents itself as multiple processors to the operating system. This is possible because a processor utilizing SMT is able to execute instructions from multiple threads of execution simultaneously (Curtis-Maury, 2008). This is accomplished by interleaving instructions from one thread of execution into the periods of latency found in another thread of execution. The net effect is that the processor is kept active by masking the latencies inherent in one thread's execution with instructions from another thread.

Multiprocessing is the use of multiple CPUs. One form of multiprocessing is Symmetric multiprocessing (SMP), which employs the use of two or more identical processors connected to a single shared memory (Hughes & Hughes, 2008). ccNUMA (cache-coherent non-uniform memory architecture) is another form of multiprocessing. ccNUMA differs from SMP in that the multiple CPUs have their own banks of memory that are simply shared with the other CPUs (Hughes & Hughes, 2008). At the workstation level, both SMP and ccNUMA are implemented

by having multiple CPUs reside on a single motherboard. Dual-socket and quad-socket motherboards can be found in the workstation space, allowing for either two or four CPUs to reside in a single system.

Chip-level multiprocessors (CMP) are single processor packages that contain two or more CPU cores. They can arise from multiple cores on a single die or multiple dies within a single package, sometimes both (Hughes & Hughes, 2008). It is the chip-level multiprocessor that has brought parallel computing to the mainstream. It is difficult to purchase a new PC that does not contain a CMP. While CMP benefits all users with improved multitasking performance among various simultaneously executing applications, a single application must be engineered with the ability to exploit the parallelism found in these CPUs directly.

2.2 Meaning of GPGPU

The graphics processing unit (GPU) is the hardware component responsible for sending images to the display device. This component evolved into a massively parallel processor as a function of the inherently parallel nature of graphics rendering, particularly 3D rendering. General Purpose computation on the GPU (GPGPU) is the utilization of the parallel processing power of the GPU for purposes other than graphics rendering. This is a particularly active area of research because the rate of GPU performance increase is outpacing that of CPUs. Figure 2-1 illustrates the increasing performance gap between the GPU and CPU (Harris, 2009, p. 43).

GPGPU research is also active because not all problems map well to the GPU. While a GPU is massively parallel, it is not nearly as versatile as a CPU. This limited versatility became accessible when the fixed-function graphics pipeline was enhanced to include programmable shaders. Programmable shaders allow the graphics developer to write custom operation sequences at specific stages of the graphics pipeline. If the instructions occur at the pipeline stage

concerned with vertex manipulation, it is known as a vertex shader or vertex program. If the instructions occur at the pixel manipulation stage, it is known as a pixel shader, fragment shader,

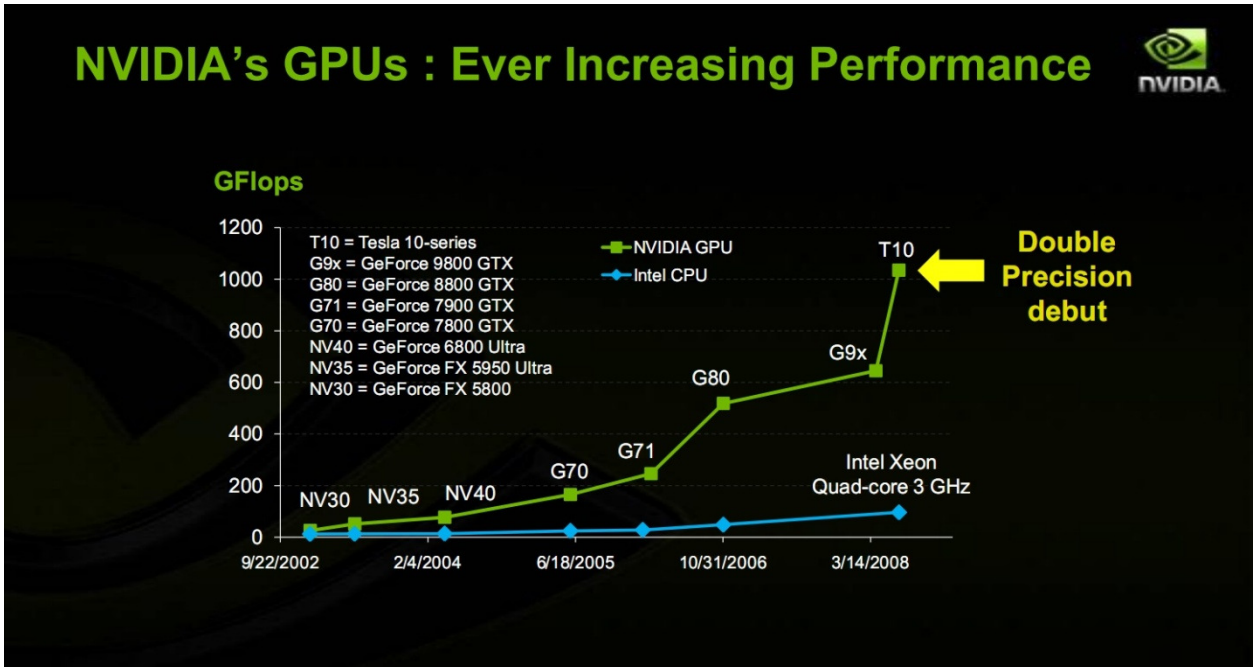

Figure 2-1. Nvidia GPU performance compared to CPU performance (Harris, 2009)

or fragment program. Application developers soon realized that shaders could be written with the purpose of performing general computation instead of graphics manipulation.

Historically, accessing the programmability of the GPU had to be done via a graphics API (OpenGL or DirectX) and a shading language (HLSL, Cg, GLSL). In the fall of 2006, Nvidia Corporation introduced the 8800GTX and changed the state of GPGPU. The 8800GTX supported CUDA, Nvidia's Compute Unified Device Architecture. CUDA enables access to the programmability of the GPU without using a graphics API or a shading language. Instead the programmer uses Nvidia's custom extensions to the C programming language and a dedicated compiler for the GPU code segments. This greatly reduces the learning curve necessary to implement software that seeks to exploit GPU parallelism, but it does not expand the types of

problems for which the GPU is suited. Because CUDA is a proprietary solution, it has not replaced GPGPU via custom shaders, it merely coexists with it.

2.3 General Cloth Simulation Research

Cloth simulation research has been underway for more than two decades. In that time, numerous works have been published. House and Breen present an excellent anthology of the work prior to 2000 in their book (House & Breen, 2000). This chapter will focus on the work that has been produced that concerns cloth simulation on parallel architectures. That work can be divided into three categories: works targeting distributed memory architectures, works targeting shared memory architectures, and works incorporating GPGPU. We will begin our discussion with distributed memory architecture research. But first, it would be appropriate to reference the work of Xavier Provot (Provot, 1995) and Baraff and Witkin (Baraff & Witkin, 1998).

In his work Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior, Provot (1995) described the use of a mass-spring model to animate cloth. This model is based upon the use of a set of point masses connect by springs that are without mass. Provot used 3 types of springs: shear, structural, and flexion. Figure 2-2 illustrates this structure (Provot, 1995, p. 148). Shear springs connect point masses that are diagonally adjacent to one another. Structural springs connect point masses that are horizontally and vertically adjacent. Flexion springs connect point masses in the horizontal and vertical direction with one point mass skipped over. Provot used the explicit integration technique known as Forward Euler to solve the differential equation of motion that the physical system produced. He noted the difficulty in using stiff springs because of its negative effect on application runtime. The lack of stiff springs created a “super-elastic” effect (Provot, 1995) that made the model unrealistic without subsequent processing.

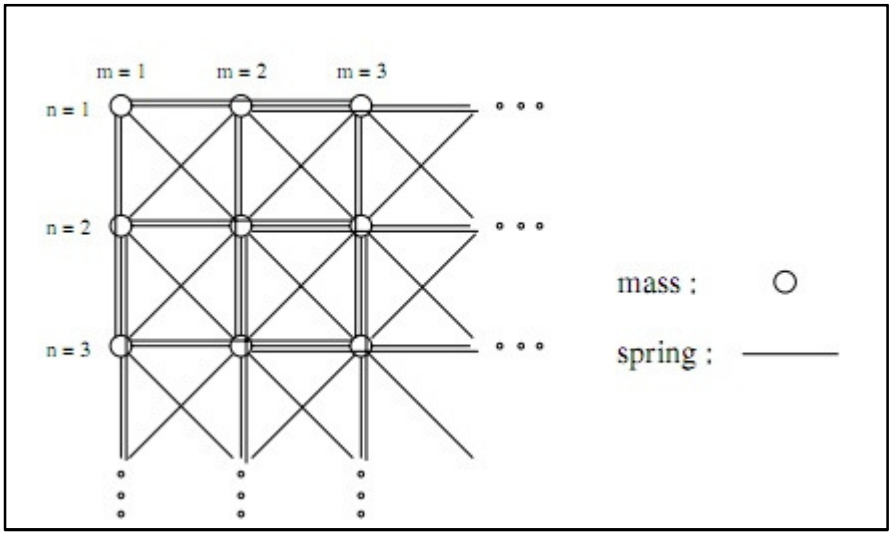

Figure 2-2. Regular mesh of masses and springs (Provot, 1995)

Large Steps in Cloth Simulation (Baraff & Witkin, 1998) is among the most cited works in cloth simulation research. In the pursuit of improved simulation performance, the authors tackled one of the principle bottlenecks in cloth simulation; numerically integrating the equations of motion that define the dynamic system. Prior to this work, explicit integration techniques such as Forward Euler and Runge-Kutta were the norm. However, explicit integration performance was limited by the small time steps required to maintain stability within the system. Baraff and Witkin proposed utilizing an implicit method, Backward Euler, instead. Backward Euler integration necessitates the solving of a system of equations at each time step. While this is more computationally demanding than the calculations necessary for forward Euler, it allows the researchers to take larger steps through time and maintain simulation stability. This computational tradeoff resulted in simulation times that pointed toward future interactive frame rates. It also ushered in a subset of cloth simulation research that focused intently on refining the numerical integration phase of the simulation. Today, implicit and semi-implicit techniques are the norm because of these prior works.

2.4 Distributed Memory Architecture Research

A distributed memory architecture is one whose computational units do not have direct access to each other's system memory. The most common distributed memory architecture utilized in cloth simulation research is that of the compute cluster comprised of commodity desktop PCs. Scientists researching computationally demanding disciplines utilize clusters to aide in their research. The price/performance ratio of clusters of commodity PCs eclipses that of the custom architected supercomputers of old. It was not long after clusters emerged as powerful computational tools that cloth simulation researchers turned their attention to these platforms.

Zara et al. were among the first to publish research on cluster based cloth simulation (Zara, Faure, & Vincent, 2002). They acknowledged the core difficulty of parallel cloth simulation; time driven simulations require sequential step by step computation. Therefore, parallelism must be exploited within a single step. They also acknowledged how important it is to attempt to develop an effective parallel cloth simulation framework. The researchers understood that PC clusters would continue to scale higher and if cloth simulation were to scale with them, appropriate parallel implementations would need to be developed. In this particular work, the focus was on the decomposition of the simulation model, which would allow the parallelization of calculations in each simulation step. Their path to decomposition was to subdivide the cloth being simulated. They realized that subdividing space would lead to an unbalanced distribution of work depending upon where in space the cloth resided.

Zara et al. (2002) implemented their simulation using Athapascan, an existing parallel programming environment. Their choice was motivated by the fact that Athapascan code could theoretically run on a wide range of parallel architectures, not just clusters. At the time of publishing this initial work, the version of Athapascan utilized didn't even support their cluster

consisting of 216 HP e-Vectra PCs, each housing a Pentium III CPU running at 733 MHz. Instead, initial results were presented utilizing a multiprocessor machine consisting of less than 10 CPUs. In 2004, the group was able to publish simulation times on their cluster, which now stood at 190 nodes (Zara, Faure, & Vincent, 2004). Their results indicated improvements in simulation times began to stagnate once 16 nodes had been reached and flattened off almost completely by 30 nodes.

Keckeisen and Blochinger (2004) published work focusing on parallelizing the integration phase of cloth simulations via the use of clusters. Their stated goals were to develop an efficient parallel system for irregular cloth meshes, to achieve good scalability on relevant problem sizes, and to utilize the cost effective platform of commodity PC clusters. Their work differed from Zara et al. (2002, 2004) in their choice of method of problem decomposition. While Zara et al. (2002, 2004) used the task-parallel Athapascan environment, Keckeisen and Blochinger used PETSc (Portable, Extensible Toolkit for Scientific Computation). PETSc provide data parallel primitives that utilize an underling message passing programming model, while Athapascan dynamically partitions tasks for the programmer based upon dependency graphs generated during runtime. In effect, Keckeisen and Blochinger (2004) were able to achieve a finer level of control over their parallel implementation.

Keckeisen and Blochinger's (2004) work also featured their TüTex cloth simulation engine, which was modified to become ParTüTex. Like Zara et al., Keckeisen and Blochinger divided the cloth into partitions that could be simulated in parallel on each node of their cluster. The significant implementation detail of their method pertained to how cloth vertices located at partition edges are handled. They referred to these vertices as ghost points because they physically belong to one processor (node), but logically belong to two processors (nodes). This

necessitated inter-node communication at each simulation step to handle these ghost points. These communications represented the major hurdle to being able to scale the level of parallelism higher. Put simply, there is an overhead associated with each additional partitioning of the cloth due to the need to synchronize shared vertices. Keckeisen and Blochinger performed their simulations on a 12 node cluster, with each node housing a 2.667 GHz Intel Xeon processor with 2GB of RAM. The nodes were interconnected by a Myrinet-2000 high-speed network. The 2000Mbs of bandwidth afforded by the Myrinet-2000 greatly exceeds the 100Mbs that connected nodes in Zara et al. That is a major factor in the near linear speedup that Keckeisen and Blochinger were able to observe up to their 12 node ceiling.

Thomaszewski and Blochinger (2006) continued the work of Keckeisen and Blochinger by adding parallelism to the collision handling phase of the cloth simulation. This work represented the first published attempt to deal with parallel cloth collision handling on PC clusters. The major challenge to parallel collision handling is the fact that unlike the modeling phase, collisions cannot be statically decomposed while maintaining balanced workloads across all compute nodes. Where collisions will occur cannot be known beforehand in interactive systems, therefore one cannot partition space or vertices at the outset of the simulation. As a result, Thomaszewski and Blochinger employed a dynamic problem decomposition procedure to the collision handling phase that differed from the static data decomposition of the modeling phase. They utilized BVHs (bounding volume hierarchy) and recursive collision tests distributed across the compute nodes in order to parallelize collision handling. The rationale of a BVH is that if areas of a cloth mesh are enclosed by a bounding volume, then one can first examine whether the bounding volumes have collided. If two bounding volumes do not intersect, then none of the vertices bound within them should intersect. If an intersection of bounding volumes

is detected, then progressively smaller bounding volumes can be tested until ultimately the vertices or faces are examined. To aide in the parallelization of this task, the researchers introduced a stack that contains untried BVH tests. By doing this, the nodes can access tests from other node's stacks as they become idle, which leads to load balancing.

Thomaszewski and Blochinger (2006) implemented their simulation using the DOTS (Distributed Object-Oriented Threads System) parallel platform. Their simulation is run on the same 12 node cluster described in Keckeisen and Blochinger's work. While the speedup of the simulation ceased to be linear beyond 5 nodes, it did continue to scale well up to their 12 node ceiling. Unfortunately Thomaszewski and Blochinger did not test identical scenes to those evaluated in Keckeisen and Blochinger, so one cannot determine the speedup achieved by the parallelization of the collision handling phase. Their goal was to communicate the extent to which additional parallelism sped up the simulation under consideration in this work alone.

Selle et al. examined the use of PC clusters for extremely high resolution cloth meshes (Selle, Su, Irving, & Fedkiw, 2009). While none of the afore mentioned works operate at interactive frame rates, their simulation times are considerably less than those obtained in this study. Forsaking speed, Selle et al. sought an unparalleled level of fine detail by simulating between 500,000 and 2,000,000 triangles in their scenes. In their most computationally demanding scene, it took an average of 45 minutes to render each frame. Like prior works, Selle et al. used decomposition of the cloth mesh to achieve parallelism in the modeling phase. However, they chose to integrate collision handling into the loops responsible for time integration. Another key difference of this work was the use of history-based self-repulsions. Self-repulsions are designed to prevent the cloth from penetrating itself by applying repulsive forces to cloth elements as they approach one another. The authors noted the difficulty in

determining the appropriate repulsive force when a simulation step is viewed in a vacuum. Therefore, they use the collision free state of a prior simulation step to inform the application of repulsions during the current step. Selle et al. ran their simulations on quad-processor AMD Opteron 2.8 GHz machines. They used either 2 or 4 machines, yielding simulations run on either 8 or 16 processors.

2.5 Shared Memory Architecture Research

Unlike distributed memory architectures, compute nodes in a shared memory system, in this case discrete processors or processing cores, share system memory with one another. This simplifies the programming of the simulation, but also restricts the extent to which it can scale. When working with clusters, one need only add more nodes to the cluster to increase available computing resources. In a shared memory system, there is an architectural ceiling to the amount of parallelism available. While this ceiling continues to rise in the broader hardware space, it cannot compete with the simplicity of adding nodes to a cluster. Shared memory systems do have the advantage of faster inter-processor communication. Rather than relying on network interconnects to transmit data, shared memory systems pass data across the system bus. While data traversal across the system bus is still a computational bottleneck, it is much less so than network communications.

Romero et al. were among the first to publish cloth simulation findings on a shared memory system (Romero, Romero, & Zapata, 2000). Their approach to problem decomposition was consistent with what has been described thus far, in that they divided the objects in the scene among the available processors. They too experienced the need to synchronize work among the processors, but this synchronization occurred across the system bus in the form of direct memory accesses and maintenance of cache coherency. They also distributed collision handling across

processors by distributing their AABB (Axis Aligned Bounding Box) tests to different processors and tasking one processor with maintaining the list of tests. The simulation was run on a SGI Origin 2000 computer that featured eight R10000 processors running at 250 MHz. They found that the speedup obtained by accessing additional processors was more linear as the complexity of the scene increased. At low levels of complexity, there isn't enough work for each processor to perform and communication overheads become a proportionally larger component of simulation runtime.

Lario et al. used OpenMP to code a shared memory cloth simulation system (Lario, García, Prieto, & Tirado, 2001). OpenMP (Open Multi-Processing) is an API designed to facilitate multiplatform shared memory programming. An energy based, multilevel technique to simulate the cloth was employed. The rationale of the multilevel technique is the awareness that global cloth motion can be calculated using a coarse mesh, with detailed motion coming with subsequent calculations on finer meshes. The point-by-point minimization techniques utilized by the researchers lend themselves to parallelization. However, there still exists a point at which increasing parallelism did not yield additional performance on cloth meshes of a particular size. This is referred to as the critical level in the work. Lario et al. leveraged the multiplatform attributes of OpenMP to run their simulations on three hardware platforms: a SGI Origin 2000 (16, 400 MHz MIPS R12000 processors), a Sun HPC 6500 (16, 400 MHz UtraSPARC-II processors), and a one IBM SP2 node (16, 375 MHz Power 3 processors). The performance difference between the three platforms was minor. The purpose of including the three platforms was to demonstrate the portability of the OpenMP code. A secondary goal was to compare the OpenMP implementation to a MPI (Message Passing Interface) implementation on each platform. The difference between OpenMP and MPI across the three platforms was also minor.

Therefore, the researchers contended that the added complexity of coding an MPI implementation was not justifiable for problems of the scale investigated when using shared memory architectures.

Gutiérrez et al. also used OpenMP, but focused on refining the memory access patterns inherent in parallel cloth simulations in order to improve performance (Gutiérrez, Romero, Romero, Plata, & Zapata, 2005). They compared a series of reduction techniques that sought to organize the cloth mesh data in such a way as to increase data locality. Data locality refers to how "local" a piece of data is to the processing element that needs it. If data are not local to the processing element, they must be fetched. This of course adds to runtime and consequently degrades performance. Gutiérrez et al. applied their techniques to the force calculation stage of their cloth simulator. Other stages are not addressed in this work.

As in their distributed memory work, Thomaszewski et al. elected to improve both the time integration and collision handling stages (Thomaszewski, Pabst, & Blochinger, 2007). Their work differs from the previously mentioned shared memory implementations in that they targeted commodity platforms instead of the highly parallelized systems mentioned earlier. They performed their simulation on a system featuring dual AMD Opteron 270 CPUs running at 2 GHz. Since each Opteron is a dual-core CMP, the simulation has access to a total of 4 processing cores. The researchers were able to accelerate the numerical integration phase by dividing the cloth mesh into regions handled in parallel. Figure 2-3 depicts a subdivided dress and identifies the three classes of vertices produced from such a subdivision (Thomaszewski, Pabst, & Blochinger, 2007, p. 72). The low CPU core count utilized was justified by acknowledging that the problem size the researchers focused on is much smaller than those implemented on clusters or massively parallel shared memory systems. They were able to achieve a near linear speedup

utilizing the 4 cores. They mentioned that future work should focus on seeing how the results scale on n-core systems.

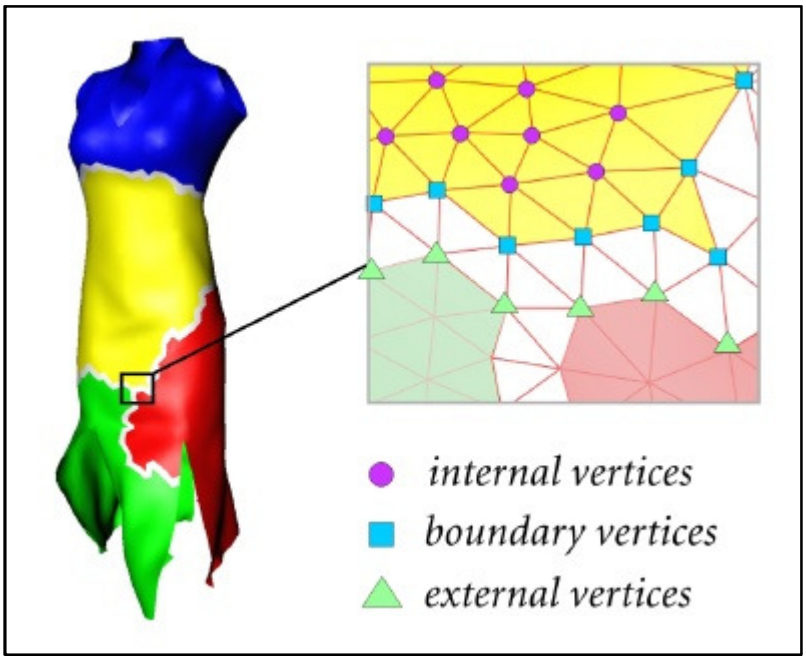

Figure 2-3. Exploiting parallelism through domain decomposition (Thomaszewski, Pabst, & Blochinger, 2007)

Bender and Bayer also conducted their simulation on a 4 core system (Bender & Bayer, 2008). However, their 4 cores came from a single quad-core Intel Q6600 at 2.4 GHz, rather than two dual-cores. Their work also used a different underlying physical model. Bender and Bayer proposed an impulse based system that can model inextensible cloth. By focusing on cloth that doesn't stretch much, the researchers are able to simplify the force calculations and speed up simulation time. The researchers divided the cloth mesh into strips of constraints that can be calculated in parallel due to their independence. In this respect, parallelism is a byproduct of the model instead of something the researchers introduce for improved hardware performance. The limitation is of course that the method is not suitable for cloth that needs to exhibit high levels of

stretch. The researchers mention that collision handling can be incorporated into the model, but do not outline the extent to which it exploits parallelism, if at all.

2.6 GPGPU Research

The final architecture category discussed is GPGPU. Before addressing works that focus on GPU cloth simulations, it is beneficial to discuss research efforts that sought the migration of computational kernels relevant to cloth simulation onto the GPU.

Bolz et al. implemented two important matrix solvers used in simulations on the GPU, conjugate gradient and multigrid (Bolz, Farmer, Grinspun, & Schröder, 2003). Baraff and Witkin's (1998) seminal work discussed earlier relied on the conjugate gradient method to solve the system of equations generated by implicit time integration. Nearly every subsequent implicit integration based cloth simulation has also utilized the conjugate gradient method. Therefore, being able to accelerate these calculations on the GPU is of interest to cloth simulation research. Bolz et al. found that on a Nvidia GeForce FX, they were able to nearly double the number of matrix multiplies per second achieved on a 3Ghz Pentium 4 CPU.

Zamith et al. also recognized the benefit of using the GPU to assist the CPU in performing mathematics and physics calculations (Zamith, Clua, Conci, Montenegro, Leal-Toledo, Pagliosa, Valente, & Feij, 2008). In their work, they proposed an architectural framework that distributes the computational load of virtual reality systems and games between the CPU and GPU. They noted that many mathematics and physics simulation problems can be structured as stream programs that can be executed on the GPU. They also recognized the fact that without proper load balancing, either the CPU or the GPU can be starved while the other is too busy. The researchers implemented their idea in an open source game engine. In this particular work, they focused on collision detection among moving solids. Their results showed

considerable improved performance during the collision detection phase by using the GPU to assist the CPU. The researchers' goal was to free up the CPU so that it can spend time processing tasks for which the GPU is not suited.

Birra and Próspero dos Santos implemented a subdivision algorithm for cloth simulation on the GPU (Birra & Próspero dos Santos, 2006). Recursive subdivision takes coarse mesh and subdivides it to form a smooth final mesh. The researchers sought a GPU implementation of this algorithm for two main reasons; one is to alleviate the CPU of these calculations and the other is to prevent the transfer of the fine mesh between the CPU and GPU. The fine mesh was a considerably larger data object than the coarse mesh, therefore transfers between the CPU and GPU of this information represented a bottleneck in the simulation. By allowing the GPU to create the fine mesh, that information need not leave the GPU since the GPU is also responsible for outputting the data to the display device.

Wang's (2005) M.S. thesis was similar to the work of Birra and Próspero dos Santos, with the addition of collision handling on the GPU. Wang utilized a coarse mesh that is simulated on the CPU and then passed to the GPU for refinement via subdivision. GPU collision handling came in the form of an image-space method. This technique involves moving the cloth vertex one depth layer to the object's surface position. Wang's work was greatly simplified by the fact that the only collisions were with the human avatar the cloth object rested upon. If a cloth vertex moved inside the volume representing the human avatar, it was simply moved out to the surface normal to its invalid position.

Another excellent M.S. thesis on GPU cloth simulation is by Kjartan Dencker (2006). Dencker compared the performance of a CPU implementation of an implicitly integrated cloth system with one implemented on the GPU. Dencker used the CG algorithm put forth by Baraff

and Witkin (1998) to iteratively solve his cloth system in both the CPU and GPU version of his program. This was all done using the Cg shading language and OpenGL. Dencker observed that the GPU version was slower at lower particle counts, less than 400, but was able to outperform the CPU at higher particle densities.

Dencker felt the 20 FPS (frames per second) he achieved with the GPU version of his program compared with only 2 FPS from the CPU version was the most impressive discrepancy. He noted that while not the ideal 30 FPS; at 20 FPS, interaction with the cloth object can occur. This was clearly an instance of the GPU being able to provide interactive frame rates with a problem size that was far too demanding for the CPU he tested. He ran his simulation on a laptop equipped with a 2.0GHz Pentium-M CPU and a Nvidia GeForce 6800 Go GPU with 256MB of graphics memory. However, as with all cloth simulations implemented fully on the GPU thus far, he is unable to detect cloth on cloth collisions.

Zeller put forth a technique for completely simulating cloth objects on the GPU (Zeller, 2007). Zeller's work is featured as part of the Nvidia Direct X 10 SDK as a demonstration of the GPU's capabilities. Zeller used Verlet integration to solve the equations of motion and relaxation passes to impose the spring constraints on the particles. Because the simulation was conducted on the GPU, the data are already available for display in video memory. This completely eliminates CPU/GPU communication bottlenecks that appear in the other works. The primary limitation of this method was that collision handling is not sophisticated. The cloth object is only capable of responding to collisions with a handful of geometric primitives and does not prevent self intersection.

2.7 Summary

The use of CPU parallelism to simulate cloth has been shown to improve performance by multiple research efforts. The use of GPU parallelism is the newer technique and has been shown to exceed the performance of sequential CPU implementations on larger mesh sizes by Dencker (2006). However, Dencker (2006), Wang (2005), and Zeller (2007) did not implement cloth-on-cloth collision detection in their GPU implementations. As a result, GPU cloth simulation is a technique suitable for a "cloth effect" that can be used as part of a broader scene, but not one in which the cloth is the central focus. Without cloth-on-cloth collisions, wrinkles and folds cannot exist when the cloth bunches on itself. It is also worth noting that all the GPGPU techniques discussed utilized graphics APIs and shader programming. Research has yet to be published that leverages Nvidia's CUDA programming model for the simulation of cloth.

This study is most directly motivated by the findings of Bolz et al. (2003), who showed that the GPU can execute matrix multiplications more quickly than the CPU. However, the work of Bolz et al. did not involve cloth simulation, therefore it is necessary to evaluate whether the performance gains observed in their study are applicable to the cloth simulation context. This study also seeks to demonstrate that the computational advantage of the GPU can be used in concert with the performance gains available from CPU parallelism. This hybrid parallel CPU/GPU approach can potentially be used to augment the performance of the sophisticated cloth models used in the work of Selle et al. (2009) and Thomaszewski et al. (2007).

Chapter 3 Methodology

This chapter describes the implementation details associated with this study. Section 3.1 addresses the research methods of the study. Section 3.2 addresses the mathematical structure of the cloth simulator. Vector quantities such as $\boldsymbol{v}$, appear in bold and are italicized. Matrix quantities such as $\mathbf{M}$, are simply bold. Scalar quantities such as Δt, are simply italicized. Section 3.3 addresses the formulation of the forces that act upon the cloth structure within the simulator. Section 3.4 describes the Conjugate Gradient method as implemented in this study. Finally, Section 3.5 describes the computational structure implemented in the study and how it was implemented across the three architectural platforms under consideration.

3.1 Methods

The primary focus of this research project is the development of a software implementation for the simulation of cloth that leverages both CPU and GPU parallelism. The goal of this hybrid parallel CPU/GPU implementation is to produce higher animation frame rates than those produced by sequential CPU or parallel CPU implementations. The implementation will be written in C++ and compiled on the Microsoft Windows XP 64bit operating system. The choice of C++ is rooted in its position as the preferred language for graphics programming and the fact that Nvidia's CUDA API is a series of extensions to the C programming language. C++ offers backward compatibility with C. The choice of a Microsoft operating system is rooted in the author's familiarity with Microsoft operating systems. The choice of Windows XP 64bit in particular is rooted in the overhead associated with the driver model employed by Windows Vista as it pertains to running CUDA enabled applications. Graphics rendering will be handled by the OpenGL graphics API. CPU parallelism will be implemented using the OpenMP parallel programming API. The choice of OpenGL and OpenMP will allow the author to port the

software implementations to non-Microsoft operating systems in the future. GPU parallelism will be handled by Nvidia's CUDA API. The choice of Nvidia's CUDA API is rooted in its ability to access the parallelism offered by the GPU without interfacing with a graphics API. There is also an absence of existing research utilizing CUDA for cloth simulations.

Results will be collected by running the software implementation variants on a custom built workstation featuring dual Intel Xeon 5520 CPUs running at 2.26Ghz on a Supermicro MBD-X8DAL-3-O motherboard. A photograph of the system appears in Figure 3-1.

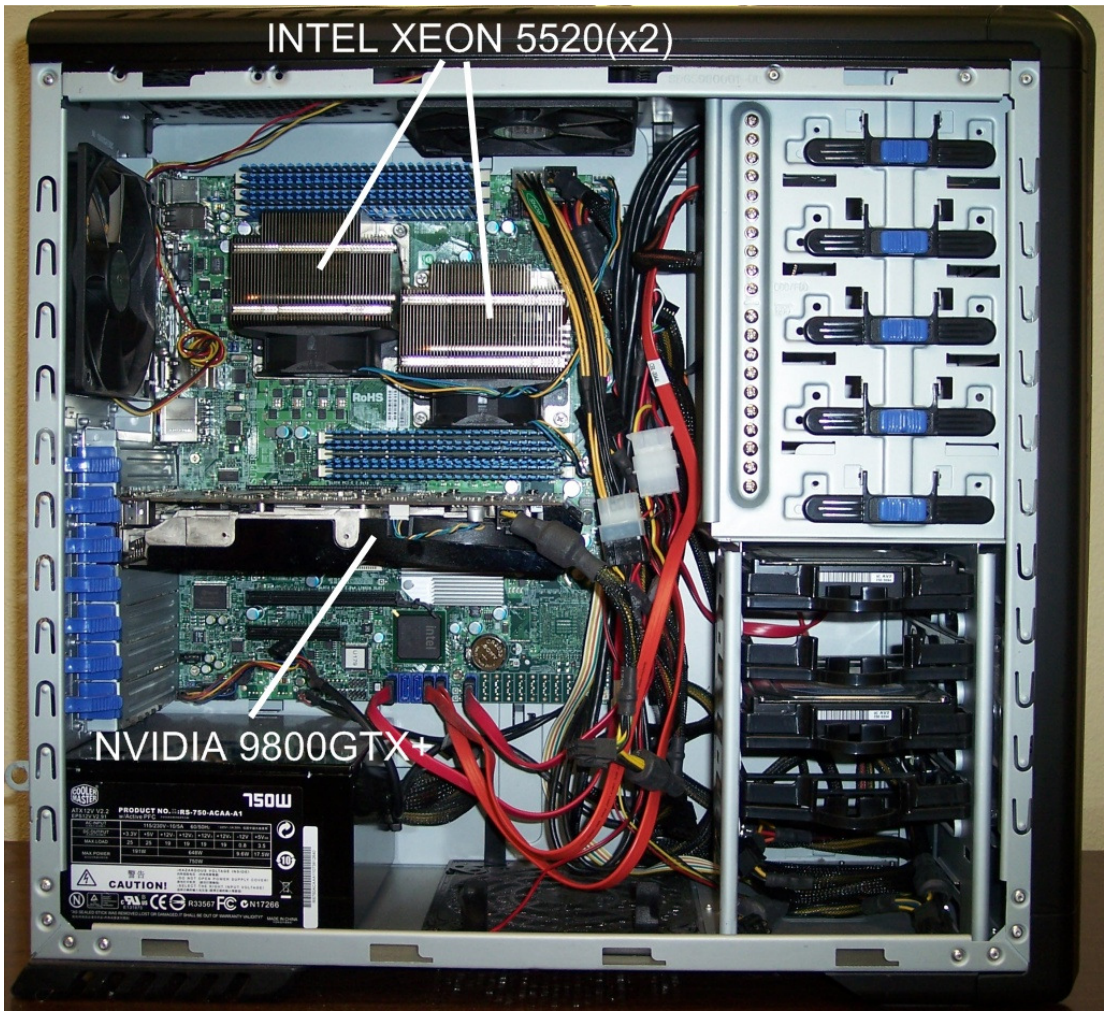

Figure 3-1. Test platform: Dual Intel Xeon 5520 CPUs and Nvidia 9800GTX+

This CPU platform was chosen because it features all three forms of CPU parallelism: CMP, SMT, and multiprocessing (ccNUMA). Each Intel Xeon 5520 is a quad-core CMP that supports Hyper-Threading (Intel Corporation's version of SMT). With Hyper-Threading enabled, each Xeon 5520 is capable of running 8 threads of execution simultaneously in hardware. The Supermicro MBD-X8DAL-3-O is a dual-socket motherboard featuring two LGA 1366 sockets. With two Intel Xeon 5520 CPUs installed, 16 threads of execution can be run simultaneously in hardware. For graphics rendering, the system features an Nvidia 9800 GTX+ graphics card. The GPU features 128 shader cores. The 9800 GTX+ features Nvidia's 4th highest shader count amongst single GPU graphics cards for desktop PCs. The choice of the 9800 GTX+ is rooted in its affordability relative to its parallel performance. This affordability will enable the researcher to experiment with multi-GPU configurations in future research at reduced cost. The 9800 GTX+ also fully supports Nvidia CUDA.

Figures 3-2, 3-3, and 3-4 depict the three observed mesh sizes. 20x20 was chosen as the lower bound mesh due to preliminary findings that showed that smaller mesh sizes are processed so quickly that meaningful data is difficult to obtain. 60x60 was chosen as the upper bound mesh due to memory allocation errors that occur at larger mesh sizes. While this study was completed using a 64bit OS, a 32-bit version of GLUT (the OpenGL Utility Toolkit) was used. This study uses general matrix storage and therefore rapidly approaches the limits of contiguous memory allocation in a 32bit application. Mesh sizes much larger than 60x60 would also exceed the 512MB of RAM available to the GPU. General matrix storage was chosen because of its compatibility with both the MKL BLAS implementation and the CUBLAS implementation. The use of differing storage formats for the two implementations would have introduced performance variability that would cloud the architectural comparison undertaken in this study.

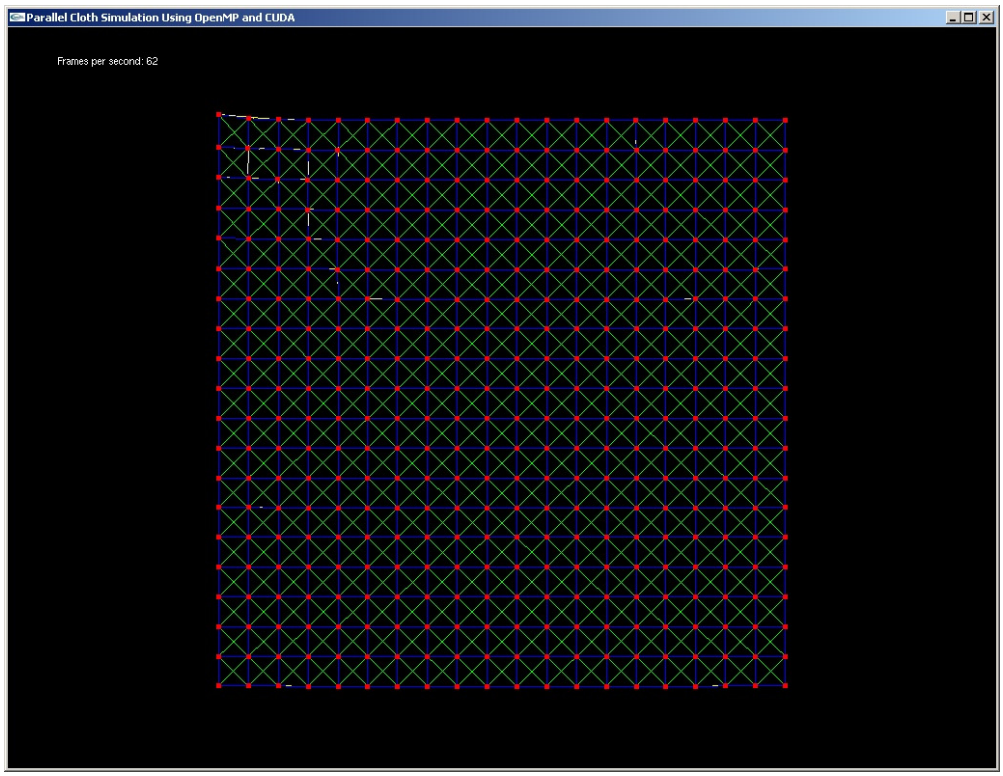

Figure 3-2. 20x20 mesh

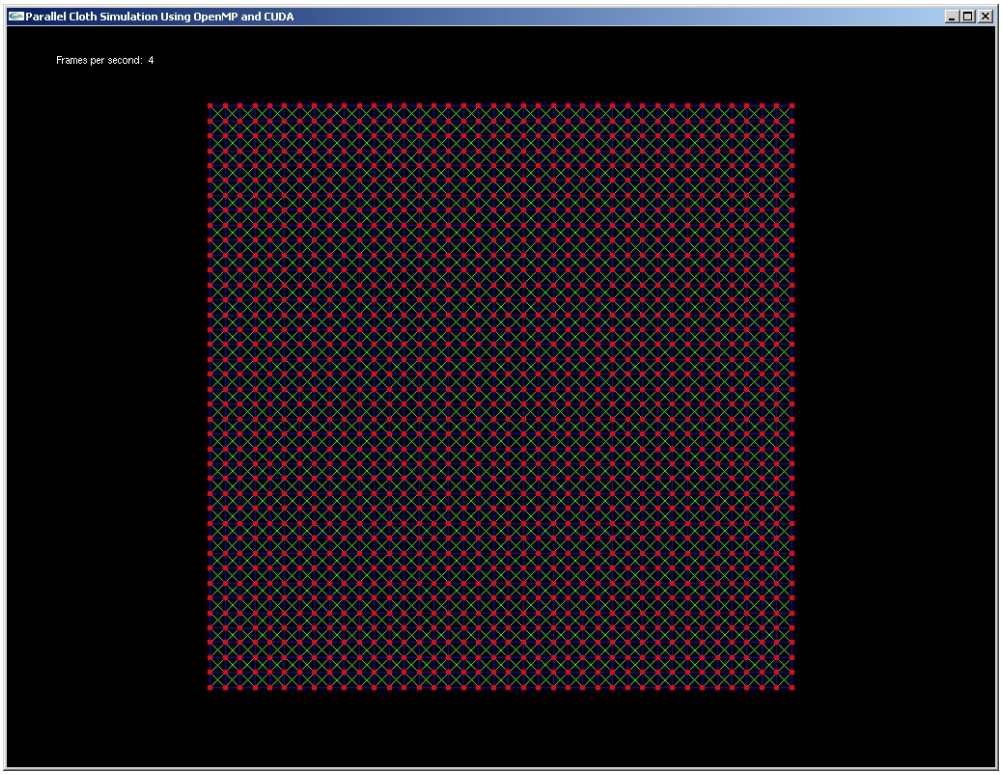

Figure 3-3. 40x40 mesh

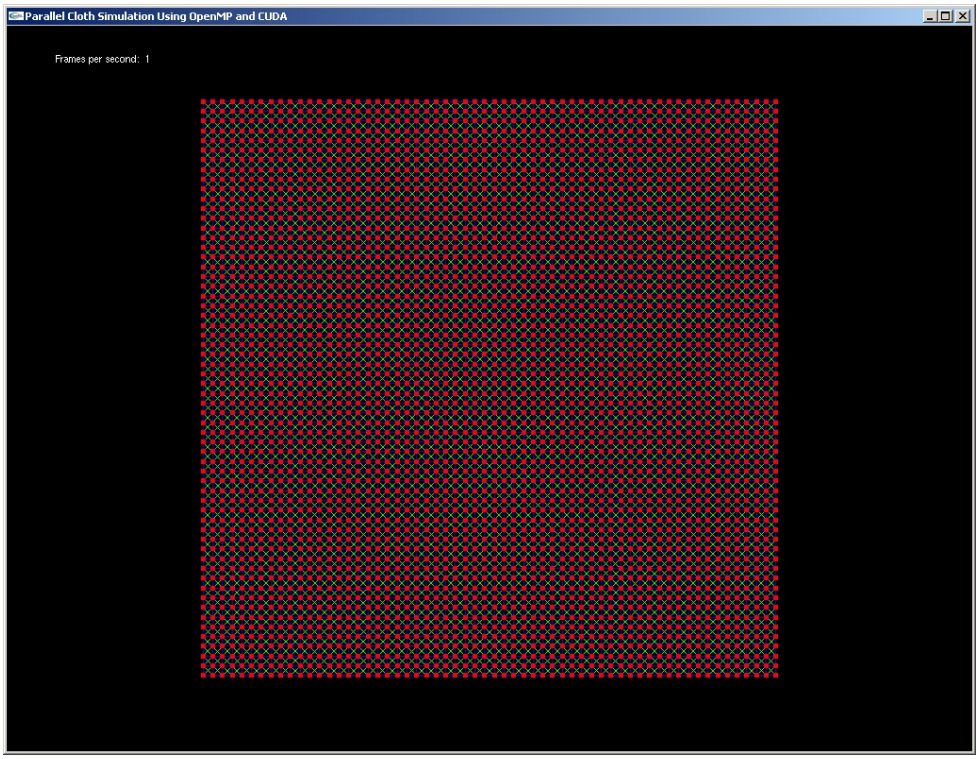

Figure 3-4. 60x60 mesh

3.2 Mathematical Structure

The mathematical structure of the simulation used in this study is derived from the work of Baraff and Witkin (1998) and the notes from their 1997 SIGGRAPH course on similar topics (Witkin & Baraff, 1997). This study implements a physically based simulation. Physically based simulations seek to arrive at animation states by using mathematical relationships between the simulation elements. These mathematical relationships are formulated based upon an attempt to describe a physical phenomenon, such as the effect of gravity on an object. The physical accuracy of the simulation is a function of how accurately the real world forces acting on the simulated object are modeled. The simulation context dictates whether absolute physical accuracy can be exchanged for perceived physical accuracy. An animation appearing in a video game or film context need not be entirely physically accurate; it need only appear physically accurate to the extent that it doesn't distract the viewer. This contrasts with a simulation

appearing in an engineering context, which would require as much physical accuracy as can be obtained with prevailing methods.

All physically based simulations are united by the requirement to describe the positions of the simulated objects as a function of time: $\boldsymbol{x}$(t). The positions objects occupy in simulated space as the simulation moves forward through time is the result of the forces acting on the simulated objects.

$$\boldsymbol{f}(\boldsymbol{x},\boldsymbol{v}) = \mathbf{M}\boldsymbol{a} \quad or \quad \boldsymbol{a} = \frac{\boldsymbol{f}(\boldsymbol{x},\boldsymbol{v})}{\mathbf{M}} = \mathbf{M}^{-1}\boldsymbol{f}(\boldsymbol{x},\boldsymbol{v}) \tag{1}$$

Equation (1) describes force as a function of position $\boldsymbol{x}$ and velocity $\boldsymbol{v}$. The force quantity is equal to the product of mass $\mathbf{M}$ and acceleration $\boldsymbol{a}$. In physically based simulations, the force quantity can be determined by the force rules that govern the simulation and the mass of the simulation objects are a parameter of the simulation. Acceleration is the unknown quantity, therefore it is logical to think of Equation (1) in the latter form.

The information necessary to calculate the acceleration of the simulated objects is contained in the simulation description, but recall that the physical simulation must describe object positions. The relationship between position and acceleration is described in Equation (2).

$$\ddot{\boldsymbol{x}} = \frac{d^2\boldsymbol{x}}{d^2t} = \boldsymbol{a} \tag{2}$$

Equation (2) states that acceleration is the second order time derivative of position. Equation (3) states that the first order time derivative of position is velocity, which leads to Equation (4) that states that the first order time derivative of velocity is acceleration.

$$\dot{\boldsymbol{x}} = \frac{d\boldsymbol{x}}{dt} = \boldsymbol{v} \tag{3}$$

$$\dot{\boldsymbol{v}} = \frac{d\boldsymbol{v}}{dt} = \boldsymbol{a} \tag{4}$$

The result of these relationships is the coupled differential equation expressed in Equation (5).

$$\frac{d}{dt}\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{v}\end{pmatrix} = \begin{pmatrix}\boldsymbol{v}\\ \mathbf{M}^{-1}\boldsymbol{f}(\boldsymbol{x},\boldsymbol{v})\end{pmatrix} \tag{5}$$

The values obtained for $\boldsymbol{x}$ and $\boldsymbol{v}$ by solving the coupled differential equations allows the simulation to march forward through time. The solution of this equation requires numerical integration for most formulations of $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{v})$ used in physically based simulations. In Section 2.3, the benefits of using implicit numerical integration techniques, specifically Backward Euler, were discussed while describing the work of Baraff and Witkin (1998). The Backward Euler method for the solution of Equation (5) is found in Equation (6).

$$\begin{pmatrix}\Delta\boldsymbol{x}\\ \Delta\boldsymbol{v}\end{pmatrix} = \Delta t\begin{pmatrix}\boldsymbol{v}_0 + \Delta\boldsymbol{v}\\ \mathbf{M}^{-1}f(\boldsymbol{x}_0 + \Delta\boldsymbol{x}, \boldsymbol{v}_0 + \Delta\boldsymbol{v})\end{pmatrix} \tag{6}$$

$$f(\boldsymbol{x}_0 + \Delta\boldsymbol{x}, \boldsymbol{v}_0 + \Delta\boldsymbol{v}) = \boldsymbol{f}_0 + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\Delta\boldsymbol{x} + \frac{\partial \mathbf{f}}{\partial \mathbf{v}}\Delta\boldsymbol{v} \tag{7}$$

$$\Delta\boldsymbol{v} = \Delta t\mathbf{M}^{-1}(\boldsymbol{f}_0 + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\Delta t(\boldsymbol{v}_0 + \Delta\boldsymbol{v}) + \frac{\partial \mathbf{f}}{\partial \mathbf{v}}\Delta\boldsymbol{v}) \tag{8}$$

The key attribute of Equation (6) is the evaluation of $\boldsymbol{f}$ at $(\boldsymbol{x}_0 + \Delta\boldsymbol{x})$ and $(\boldsymbol{v}_0 + \Delta\boldsymbol{v})$. These sums represent the position and velocity the simulation will achieve after it has taken a single step through time: Δt. It is important to realize that Equation (6) represents a system of equations because $\Delta\boldsymbol{x}$ and $\Delta\boldsymbol{v}$ are evaluated in terms of a function in which they are arguments. The goal of the simulation is to determine values of $\Delta\boldsymbol{x}$ and $\Delta\boldsymbol{v}$ that are correct when plugged into Equation (6). Equation (7) replaces the explicit evaluation of $\boldsymbol{f}$ at the new position and new velocity with its first order Taylor series expansion (Baraff & Witkin, 1998). This allows Equation (6) to be collapsed into the single linear system found in Equation (8) by simply substituting $\Delta\boldsymbol{x}$ with $\Delta t(\boldsymbol{v}_0 + \Delta\boldsymbol{v})$ where it appears in the lower half of Equation(6); note this substitution is derived from the upper half of Equation (6). The simulation now need only arrive at a value for Δv that holds true when plugged into Equation (8). From there $\Delta\boldsymbol{x}$ can be trivially computed by solving

the upper half of Equation (6). For reasons that will become evident in Section 3.4, it is more expedient to describe Equation (8) in the form that appears in Equation (9).

$$\left(I - \Delta t \mathbf{M^{-1}} \frac{\partial \mathbf{f}}{\partial \mathbf{v}} - \Delta t^2 \mathbf{M^{-1}} \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right) \Delta v = \Delta t \mathbf{M^{-1}} \left(\boldsymbol{f}_0 + \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \boldsymbol{v}_0\right) \tag{9}$$

Equation (9) is then multiplied on both sides by **M** to achieve Equation (10). Equation (10) is the calculation that ultimately governs the simulation's progression through time in this study.

$$\left(\mathbf{M} - \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{v}} - \Delta t^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right) \Delta v = \Delta t \left(\boldsymbol{f}_0 + \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \boldsymbol{v}_0\right) \tag{10}$$

3.3 Force Definitions

Equation (10) contains three force terms that must be evaluated at each time step the simulation takes: $\boldsymbol{f}_0$, $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, and $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$. There are only two forces that contribute to these terms in the cloth simulation model utilized in this study. The first force is a gravitational impulse and the second force is an internal spring forces that holds the cloth together. There are three types of springs that hold the cloth together, stretch springs, shear springs, and bend springs. Figure 3-5 is a screen capture of the simulation implemented in this study while displaying a 10x10 particle mesh. The stretch springs connect vertically and horizontally adjacent cloth particles and are rendered in blue. The shear springs connect cloth particles that are diagonally adjacent to one another and are rendered in green. The bend springs connect cloth particles to their vertical and horizontal neighbor, one particle removed, in each direction and are rendered in yellow. The 10x10 mesh is chosen for visual clarity, no data is collected at this resolution.

Each of these springs is modeled by a simple linear spring with damping as put forth by Macri (2008), who simplified the model used by Baraff and Witkin (1998). The contribution that

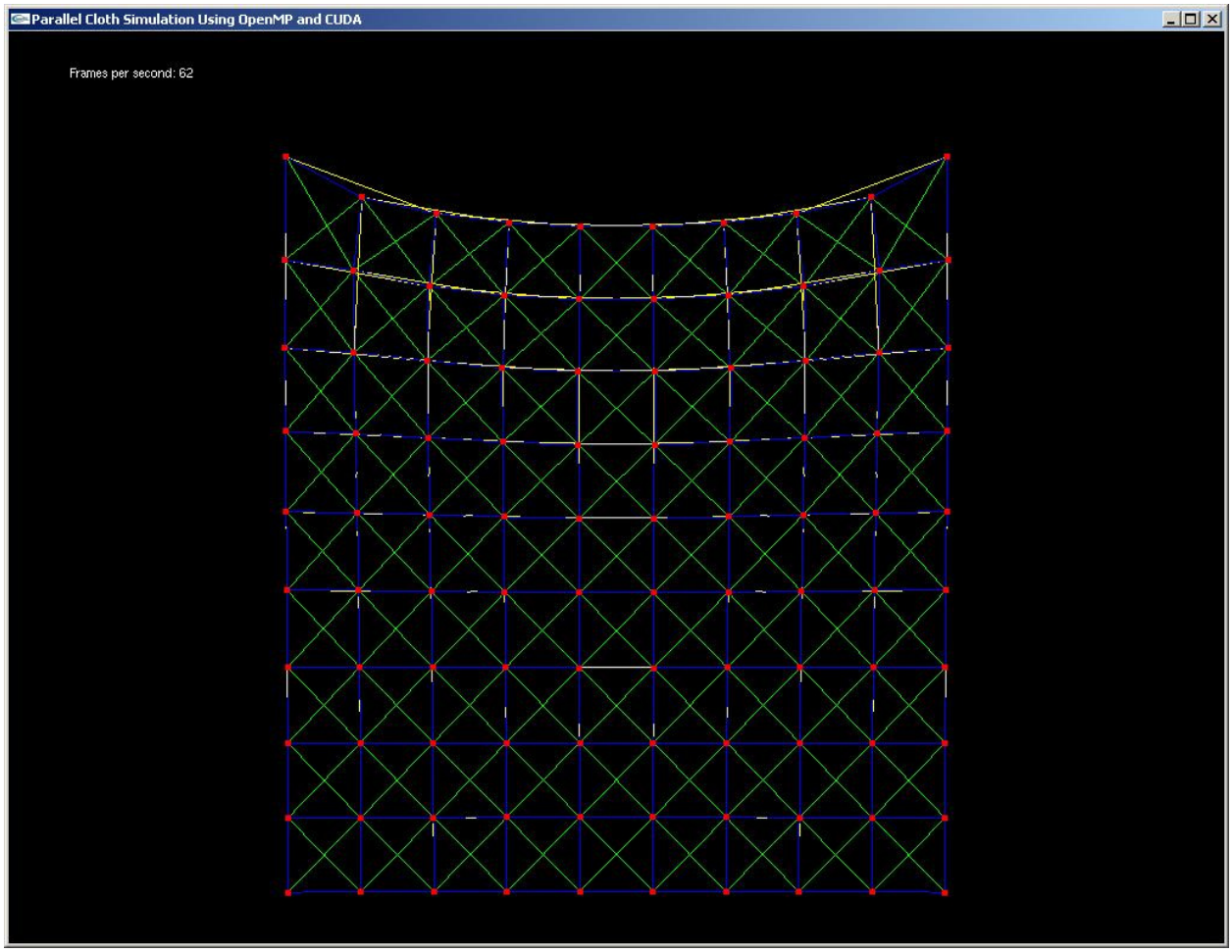

Figure 3-5. Stretch (blue), shear (green), and bend (yellow) springs of a 10x10 particle mesh

each spring makes to the system's forces can be modeled through the use of a condition function *C* that takes as its arguments the position vectors of the two particles joined by the spring. The model used in this study uses the same condition function for all three spring types, as was done by Macri (2008). This is far less physically accurate than Baraff and Witkin's use of distinct condition functions for each type of spring. The rational for this choice is governed by the primary purpose of this study; which is to compare the computational prowess of three hardware architectures, not to evaluate the underlying physical accuracy of the cloth simulation. Equation (11) describes the condition function *C* in terms of $\boldsymbol{L}$ (the difference between position vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$), where *r* is the rest length of the spring. Equation (12) describes $\boldsymbol{L}$ in terms of the two

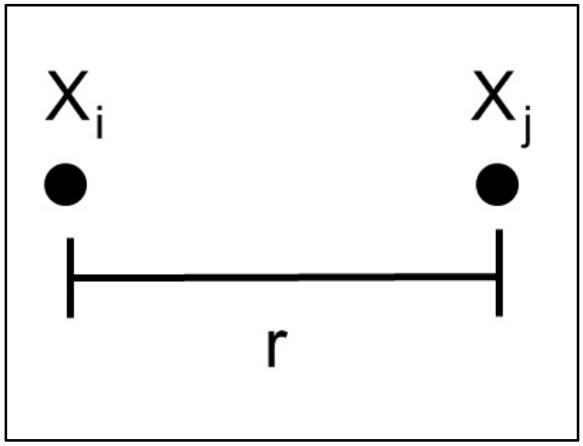

Figure 3-6. Particles X_i and X_j separated by rest distance r

position arguments of C. Figure 3-6 illustrates two particles in space, $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$, separated by a predefined distance r. When the distance $|\boldsymbol{L}|$ is equal to r, the condition function returns zero. As the particles move through space, the magnitude of C is dictated by the extent to which the distance $|\boldsymbol{L}|$ differs from r.

In addition to C itself, the first order partial derivatives with respect to each argument of C are defined in Equation (13). Equations (14) and (15) define the second order partial derivative of C with respect to each argument successively as defined by Macri (2008). Equation (16)

$$C(\boldsymbol{x_i}, \boldsymbol{x_j}) = |\boldsymbol{L}| - r \tag{11}$$

$$\boldsymbol{L}(\boldsymbol{x_i}, \boldsymbol{x_j}) = \boldsymbol{x_i} - \boldsymbol{x_j} \tag{12}$$

$$\frac{\partial C(\boldsymbol{x_i},\boldsymbol{x_j})}{\partial \boldsymbol{x_i}} = \frac{\boldsymbol{L}}{|\boldsymbol{L}|} \; and \; \frac{\partial C(\boldsymbol{x_i},\boldsymbol{x_j})}{\partial \boldsymbol{x_j}} = -\frac{\boldsymbol{L}}{|\boldsymbol{L}|} \tag{13}$$

defines the time derivative of C and it is also needed to calculate the force quantities. With C and each of its various derivatives defined; $\boldsymbol{f}_0$, $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, and $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ can now be defined

$$\frac{\partial^2 C(x_i,x_j)}{\partial x_i \partial x_j} = \begin{bmatrix} -\frac{(x_{ix}-x_{jx})^2}{|L|^3} + \frac{1}{|L|} & 0 & 0 \\ 0 & -\frac{(x_{iy}-x_{jy})^2}{|L|^3} + \frac{1}{|L|} & 0 \\ 0 & 0 & -\frac{(x_{iz}-x_{jz})^2}{|L|^3} + \frac{1}{|L|} \end{bmatrix} \tag{14}$$

$$\frac{\partial^2 C(x_i,x_j)}{\partial x_j \partial x_i} = \begin{bmatrix} \frac{(x_{ix}-x_{jx})^2}{|L|^3} - \frac{1}{|L|} & 0 & 0 \\ 0 & \frac{(x_{iy}-x_{jy})^2}{|L|^3} - \frac{1}{|L|} & 0 \\ 0 & 0 & \frac{(x_{iz}-x_{jz})^2}{|L|^3} - \frac{1}{|L|} \end{bmatrix} \tag{15}$$

$$\dot{C}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{\boldsymbol{L}\cdot(\boldsymbol{v}_i - \boldsymbol{v}_j)}{|\boldsymbol{L}|} \tag{16}$$

in Equations (17), (18), and (19) respectively; note that k_s and k_d refer to a spring coefficient and spring damping coefficient.

$$\boldsymbol{f}_i = -\left(k_s C(\boldsymbol{x}_i, \boldsymbol{x}_j) + k_d \dot{C}(\boldsymbol{x}_i, \boldsymbol{x}_j)\right) \frac{\partial C(x_i,x_j)}{\partial x_i} \tag{17}$$

$$\frac{\partial \mathbf{f_i}}{\partial \mathbf{x_j}} = -k_s \left(\frac{\partial C(x_i,x_j)}{\partial x_i} \frac{\partial C(x_i,x_j)}{\partial x_j}^T + \frac{\partial^2 C(x_i,x_j)}{\partial x_i \partial x_j} C(\boldsymbol{x}_i, \boldsymbol{x}_j) \right) - k_d \left(\frac{\partial^2 C(x_i,x_j)}{\partial x_i \partial x_j} \dot{C}(\boldsymbol{x}_i, \boldsymbol{x}_j) \right) \tag{18}$$

$$\frac{\partial \mathbf{f_i}}{\partial \mathbf{v_j}} = -k_d \left(\frac{\partial C(x_i,x_j)}{\partial x_i} \frac{\partial C(x_i,x_j)}{\partial x_j}^T \right) \tag{19}$$

3.4 The Conjugate Gradient Method

Recall that Equation (10) is the equation that defines the progression of the entire system through time. Equations (17)-(19) define each particle's contribution to the global force vector of the system and the two matrices that define the partial derivatives of this global force vector with respect to the global position and velocity vectors. The entire system is elegantly viewed as one monolithic collection of data stored in 3 matrices ($\mathbf{M}$, $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$) and 3 vectors($\boldsymbol{x}$, $\boldsymbol{v}$, $\boldsymbol{f}$). It is in fact the size of the vectors and particularly the size of the matrices that make cloth simulation

computationally demanding. The cloth mesh depicted in Figure 3-5 is a 10x10 collection of discrete mass points held together by the three types of springs outlined in Section 3.3. Each mass point, or particle, can be described by a position vector $\boldsymbol{x_i}$ in $\mathbf{R}^3$, <x,y,z>; with x,y,z corresponding to the values that locate the particle in relation to the three coordinate axis found in the 3D Cartesian coordinate system.

The global position vector $\boldsymbol{x}$ is a vector in $\mathbf{R}^{3n}$, where n is the number of particles in the mesh (n=100 for the mesh in Figure 3-5). This vector holds the positions of all the particles in the system as such: $<x_1,y_1,z_1,\ldots,x_n,y_n,z_n>$. Witkin and Baraff (1997) refer to this as block form, each particle contributing an $\mathbf{R}^3$ "block" to the global vector. The global velocity vector $\boldsymbol{v}$ and the global force vector $\boldsymbol{f}$ are also in $\mathbf{R}^{3n}$ and in block form. The three matrices are also in block form, but the blocks differ. The mass matrix $\mathbf{M}$ is in $\mathbf{R}^{3nx3n}$, which means that it is a square array of numbers with 3n rows and 3n columns (300x300 for the mesh in Figure 3-5). This matrix contains the mass of each particle repeated in blocks of 3 numbers along the main diagonal. Zero exists at every other position of the matrix. The matrices $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ and $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ are also in $\mathbf{R}^{3nx3n}$ but their blocks are in $\mathbf{R}^{3x3}$. The forces acting on each particle contribute an $\mathbf{R}^{3x3}$ block to the global derivative matrices. The blocks are indexed based upon the two particles connected by the spring. If a spring connects particles i and j, then the 3x3 block will begin at the position (3i,3j) in the global force derivative matrices.

The elegance of this abstraction is most apparent when one views Equation (10) simply as a linear system $\mathbf{A}\boldsymbol{x}=\boldsymbol{b}$. Where $\mathbf{A}$ is the known matrix $-\Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{v}} - \Delta t^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, $\boldsymbol{b}$ is the known vector $\Delta t \left(\boldsymbol{f}_0 + \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \boldsymbol{v}_0 \right)$, and one solves for the unknown vector $\boldsymbol{x} = \Delta v$ that when $\mathbf{A}$ is multiplied against, yields $\boldsymbol{b}$. This solution vector tells us the changes in velocity necessary to satisfy the Backward Euler integration of Equation (5). There are numerous techniques for solving such a

system. Baraff and Witkin's (1998) use of a modified Preconditioned Conjugate Gradient algorithm in their seminal work highlighted the efficacy of the method for the solution of the sparse symmetric system that the simulation of cloth produces in Equation (10). This study implements a conventional Conjugate Gradient (CG from this point forward) algorithm without the constraint filtering that allowed Baraff and Witkin to model dynamic cloth constraints. Preconditioning is also not utilized. While this affects the rate of convergence of the algorithm, this has no bearing on the relative performance of the architectures under review in this study because they each share this disadvantage.

Figure 3-7 (Shewchuk, 2004) outlines the CG algorithm. The algorithm iterates through a series of calculations that with each successive application, arrive at an approximate value of $\boldsymbol{x}$

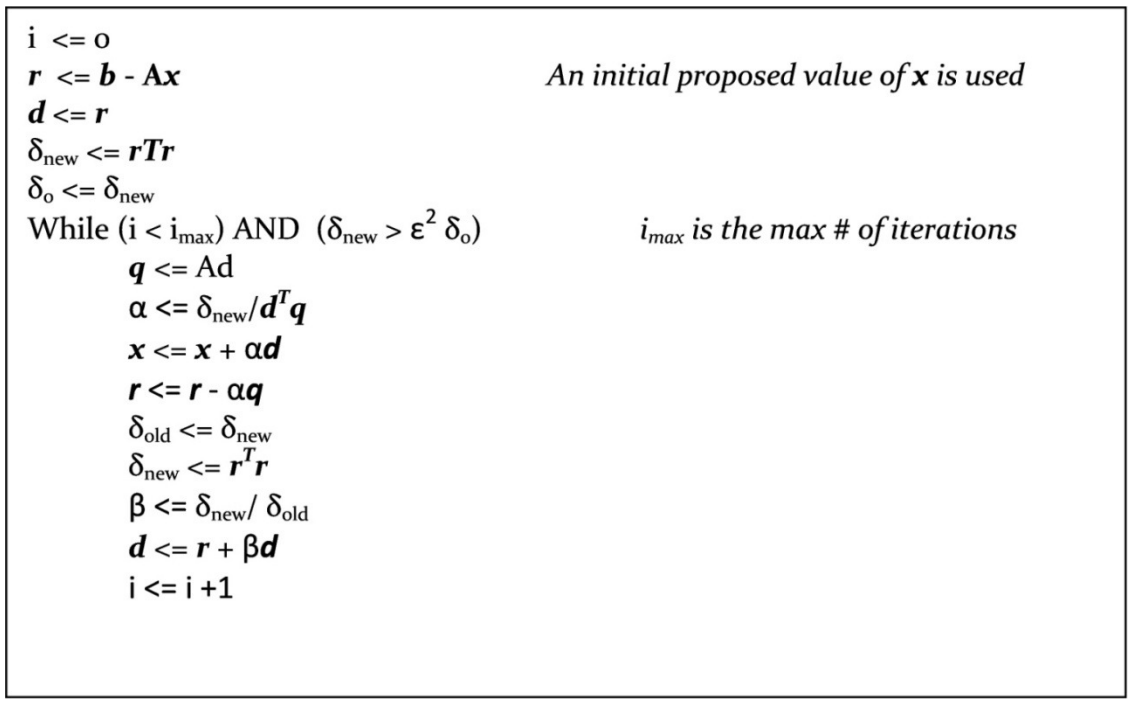

Figure 3-7. The Conjugate Gradient algorithm

that gets closer to the actual value of $\boldsymbol{x}$. The major advantage of this method, as it pertains to cloth simulation, is the ability to place a cap on the maximum number of iterations the algorithm will take. This has the potential to ensure that one stays within one's temporal budget for each

frame, with some measure of accuracy sacrificed. The CG algorithm is comprised of four operation classes: scalar division, vector addition, vector/scalar multiplication, and the computationally demanding vector/matrix multiplication. The vast majority of time spent arriving at a solution using the CG algorithm is spent performing the matrix/vector multiplication that occurs at the beginning of each passage through the While loop. Depending on whether a non-zero initial approximation of x is supplied, one or two matrix/vector multiplications occur prior to entering the CG loop. A zero vector (a vector containing only zeros) is used in this study as the initial approximation of x, so only one pre-loop matrix/vector multiplication occurs. The consequence of using a zero vector is that the method starts off blindly rather than with an educated guess. This ultimately increases the number of iterations required to converge but this limitation occurs across all hardware platforms.

3.5 Computational Structure

The key insight that forms the basis of the computational structure of this study is rooted in the fixed calculation classes utilized by the CG algorithm. Recognizing that the vector/matrix multiply is the most computationally demanding aspect of the algorithm, this study gets distilled into the simple task of finding out which parallel architecture speeds up vector/matrix multiplication the most. The researcher chose to evaluate this by relying on BLAS (Basic Linear Algebra Subprograms) calls to perform all operations on the large 3n vectors and 3nx3n matrices that are at the core of the monolithic view of the particle system being simulated. BLAS is a collection of subprograms that perform operations at three levels (BLAS - Basic Linear Algebra Subprograms, 2009). Level 1 BLAS deals with vector operations; level 2 BLAS deals with matrix/vector operations; and level 3 BLAS deals with matrix/matrix operations. The software

developed as part of this study only makes calls to BLAS levels 1 and 2. The only matrix/matrix computations are addition and subtraction, which is not computationally demanding.

3.5.1 Sequential CPU Implementation

The sequential CPU implementation of the software is the foundation from which the parallel CPU and hybrid parallel CPU/GPU implementations are built. The sequential CPU version relies on the optimized BLAS implementation developed by Intel contained in the Intel Math Kernel Library (MKL). This study is implemented using version 10.2.1. MKL's BLAS implementation is optimized to run on Intel CPUs. It is also capable of taking advantage of available parallelism within the system if allowed to do so by the programmer. This was the primary reason that it was chosen. Chapter 5 will address the impact of that choice. The software developed simply renders a cloth mesh suspended by its upper left mass point/particle and allows the cloth mesh to fall under the influence of a gravitational impulse for 250 frames.

The choice of 250 frames was arbitrary. Simulations were allowed to run longer while the software was being developed, but the data collection phase required a consistent simulation routine. The choice of 250 frames kept the total run time for the dense 60x60 particle mesh down. The simulation speed for a given mesh size/architecture combination did not change appreciably as the simulation progressed; so the point at which data collection stopped was arbitrary. For this reason, the arithmetic mean of the processing time for each frame over the course of the 250 frame run is a suitable measure of the animation rate. If the potential existed for complex interactions with other objects or forces as the animation progressed, this would not be a suitable measure of performance. Data is collected for a single simulation run. The researcher observed little to no variation from run to run. The final simulation runs used as data

for this study were collected within hours of one other in order to maintain a consistent performance environment on the workstation.

Suspension of the mass point is achieved by assigning the mass point a very large mass. This results in a value for acceleration that is close to zero when forces are applied. Recall that $\boldsymbol{a} = \boldsymbol{f}/\mathrm{m}$, therefore having a large value for mass produces a very small value for acceleration. Macri (2008) was able to achieve an approximation for infinite mass by storing the inverse mass of his mass points/particles and calculating: $\boldsymbol{a} = \boldsymbol{f} * 1/\mathrm{m}$. When the inverse mass is stored, setting the inverse mass equal to zero has the net effect of completely canceling the forces applied to that mass point with the understanding that such a quantity could not actually exist. For the purposes of this study, a very large mass was sufficient to suspend the particle within a stably running system.

GLUT was used to create the empty OpenGL window that the simulation exists in (Jacobs, 2007). The code that establishes an initial position for the particles in $\mathbf{R}^3$ and displays the particles and springs was derived from Paul Baker's cloth simulation example (Baker, 2003). The code used to calculate the forces and force derivatives for linear springs was derived from Dean Macri's cloth simulation example (Macri , 2008). The CG algorithm used was derived from Daoqi Yang's text book on numeric computing (Yang, 2001). A frame rate counter written by Toby Howard was modified for use in this study (Howard, 2006). Witkin and Baraff's SIGGRAPH course notes on physically based animation provided insights into the data structures necessary to simulate physically based animation systems (Witkin & Baraff, 1997). The software implementations were written using Microsoft Visual Studio 2005 with code compiled using Intel's C++ compiler version 11.1.038.

3.5.2 Parallel CPU Implementation

The parallel CPU implementation is identical to the sequential CPU version except that it adds OpenMP `#pragma` statements at strategic points within the code; mostly in front of long `For` loops. OpenMP `#pragma` statements send instructions to the compiler to parallelize the block of code immediately following the `#pragma` statement. The number of OpenMP threads was set to 16 using the `omp_set_num_threads()` function. The 16 threads correspond to the 16 logical cores that the Windows XP-64 Professional OS perceives while running the dual Xeon 5520 test platform with Hyper-threading enabled. The benefit of using OpenMP is that it is portable to other operating systems and more importantly, allows one to parallelize the code in sections. More sophisticated parallelization techniques often require a complete rewrite of the sequential code. The reward for doing this is usually improved performance over the simple OpenMP `#pragma` statements used in this study.

3.5.3 Hybrid Parallel CPU/GPU Implementation

The hybrid parallel CPU/GPU implementation represents the core contribution of this study. Research in cloth simulation has been devoid of references to the coupling of CPU parallelism with GPU parallelism. The parallel GPU architecture is accessed in this study through the use of Nvidia's CUDA framework. CUDA allows the programmer to access the compute capability of the GPU using native C, rather than a graphics API such a OpenGL or DirectX. What is tremendously important for this study, is the fact that the CUDA framework contains a GPU based implementation of BLAS called CUBLAS. The parallel CPU/GPU version of the software is identical to the parallel CPU version except that it replaces calls to the MKL BLAS library with calls to the CUBLAS library for operations performed on the vectors

and matrices associated with the CG algorithm. It also contains additional data structures necessary for managing the data on the GPU. All OpenMP `#pragma` statements are kept.

Chapter 4 Results and Analysis

This chapter presents the results of the simulation runs on the various architecture platforms on the three mesh sizes under consideration, along with an analysis of the data. Section 4.1 presents the sequential CPU results. Section 4.2 presents the parallel CPU results. Section 4.3 presents the hybrid parallel CPU/GPU results. Section 4.4 compares the results obtained from each architecture platform.

4.1 Sequential CPU Implementation

As previously noted, the quantities under observation in this study are the time spent in seconds applying forces to the system, the time spent solving the system using the CG algorithm, the time spent within the CG algorithm performing matrix/vector multiplications, and the total time taken to generate a frame ready to be sent to the display. The level 2 BLAS function that multiplies a matrix by a vector is called *GEMV, where the * denotes whether the values are single precision or double precision and whether they are real or complex numbers. SGEMV is used for real single precision matrices and vectors; this call is used in the application. DGEMV would be used for double precision values. CGEMV and ZGEMV are used for single and double precision complex values respectively (BLAS - Basic Linear Algebra Subprograms, 2009).

Table 4-1 contains the average value recorded for each of the observation quantities over the course of 250 frames of animation for each cloth mesh resolution. Figure 4-1 presents pie

Table 4-1. Sequential CPU: Average task run-time in seconds over 250 frames

Mesh Size	Force Application	CG GEMV	CG Solve	Total Time	Standard Deviation of Total Time
20x20	0.000569	0.001198	0.001508	0.00227	0.005523194
40x40	0.044468	0.15419	0.188887	0.237762	0.009421519
60x60	0.227298	0.813879	0.978891	1.207762	0.030709077

charts that visualize the amount of time spent in each phase as a percentage of total time spent preparing the frame for rendering. Figure 4-1 illustrates how demanding the CG algorithm is relative to the rest of the simulation, requiring 67 - 81% of the total calculation time. The figure also illustrates how computationally demanding the GEMV matrix/vector multiplications are, requiring 79 - 82% of the total time spent running the CG algorithm. These percentages are consistent with those observed by Baraff and Witkin (1998) when considering the relative time

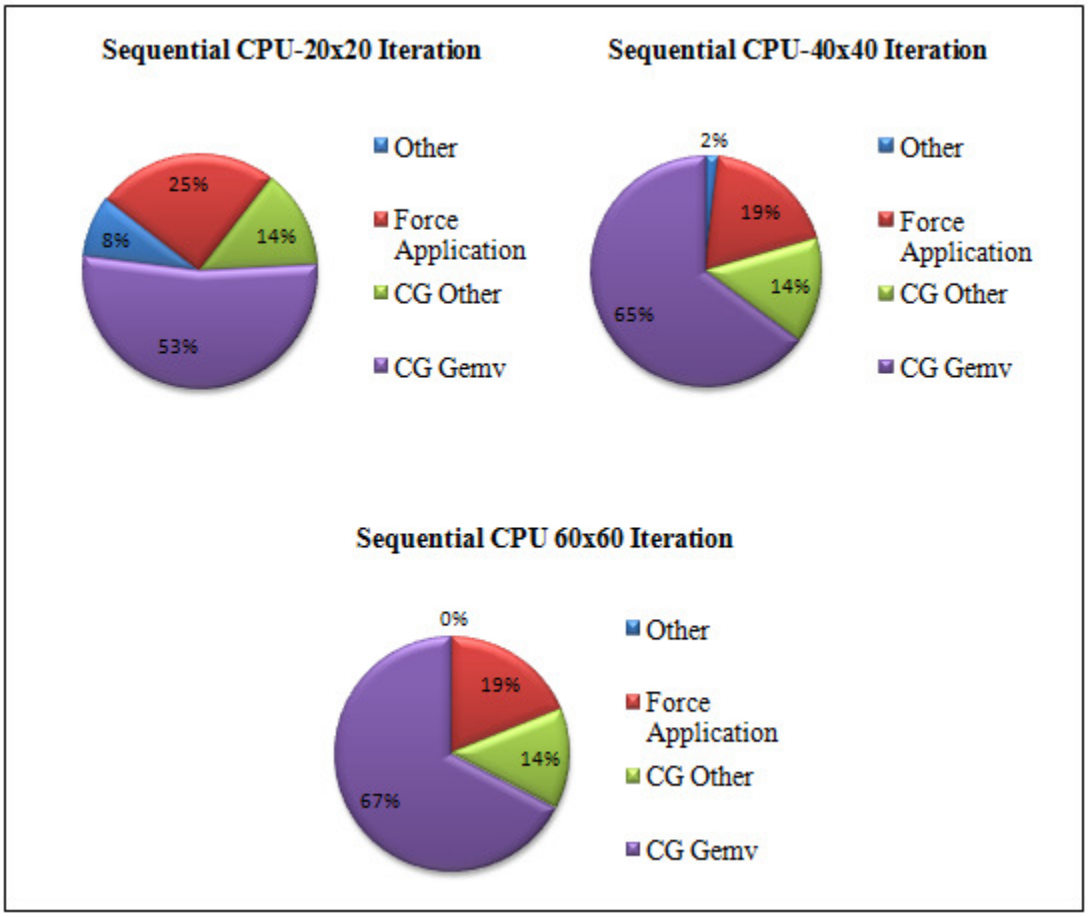

Figure 4-1. Sequential CPU task breakdown percentages

spent performing their CG implementation and force application phases. It is worth noting that Baraff and Witkin also perform collision detection, which adds to the total runtime of their simulation, making the time spent performing CG calculations and force applications a smaller

percentage of their total simulation time. The transition from a 20x20 to a 40x40 mesh increases run-time by a factor of 104.7. The transition from a 40x40 mesh to a 60x60 mesh increases run-time by a factor of 5.08.

4.2 Parallel CPU Implementation

Figure 4-2 and Table 4-2 present the task breakdown percentages and average run-times observed for the Parallel CPU implementation of the software. The force application phase of the simulation is greatly aided by the use of OpenMP #pramga statements. The application of forces when parallelized happens so quickly at low mesh densities that it does not even register as having happened with the 20x20 mesh. There are also significant reductions in force application time for the 40x40 and 60x60 meshes. At the 60x60 mesh size, force application occurs 2.74 times more quickly than using the sequential code. The researcher expects this ratio to increase as mesh density increases. However this is far from a linear speedup considering the availability of 16 threads. While gains were achieved during the force application phase, the CG algorithm did not benefit from the introduction of Open MP `#pragma` statements to the blocks of code suitable for such insertion within the CG algorithm. In fact, the CG algorithm takes longer than in the sequential cases for larger meshes. The overhead associated with the creation and destruction of threads by Open MP negates any nominal performance gain that may have been derived.

Table 4-2. Parallel CPU: Average task run-time in seconds over 250 frames

Mesh Size	Force Application	CG GEMV	CG Solve	Total Time	Standard Deviation of Total Time
20x20	0	0.001073	0.001456	0.001948	0.005166193
40x40	0.015944	0.164294	0.190149	0.207214	0.019019665
60x60	0.082935	0.881956	0.996698	1.090496	0.128663101

Recall from Chapter 3, that Intel's MKL was chosen because of its ability to take advantage of CPU parallelism when run on a system featuring multiple processors. MKL

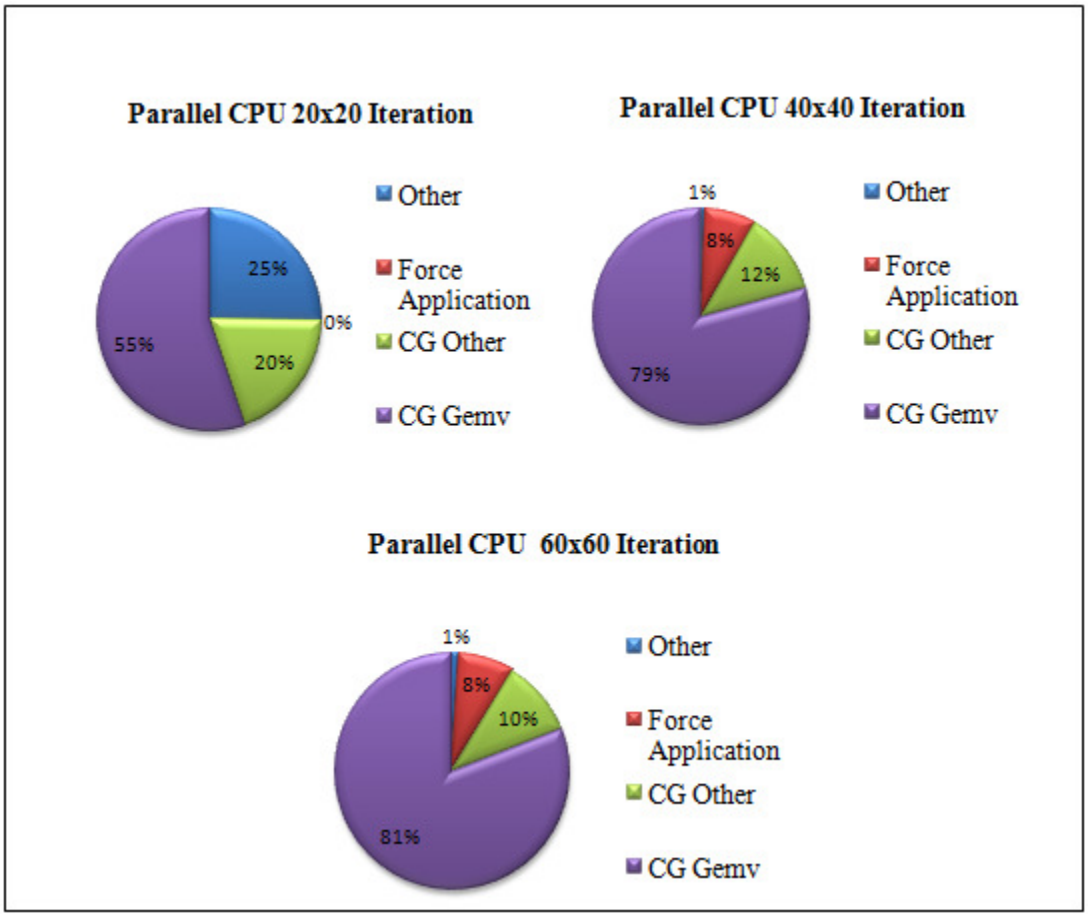

Figure 4-2. Parallel CPU task breakdown percentages

parallelism is not occurring in the calls to CBLAS_SGEMV. MKL parallelism exists primarily in Level 3 BLAS, which is not utilized in this study. The lack of parallelism observed could be a function of a defect in the software implementation or a reality of Intel's design. Future experiments are required. However, what is apparent from the data is that OpenMP `#pragma` statements preceding `For` loops do improve the performance of the cloth simulation algorithm implemented in this study. The performance gains are negligible relative to the number of threads available and suggest that this is not the best way to parallelize a cloth algorithm if the

goal is to fully exploit the available CPU parallelism. Thomaszewski and Blochinger (2006) achieved much better results with their application of domain decomposition techniques, recording almost linear speedup when using 4 cores. The transition from a 20x20 mesh to a 40x40 mesh increases run-time by a factor of 106.4. The transition from a 40x40 mesh to a 60x60 mesh increases run-time by a factor of 5.26.

4.3 Hybrid Parallel CPU/GPU Implementation

Table 4-3 and Figure 4-3 illustrate that the hybrid parallel CPU/GPU implementation has performance characteristics that differ from the CPU methods. The breakdown of the time spent running GEMV calculations is actually substantially smaller than the other CG operations in the 20x20 mesh case. This skews the impact of the matrix/matrix additions that occur at the outset of the call to CG. Recall that these additions are not handled by BLAS. Because the matrix/vector multiplies are being handled by the GPU, this data must be transferred across the PCI-Express bus. This data transfer is a source of additional overhead that the GPU introduces and is included in the GEMV times for this implementation as reported in Table 4-3 and Figure 4-3. Figure 4-4 illustrates how large this overhead can be. For all but the smallest mesh, the amount of time it takes to get the data to the GPU exceeds the amount of time it takes to perform the calculations. This is especially problematic for interactive simulations that require high frame rates.

Figure 4-5 illustrates the amount of time spent by the GPU performing data transfers and matrix/vector multiplies for each mesh density observed. The figure clearly illustrates how

Table 4-3. Hybrid parallel CPU/GPU: Average task run-time in seconds over 250 frames

Mesh Size	Force Application	CG GEMV	CG Solve	Total Time	Standard Deviation of Total Time
20x20	0.000125	0.00019	0.002645	0.003028	0.006805214
40x40	0.015839	0.073895	0.111153	0.127935	0.00983898
60x60	0.06452	0.241669	0.545702	0.616327	0.024251427

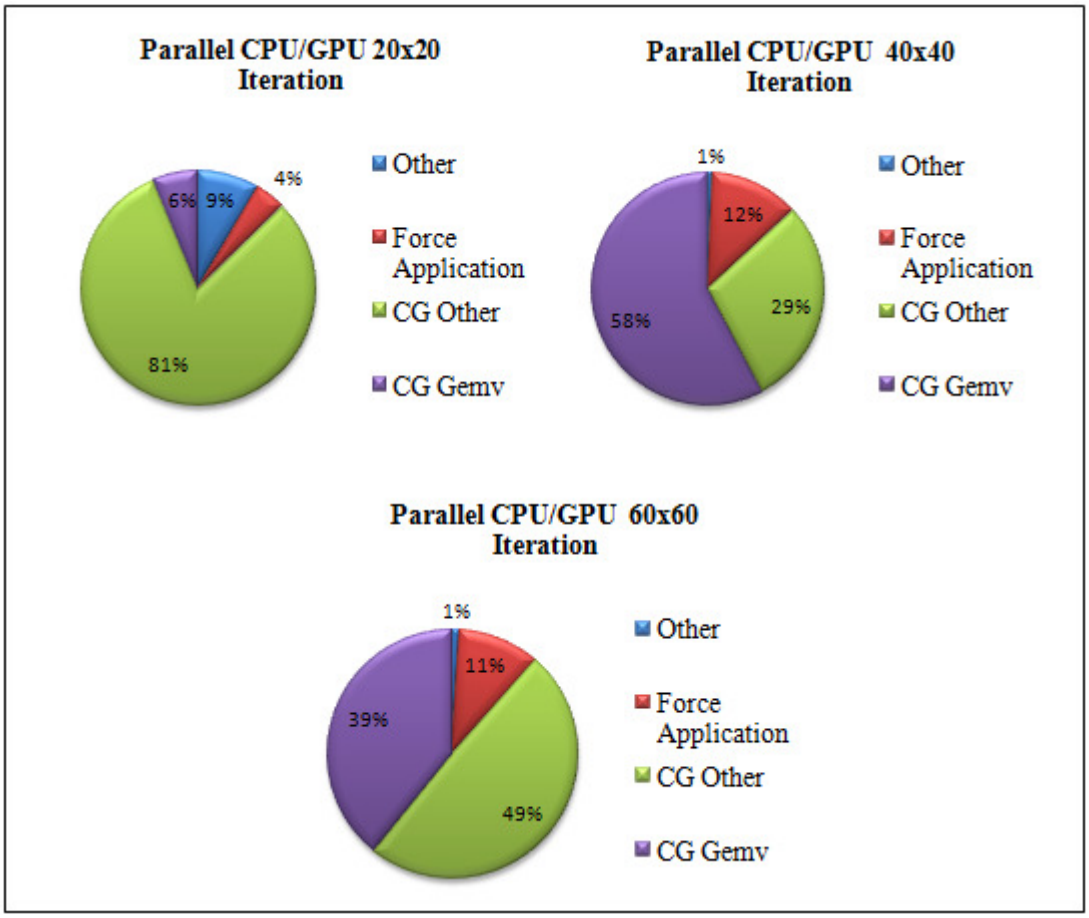

Figure 4-3. Hybrid parallel CPU/GPU task breakdown percentages

demanding the 60x60 mesh is relative to the lower resolution meshes. The data illustrated in Figure 4-4 and 4-5 was obtained from Nvidia's CUDA Visual Profiler. The visual profiler runs a selected application and collects information about its performance directly from the GPU. Note that the application runs slower while it is being profiled, so this data is only useful to the extent in which it compares profile times relative to one another. The transition from a 20x20 mesh to a 40x40 mesh increased runtime by a factor of42.3. The transition from a 40x40 mesh to a 60x60 mesh increased runtime by a factor of 4.82.

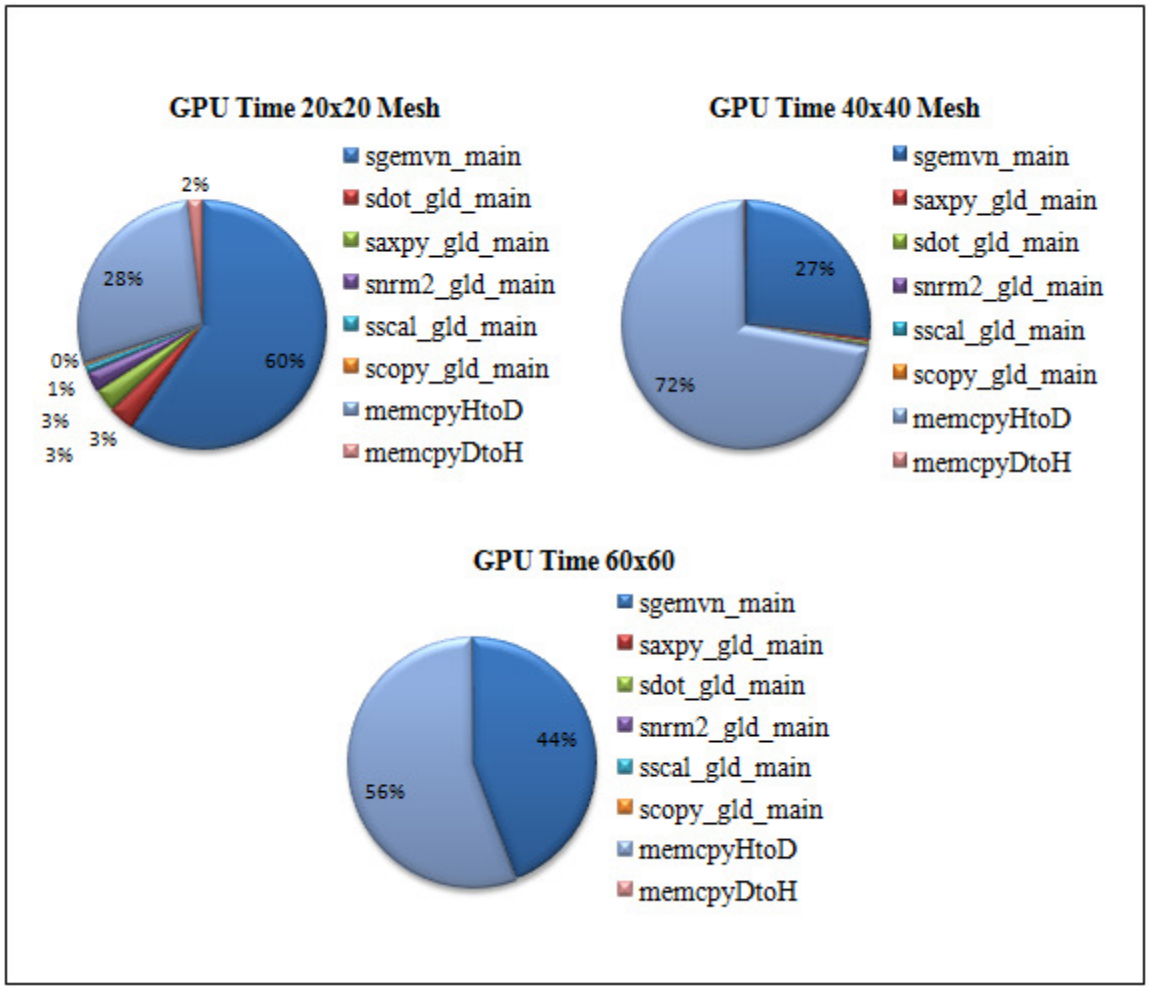

Figure 4-4. GPU task breakdown percentages

4.4 Architectural Comparisons

Figure 4-6 illustrates the relative performance of the implementations when processing a 20x20 mesh. The parallel CPU implementation emerges as the fastest. It is able to outperform the sequential CPU implementation by virtue of its lower force application times. The hybrid parallel CPU/GPU implementation is limited by its longer total CG times despite having faster matrix/vector multiplication times. This is a clear example of the penalty associated with transferring data to and from the GPU not being overcome by the speed of the GPU due to the small problem size. Figure 4-7 illustrates the relative performance of the implementations when processing a 40x40 mesh. The hybrid parallel CPU/GPU implementation is the fastest

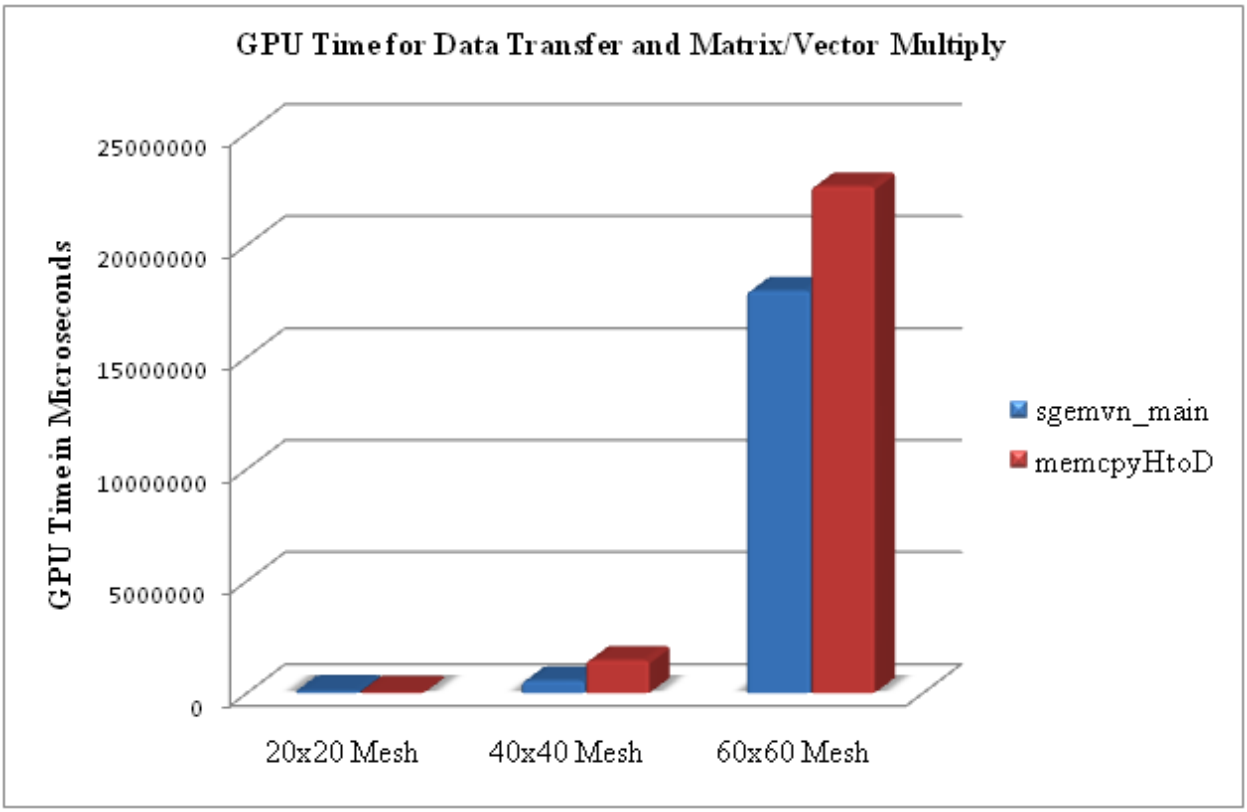

Figure 4-5. GPU time for data transfer and matrix/vector multiply

method. The speed with which the GPU is able to process the matrix/vector multiplies offsets the disadvantage of transferring the matrix data to it over the PCI-Express bus. The nearly identically force application times between the hybrid method and the parallel CPU method indicate that the primary advantage of using the OpenMP `#pragma` statements in this study is not compromised by allowing the GPU to handle the CG algorithm.

Figure 4-8 illustrates the relative performance of the implementations when processing a 60x60 mesh. The hybrid parallel CPU/GPU implementation is able to further exploit its computational advantage over the other implementations. The matrix/vector multiplication is nearly 4 times faster than the other methods. The overall advantage is reduced due to the data transfer penalty but the hybrid method is approaching being twice as fast as the sequential CPU at this mesh size. Figure 4-9 illustrates the increase in run-time that results from transitioning

from the 20x20 mesh to the 40x40 mesh, and transitioning from the 40x40 to the 60x60 mesh. The sequential CPU implementation and the parallel CPU implementation experience nearly identical rates of performance degradation. The hybrid parallel CPU/GPU implementation experiences far less degradation in transitioning from the 20x20 mesh to the 40x40 mesh than the

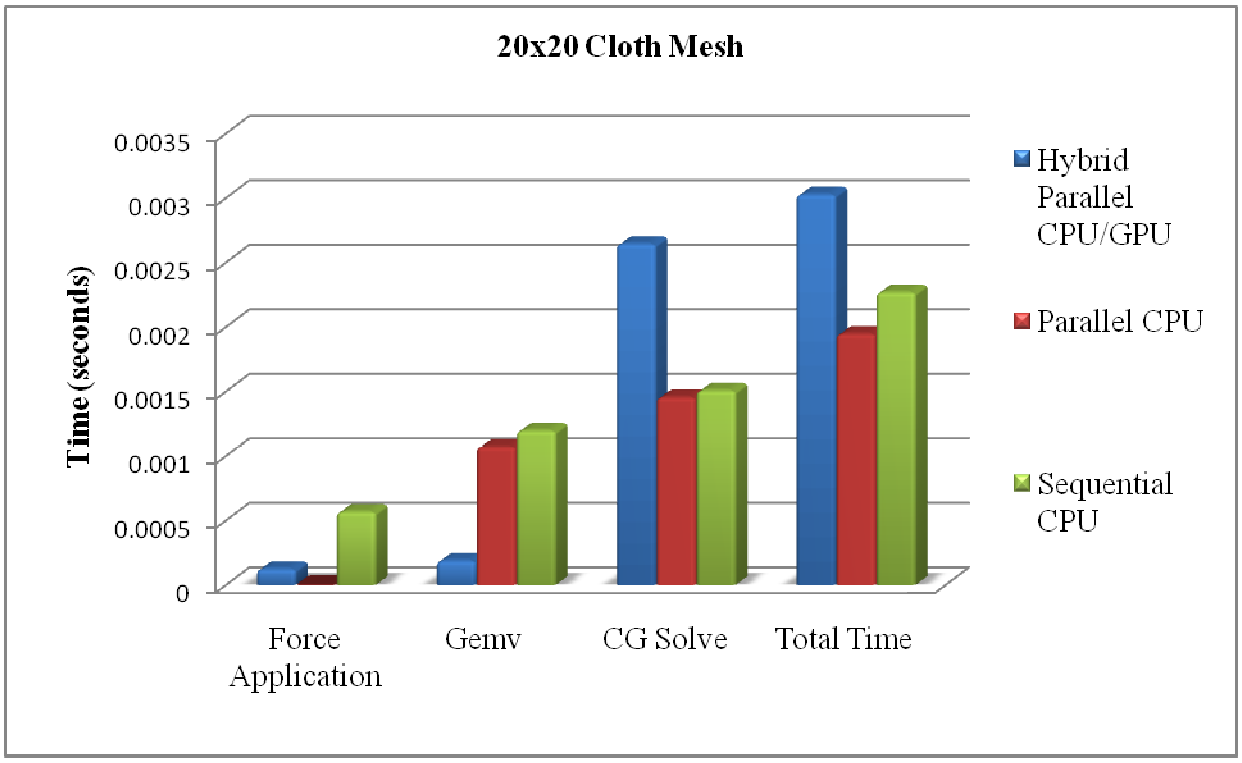

Figure 4-6. Architectural comparisons on a 20x20 cloth mesh

other two implementations. While it experiences less degradation in transitioning for the 40x40 mesh to the 60x60 mesh, the difference is far less dramatic. The rate of performance gain achieved by using the hybrid implementation degrades as the mesh get's larger. This is not immediately obvious from inspecting Figures 4-6, 4-7, and 4-8. The key insight comes from appreciating the fact that the hybrid implementation went from being slower at the 20x20 mesh to faster at the 40x40 mesh. This rate of performance increase exceeds the rate at which it improves upon its speed advantage when transitioning from the 40x40 mesh to 60x60 mesh.

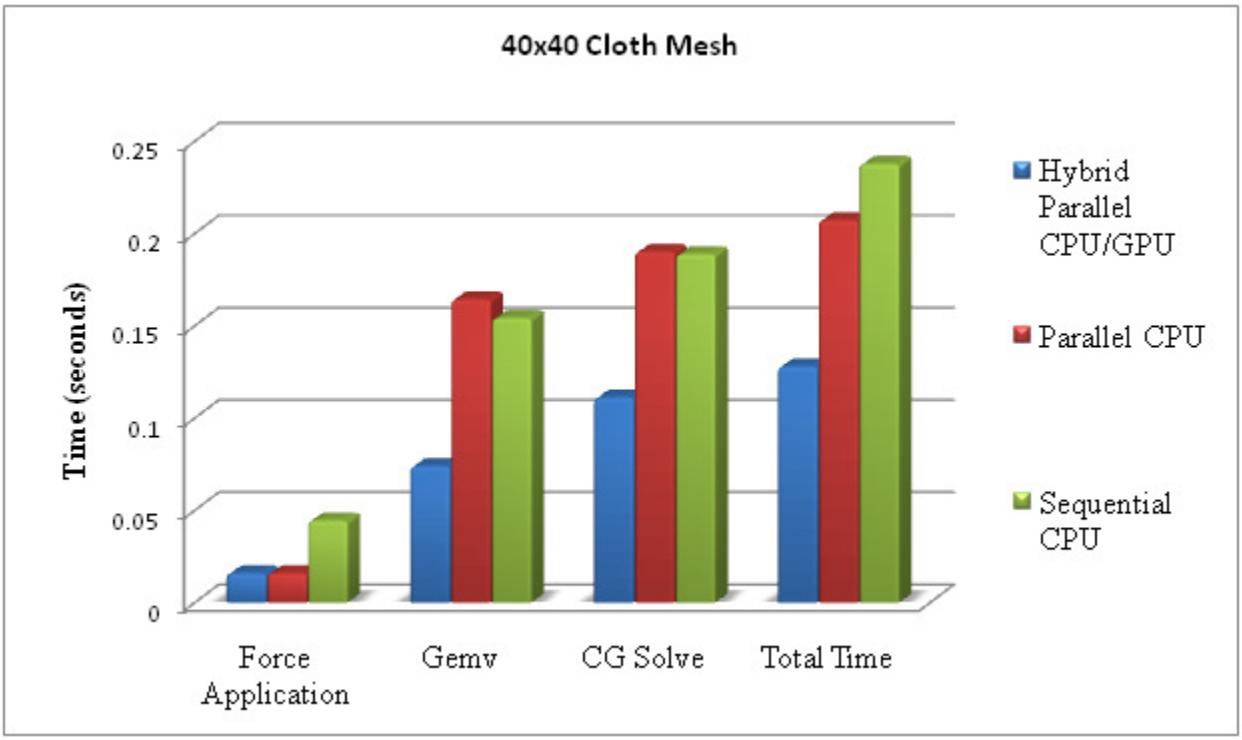

Figure 4-7. Architectural comparisons on a 40x40 cloth mesh

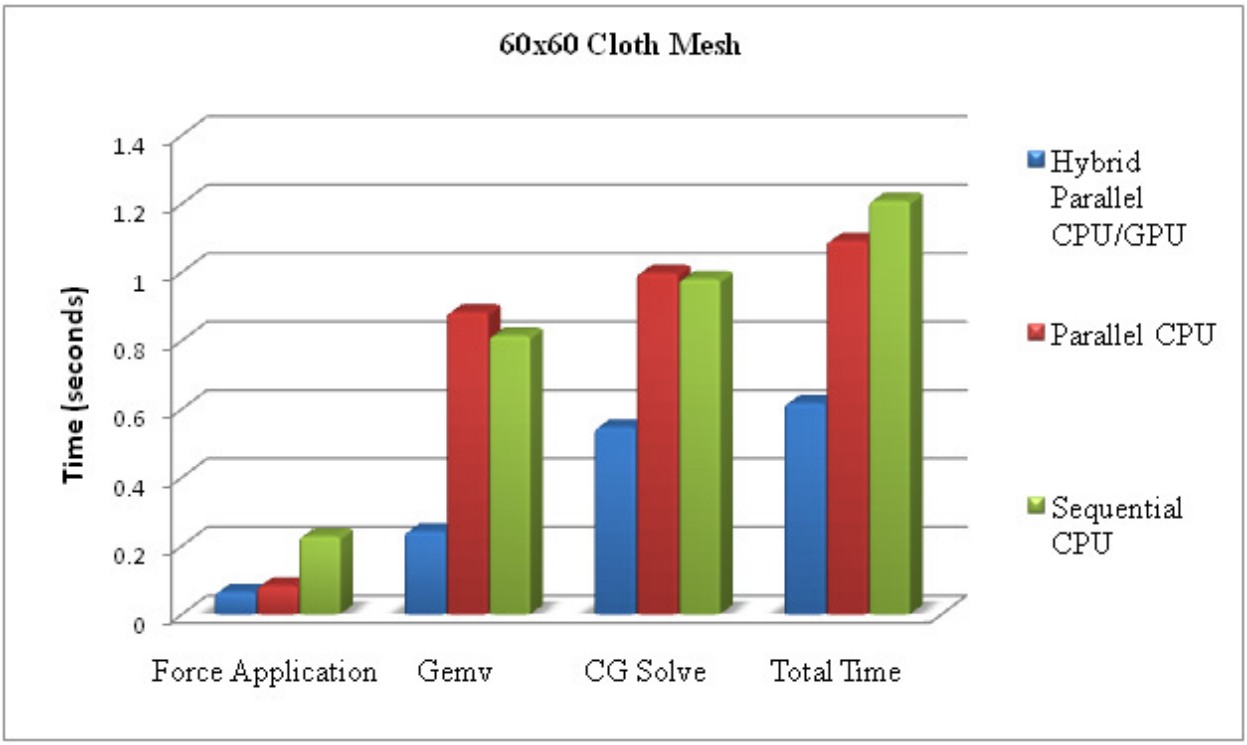

Figure 4-8. Architectural comparisons on a 60x60 cloth mesh

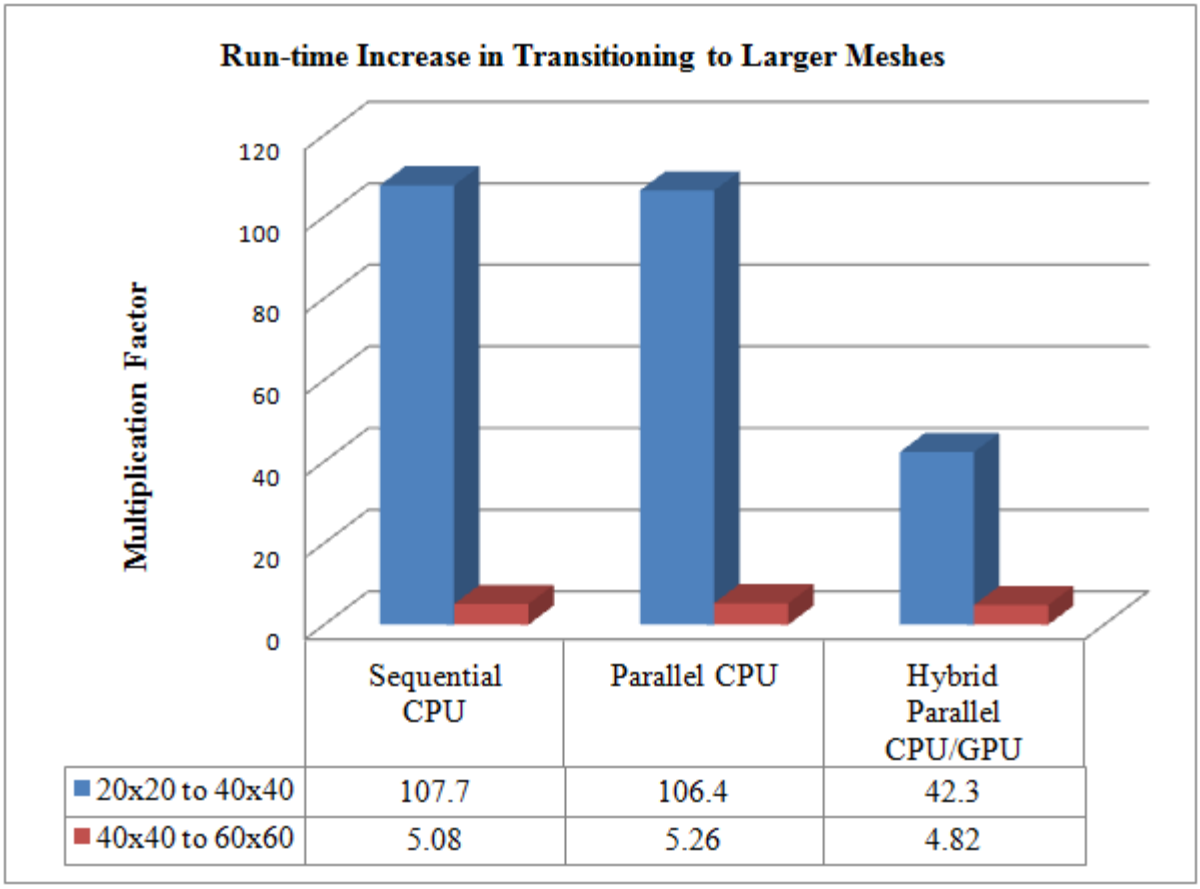

Figure 4-9. Run-time increase in transitioning to larger meshes

Note that we are merely discussing the rate of increasing speed advantage, not the presence of a speed advantage. This change in the rate of change does potentially indicate the existence of a mesh size in which the hybrid method will cease to be faster. The lack of discrepancy between the sequential and parallel CPU methods is another indication of how little impact the simple use of OpenMP `#pragma` statements used in this study are. Since the `#pragma` statements primarily affect the speed gains observed in the force application stage, the lack of improvement in the CG solve stage is able to overshadow them because the CG solve is the dominate computational bottleneck.

Chapter 5 Conclusions and Recommendations

This study evaluated the use of a hybrid parallel CPU/GPU implementation of a cloth simulation algorithm. The purpose of this study was to determine whether this hybrid implementation could offer higher animation frame rates than both sequential CPU and parallel CPU implementations of the same algorithm. This purpose was fulfilled through the completion of the four objectives.

Fulfillment of Objective 1 was achieved by developing the base sequential CPU implementation of the cloth simulator. The simulator relied on the Backward Euler implicit integration technique to setup the system of equations that represent the state of the cloth system. The Conjugate Gradient algorithm was used to solve this system of equations at each time step. The matrix/vector, vector/vector, vector/scalar operations associated with the CG algorithm were performed using calls to the implementation of BLAS found in the Intel Math Kernel Library.

Fulfillment of Objective 2 was achieved when the above cloth implementation was modified to include OpenMP `#pragma` statements that parallelized the large loop associated with the application of forces in the simulated system. Fulfillment of Objective 3 was achieved by modifying the software produced in fulfillment of object two. This modification consisted of performing the CG algorithm using the Nvidia CUDA based BLAS implementation CUBLAS instead of the Intel MKL BLAS implementation.

Objective 4 was fulfilled by running the simulation for 250 frames while it outputted to a console window the times required to complete the force application stage, the CG Solve stage, the total time spent during the CG Solve stage performing matrix/vector multiplications, and the total time required to produce the data for rendering each frame.

The results of this study showed that the hybrid implementation was able to maintain the benefits obtained through CPU parallelism, while adding the benefit of GPU parallelism for the 40x40 mesh and the 60x60 mesh. The speedup obtained in the matrix/vector multiplications could not overcome the time penalty associated with transferring data to the GPU over the PCI-express bus in the case of the 20x20 mesh. While one can reject $H1_0$ for the meshes of all observed dimension, one can only accept $H1_A$ for the 40x40 and 60x60 meshes. One can also reject $H2_0$ for all observed mesh resolutions, however the lack of a more effective implementation of CPU parallelism does not allow one to accept $H2_A$ for any observed mesh resolution.

The limited effectiveness of the CPU parallelism implemented in this study is the major barrier to one's ability to generalize the results of this study to all parallel CPU implementations. That being said, the author believes that there are few parallel CPU implementations that wouldn't benefit from allowing the GPU to perform the matrix/vector multiplications associated with the CG algorithm when working with high resolution meshes. Determining which parallel CPU algorithms benefit from allowing the GPU to execute the CG algorithm is an excellent topic for future research, most notably parallel algorithms that subdivide the cloth mesh.

The results of this research could also be evaluated in a context that utilized more efficient matrix storage techniques. The use of general matrix storage on the sparse matrices associated with the simulation of cloth increases the amount of data that must be sent to the GPU and from system memory to the CPU. Using sparse storage techniques reduces the memory requirements of the matrix but adds complexity to the matrix/vector multiplication algorithm. This added complexity could have an effect on the performance advantage observed in this study for the GPU.

The use of multiple GPUs is also a logical choice for continued investigation, especially if combined with the domain decomposition CPU parallelism techniques used by Thomaszewski et al. (2007). Under this scenario, each subdivision of the original cloth mesh could be paired with a GPU for the execution of the matrix/vector operations contained within the CG algorithm. Presently, one can equip a workstation with 8 Nvidia GPUs. All that is required is the use of a motherboard with four PCI-Express x16/x8 slots and the use of four Nvidia graphics cards that feature two GPUs, such as the 9800GX2 or the GTX295. The idea of paring a GPU with each cloth subdivision is more easily realized in those cloth simulations that utilize clusters. Under the cluster scenario, one need only equip each cluster node with a single GPU. One could also equip each node with multiple GPUs and combine the two approaches. It is important to note that the penalty for transferring the matrix data to the GPU observed in this study is far less severe than the penalty associated with the transfer of data over a switched network to each cluster node.

While the author believes that the results of this study are a valuable addition to cloth simulation research associated with pre-rendered animation, it has limited impact on research intended for application in interactive animations. The time penalty associated with the transference of the matrix data to the GPU is too severe to maintain interactive frame rates. Offloading the entire cloth simulation to the GPU (Zeller, 2007) (Dencker, 2006), remains the fastest method for simulating cloth in interactive applications that do not require a very accurate physical model of the cloth. Emerging architectures such as Intel's Larrabee that attempt to blur the distinction between the CPU and GPU will also have an impact on cloth simulation research going forward. The many-core approach to CPU design will reduce the advantage that the many-core GPU currently has in performing matrix/vector operations. As the matrix/vector operation

performance that is achievable with CPUs approaches that of GPUs, the penalty for transferring matrix data to the GPU will be insurmountable.

References

1. Baker, P. (2003, 1 12). *Paul's Projects - Cloth Simulation.* Retrieved 2009, from Paul's Projects: http://www.paulsprojects.net/opengl/cloth/cloth.html

2. Baraff, D., & Witkin, A. (1998). Large steps in cloth simulation. *Proceedings of ACM SIGGRAPH 98* (pp. 43-54). ACM Press.

3. Bender, J., & Bayer, D. (2008). Parallel simulation of inextensible cloth., (pp. 47-56).

4. Birra, F. P., & Próspero dos Santos, M. (2006). A GPU Subdivision Algorithm for Cloth Simulation. *VIRtual , Special Edition: Advances in Computer Graphics in Portugal*.

5. *BLAS - Basic Linear Algebra Subprograms*. (2009). Retrieved from The Netlib: http://www.netlib.org/blas/

6. Bolz, ,. J., Farmer, ,. I., Grinspun, ,. E., & Schröder, ,. P. (2003). Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph. , 22*, 917--924.

7. Curtis-Maury, M. F. (2008). *Improving the Efficiency of Parallel Applications on Multithreaded and Multicore Systems.* Blacksburg: Virginia Polytechnic Institute and State University.

8. Dencker, K. (2006). *Cloth Modelling on the GPU.* Trondheim: Norwegian University of Science and Technology.

9. Gutiérrez, E., Romero, S., Romero, L., Plata, O., & Zapata, E. (2005). Parallel techniques in irregular codes: cloth simulation as case of study. *J. Parallel Distrib. Comput. , 65*, 424--436.

10. Harris, M. (2009). *Workshop on High Performance GPU Computing with NVIDIA CUDA.* Retrieved from School of Computer Science and Engineering University of New South Wales: http://www.cse.unsw.edu.au/~pls/cuda-workshop09/slides/01_TeslaCUDAIntro.pdf

11. House, D., & Breen, D. (Eds.). (2000). *Cloth Modeling and Animation.* A.K. Peters.

12. Howard, T. (2006). *Toby Howard :: OpenGL frame rate displayer.* Retrieved from Toby Howard :: Some hopefully useful things: http://www.cs.manchester.ac.uk/~toby/frameratedisplayer.htm

13. Hughes, C., & Hughes, T. (2008). *Professional Multicore Programming - Design and Implementation for C++ Developers.* Indianapolis: Wiley Publishing, Inc.

14. Jacobs, B. (2007). *OpenGL Video Tutorial - Basic Shapes.* Retrieved from OpenGL Tutorial: http://www.videotutorialsrock.com/

15. Keckeisen, M., & Blochinger, W. (2004). Parallel Implicit Integration for Cloth Animations on Distributed Memory Architectures. *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization.* Blackwell Publishing.

16. Lario, R., García, C., Prieto, M., & Tirado, F. (2001). Rapid parallelization of a multilevel cloth simulator using OpenMP. *Proceedings of European Workshop on OpenMP (EWOMP 2001).*

17. Macri, D. (2008, October 22). *Simulating Cloth for 3D Games - Intel® Software Network.* Retrieved from Intel Software Network Communities - Intel Software Network: http://software.intel.com/en-us/articles/simulating-cloth-for-3d-games/

18. Provot, X. (1995). Deformation constraints in a mass-spring model to describe rigid cloth behavior. *Proceedings of Graphics Interface (GI 1995)* (pp. 147-154). Canadian Computer-Human Communications Society.

19. Romero, S., Romero, L., & Zapata, E. (2000). Fast cloth simulation with parallel computers. *Proceedings of 6th International Euro-Par Conference* (pp. 491-499). Springer-Verlag.

20. Selle, A., Su, J., Irving, G., & Fedkiw, R. (2009). Robust High-Resolution Cloth Using Parallelism, History-Based Collisions, and Accurate Friction. *IEEE Transactions on Visualization and Computer Graphics , 15* (2), 339-350.

21. Shewchuk, J. R. (2004, August). *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.* Retrieved from Jonathan Shewchuk's papers: http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

22. Thomaszewski, B., & Blochinger, W. (2006). Parallel Simulation of Cloth on Distributed Memory Architectures.

23. Thomaszewski, B., Pabst, S., & Blochinger, W. (2007). Exploiting Parallelism in Physically-Based Simulations on Multi-Core Processor Architectures. *Eurographics Symposium on Parallel Graphics and Visualization 2007*, (pp. 69--76,). Lugano, Switzerland},.

24. Wang, Y. (2005). *GPU Based Cloth Simulation on Moving Avatars.* Baltimore: University of Maryland at Baltimore County.

25. Witkin, A., & Baraff, D. (1997). *Physically Based Modeling: Principles and Practice.* Retrieved from http://www.cs.cmu.edu/~baraff/sigcourse/

26. Yang, D. (2001). *C++ and Object Oriented Numeric Computing for Scientists and Engineers.* New York: Springer-Verlag.

27. Zamith, M., Clua, E., Conci, A., Montenegro, A., Leal-Toledo, R., Pagliosa, P., Valente, L., & Feij, B. (2008). A game loop architecture for the GPU used as a math coprocessor in real-time applications. *Comput. Entertain. , 6*, 1--19.

28. Zara, F., Faure, F., & Vincent, J. M. (2002). Physical cloth simulation on a PC cluster. *Proceedings of 4th Eurographics Workshop on Parallel Graphics and Visualization* (pp. 105-112). ACM Press.

29. Zara, F., Faure, F., & Vincent, J.-M. (2004). Parallel Simulation of Large Dynamic System on a PCs Cluster: Application to Cloth Simulation. *Int. J. Comput. Appl. , 26*, 173--180.

30. Zeller, C. (2007, February 15). Cloth Simulation. Nvidia Corporation.

Appendix: CG Algorithm Using CUBLAS

```
void ConjugateGradient(){

const int maxiter = 10000;
const float eps = 1.0e-7;
float twoNorm;
float h2;
float gemvTotal = 0;
SYSTEMTIME gemvStart,gemvStop;

int iter;
h2 = STEP_TIME*STEP_TIME;

// Establish A for CG Solution of Ax = b
// A = massMatrix - (deltaT*dfdv) - ((deltaT*deltaT)*dfdx);
#pragma omp parallel for
for (int i=0; i<n3*n3; i++)
     A[i] = massMatrix[i] - (STEP_TIME*dfdv[i]) - (h2*dfdx[i]);

// Establish b for CG Solution of Ax = b
// b =  deltaT * (f + (deltaT*(dfdx)*(v)))

// Copies vector f into dfdxTimesV variable in order to avoid a
// separte sumation of f + (deltaT)(df/dx)v

cblas_scopy(n3,forceVector,1,dfdxTimesV,1);

// Start gemv time accumulator
GetSystemTime(&gemvStart);

// Send matrix and vectors to GPU
cublasSetMatrix (n3, n3, sizeof(*dfdx), dfdx, n3, devPtrdfdx, n3);
cublasSetVector (n3, sizeof(*velocityVector), velocityVector, 1,
                 devPtrvelocity, 1);
cublasSetVector (n3, sizeof(*dfdxTimesV), dfdxTimesV, 1,
                 devPtrdfdxTimesV, 1);
cublasSetVector (n3, sizeof(*deltaV), deltaV, 1, devPtrdeltaV, 1);

// gemv: y = alpha*A*x + beta*y
//       y = f + (deltaT*(df/dx)*(v)
// Note f is the result of copying f into dfdxTimesV above
cublasSgemv('N',n3,n3,STEP_TIME,devPtrdfdx,n3,devPtrvelocity,1,1,devPt
            rdfdxTimesV,1);

// check for error after gemv
stat = cublasGetError();
if (stat == CUBLAS_STATUS_EXECUTION_FAILED)
     cout << "Function failed" << endl;

// Reuse GPU memory for df/dx to store A
```

```
cublasSetMatrix (n3, n3, sizeof(*A), A, n3, devPtrdfdx, n3);

// End first gemv accumulator
GetSystemTime(&gemvStop);
gemvTotal = gemvTotal+((gemvStop.wMinute - gemvStart.wMinute)*60)+
                ((gemvStop.wSecond-gemvStart.wSecond))+
                ((gemvStop.wMilliseconds -
                 gemvStart.wMilliseconds)/1000.0);

// b =  deltaT * (f + (deltaT*(dfdx)*(v)))
// b =  deltaT * devPtrdfdxTimesV
cublasSscal(n3,STEP_TIME,devPtrdfdxTimesV,1);
cublasScopy(n3,devPtrdfdxTimesV,1,devPtrb,1);

//r = b;
cublasScopy(n3,devPtrb,1,devPtrr,1);

//p = r;
cublasScopy(n3,devPtrr,1,devPtrp,1);

float zr;
// inner product of (r,r);
zr = cublasSdot(n3,devPtrr,1,devPtrr,1);

// twoNorm of b
twoNorm = cublasSnrm2(n3,devPtrb,1);

const float stp = eps * twoNorm;

for (iter = 0; iter < maxiter; iter++){

      float mpDOTp;

      // mp = A*p
      GetSystemTime(&gemvStart);
      // devPtrdfdx is actually A

      cublasSgemv('N',n3,n3,1.0,devPtrdfdx,n3,devPtrp,1,0,devPtrmp,1);
      GetSystemTime(&gemvStop);
      gemvTotal = gemvTotal+((gemvStop.wMinute -
                  gemvStart.wMinute)*60)+
                  ((gemvStop.wSecond-gemvStart.wSecond))+
                  ((gemvStop.wMilliseconds -
                  gemvStart.wMilliseconds)/1000.0);


      // mpDOTp = inner product of (mp,p);
      mpDOTp = cublasSdot(n3,devPtrmp,1,devPtrp,1);

      float alpha = zr/mpDOTp;

      // saxpy: y = y + alpha*x
```

```
      // deltav = deltav + alpha*p
      cublasSaxpy(n3,alpha,devPtrp,1,devPtrdeltaV,1);

      // saxpy: y = y + alpha*x
      //        r = r - alpha*mp
      cublasSaxpy(n3,-alpha,devPtrmp,1,devPtrr,1);

      //twoNorm of r
      twoNorm = cublasSnrm2(n3,devPtrr,1);

      if (twoNorm <= stp) break;

      float zrold = zr;
      //zr = inner_prod of (r,r)
      zr = cublasSdot(n3,devPtrr,1,devPtrr,1);

      // p = r + (zr/zrold)*p
      cublasSscal(n3,(zr/zrold),devPtrp,1);
      cublasSaxpy(n3,1,devPtrr,1,devPtrp,1);
}
cublasGetVector (n3,sizeof(*deltaV),devPtrdeltaV,1,deltaV,1);
cout << "; Gemv: " << gemvTotal;
}
```

Vita

Gillian David Sims was born in July, 1977, in New York, New York. In 1999, he graduated *Magna Cum Laude* from Louisiana State University, receiving a Bachelor of Science with a concentration in textiles, apparel design, and merchandising. In 2002, he received a Master of Science degree from the School of Human Ecology at Louisiana State University. He currently serves as an adjunct instructor at Southern University in the Department of Family and Consumer Sciences.