USING GPU-BASED COMPUTING TO ACCELERATE FINITE ELEMENT PROBLEMS

by

JACOB WATT

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN AEROSPACE ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2012

ACKNOWLEDGEMENTS

I would first like to give my deepest gratitude to my research advisor Dr. Brian Dennis for all his support and encouragement throughout my classes and research. Without his help and guidance my research would not have been possible.

I would also like to thank NVIDIA for their generous gift of the Quadro 5000 graphics card on which my research was conducted.

Most importantly, I would like to thank my parents, siblings, grandparents, aunts, uncles and cousin for all their support through all the highs and lows in my life.

April 13, 2012

ABSTRACT

USING GPU-BASED COMPUTING TO ACCELERATE

FINITE ELEMENT PROBLEMS

Jacob Watt, M.S.

The University of Texas at Arlington, 2012

Supervising Professor: Brian Dennis

Historically Graphics Processing Units (GPU) have been used for offloading graphical visualization and made popular in use for video games, but with the development of NVIDIA's CUDA architecture and programing language there has been an increase in the use of GPUs in general purpose (GPGPU) programing. Problems involving large systems of linear equations, such as the Finite Element Analysis (FEA), can benefit greatly from the parallel computing capabilities of GPUs. In my thesis I will solve Poisson's equation and discuss the advantages and disadvantages of the massively parallel environment of GPUs. I will show that on a unstructured grid that the matrix-vector multiplication can be run 15.4 times faster on the GPU when compared to an Intel i5 CPU. For all cases, 2-D triangular elements with linear basis functions were used. When the linear algebra problem is solved using the biconjugate gradient stabilized method (BiCGSTAB), the method runs 14.4 faster on the GPU as compared to the serial C code. And lastly, solving the whole FEA including data setup, element integration, assembly, and memory transfer times preforms 11.8 faster on the NVIDIA GPU compared to the serial code run on an Intel i5 CPU.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

1.1 Introduction

Up until a few years ago, CPUs focused heavily on clock rate for increase in performance. This was a practical and simple strategy for many years. Eventually, heat dispersal due to increasing power density on the CPU core led to a near immediate halt in clock rate increases at about 4 GHz. To increase computing power, two or more chips were placed on the same CPU. Four and six core processors are quickly becoming the new standard. These multi-core processors are very useful in general purpose computing use but still lack the raw potential for high performance mathematical computations. If more cores where desired, several workstations would need to be networked into a large system called a cluster. These clusters are typically expensive and provide a poor performance to cost ratio as well as bringing diminishing returns for each additional workstation as explained by Amdahl's Law [17]. Performance has been further increased through technical improvements in RAM access latency and clocking speed as well as increased size and performance of CPU onboard memory cache. Both will eventually hit the point where further improvements are either technically infeasible or simply no longer bring any benefit to CPU performance.

GPUs, on the other hand, contain upwards of hundreds of cores on a single chip. These cores, unlike the cores of a CPU, have a very specific purpose and are not built to handle complex tasks. Despite their overall simplicity, the cores are very efficient at handling mathematical computations. Because of the computational efficiency and number of cores of a GPU the core clock speed is much lower than what is found on most desktop grade CPUs. Having a lower core speed gives both a better performance to power consumption ratio as well as a decreased thermal management requirement. High core clock speeds maybe become feasible as thermal management technologies improve.

Because of the nature of the architecture of GPUs, performance of the code is highly dependent on the degree by which the data can be solved in a parallel manner. Problems such as matrix-vector multiplication can benefit greatly from the use of GPUs, as each value in the solution vector can be computed independently of each other. Conversely, problems such as dot product are inherently serial in nature and run inefficiently on GPUs without modification to the method. Speedups with dot product are possible on GPUs but special consideration of the hardware capabilities must be taken into effect.

In this paper, I will investigate how simple implementation of a sparse iterative solver on NVIDIA's CUDA architecture can result in significant speedup of a Galerkin Finite Element Method problem. I will also show that a speedup can be achieved without taking memory access or memory cache utilization into special consideration.

1.2 GPGPU

General programing for graphics cards (GPGPU) is not a new concept and was first developed in 1978 as part of the Ikonas programmable raster display system used in cockpit instrumentation. The co-founder of Ikonas Graphics Systems, Nick England, laid out the framework for Ikonas in a paper [20] for his graduate research. Before GPGPU became supported by graphics chip manufacturers, programmers would have to take advantage of the existing GPU architecture. In June 2007 NVIDIA released the first CUDA enabled chip, the G80. Shortly there after in 2008, ATI released a driver to fully support their previously released GPGPU architecture. Also in 2008 the Khronos Group released the framework Open Computing Language (OpenCL), which is supported by many CPU and GPU manufacturers. NVIDIA's

Compute Unified Device Architecture (CUDA) SDK is supported on Windows, Linux and Mac OS X operating systems. Every NVIDIA card since the G8x series is CUDA capable.

1.3 Problem Description

A simple and verifiable yet highly computationally intensive problem was desired to test the speed and accuracy of CUDA. The Poisson's equation, $\Delta\varphi=f$, is an elliptic partial differential equation identical to the Laplace equation except for the non-homogeneous source term. For this problem, a uniform source of magnitude 10 was selected and Dirichlet boundary conditions of 10 are applied to each of the four sides. The square grids, with properties listed in tab. 1, are composed of triangular cells.

Stiffness matrices from FEM are inherently sparse by nature with grids that often have poorly numbered nodes causing inefficient memory recall. Sparsely populated matrices contain large amounts of zeros which result in inefficient storage and unnecessary computations. Utilization of an appropriate storage scheme can reduce required storage and calculation requirements by many orders of magnitude and become even more efficient with larger matrices. In this paper I will show that a simple sparse storage scheme can be easily integrated into a CUDA code.

Some of the additional requirements for this thesis included learning new operating systems and programing languages. In the early stages of my work it became clear that Linux would be easier to work with when dealing with compiling and debugging code. In addition Linux has very thorough support in the CUDA development community to help with any coding issues that may have arisen. I looked at several different Linux distributions but ultimately picked Ubuntu for its stability in both operation and compatibility with my hardware.

Prior to my work on my thesis I had minimal experience with the C programing language, which is an integral part of programing in CUDA. Familiarizing myself with the C language was accomplished by reading online tutorials [19] and by converting some of my academic and personal MATLAB codes into C. To learn CUDA I mostly relied on the book *CUDA by Example* [9], as well as sample codes provided by NVIDIA and the NVIDIA community.

CHAPTER 2

DESCRIPTION OF CUDA

2.1 GPU Hardware

While CPUs are integrated into motherboards, aftermarket GPUs are typically discrete pieces of hardware which must be attached to a motherboard through a connection called a Peripheral Component Interconnect Express (PCIe) slot. Memory shared between the RAM onboard the GPU and the RAM attached to the motherboard must pass through a connection called a bus. The bus for PCIe x16 graphics cards is limited to 8GB/s in transfer. While this number is very impressive, it becomes a major bottleneck when a large number of memory transfers are required in a code. In fact, memory transfer times can easily nullify any performance gains made from use of the GPU. Maximizing GPU usage and minimizing memory transfers must be a priority when writing a program in CUDA C.

Both CPUs and GPUs share the same basic building blocks in their architecture: Random Access Memory (RAM), cache, control unit, and arithmetic logic unit (ALU). The main difference between the two are that CPUs have one control unit per each core while GPUs contain multiple control units, with each unit controlling the functionality of several ALUs. This gives an advantage in that a single instruction can be executed several times in parallel over multiple pieces of data. The figure below gives a very basic look into the structure of a GPU.

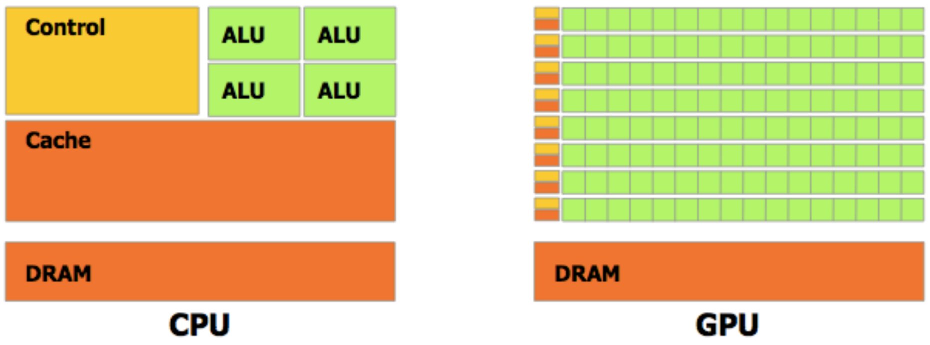

Figure 2.1 CPU and GPU Comparison. [1]

2.2 CUDA Basics

CUDA programs use blocks of code called kernels to execute code in parallel. Kernels are subroutines, which execute on the GPU as opposed to the CPU. Like subroutines they pass pointers to memory and can be used multiple times within a code. More often than not there are more kernel calls than there are cores on the GPU. To manage this CUDA has the ability to systematically handle large sets of data and instructions with minimal input from the programmer. An individual set of instructions is called a thread [2]. Each thread is assigned its own unique identification number, which makes it easy for a thread to access the appropriate entry in a set of data. A block is a set of threads which are executed concurrently; shared memory is accessible by any thread inside of a block. Cards with a compute capability of 3.0 can have up to 1024 threads per block. Older versions support up to 512 threads per block. Similar to threads, blocks also have a unique identification number. A grid is a collection of blocks, stored in an array, with up to three dimensions. A grid execution reads to and writes from global memory as a set. CUDA supports up to 65,535 blocks per grid and has a built in thread scheduler which automates most of the thread execution. The programmer is only responsible for declaring a constant value for threads per block and blocks per grid. Each kernel can have its own values for threads per block and blocks per grid.

2.3 CUDA Limitations

The CUDA architecture is constantly improving and adding features. Graphics cards with compute capability 1.X suffered from two major drawbacks which effect convergence to and accuracy of the solution. The first drawback, which is believed to have caused convergence issues, was the lack of support for standard IEEE 754 rounding. Chop and round-to-nearest even were originally the only available rounding modes, however starting with compute capability 2.0 (Fermi) nearly full IEEE 754 rounding is supported. The second major drawback is that cards with compute capability of 1.2 and below only support single precision floating-point numbers. There was limited support in 1.3 capable cards. Atomics and full double precision support was added starting with 2.0 compute capable cards [3].

CHAPTER 3

DESCRIPTION OF GRIDS

A grid, sometimes referred to as a mesh, is a discretized representation of an area or volume. Grids are used when exact analytical solutions are impossible or too difficult and approximate solutions must be calculated over discrete regions.

3.1 Grid Generation Algorithm

The unstructured mesh generation method outlined below by Marcum and Weatherill [4] was the basis for the program AFLR2D written by Prof. David Marcum. His program was used to generate the grids implemented in my code.

1) Generate a boundary surface grid for the given configuration.

2) Obtain a valid triangulation of the boundary points and recover all boundary surfaces.

3) Assign a point distribution function to each initial boundary point.

4) Initialize the data structure. For each element, the element points, element neighbors, and an active/off flag are saved. Initially, all elements are made active.

5) Turn off each active element which satisfies the point distribution function.

6) Create a new point for each active element.

7) For each new point distribution function for the new point from the containing element.

8) Interpolate the point distribution function for the new point from the containing element.

9) Reject new points that are too close to an existing point or another new point.

10) Directly insert each accepted new point by subdividing the containing element. Make all new element active.

11) For each active element, compare the reconnection criterion for all allowable connectivities with adjacent elements and reconnect using the most optimal connectivity.

12) Repeat the local reconnection process, step 11, until no elements are reconnected.

13) Repeat the field point generation process, steps 5-12, until no new points are created or accepted.

14) Smooth the coordinates of the field grid points. For three dimensions, smoothing is skipped on the first pass of quality improvement, steps 14-16.

15) Repeat the iterative local reconnection process, steps 11 and 12, with all elements activated.

16) Eliminate isolated sliver elements on the boundary surfaces. This step is used only for three dimensions.

17) For three dimensions, repeat the quality improvement process, steps 14-16, for a total of three passes.

The process of grid generation starts with a grid comprised of an outer and usually inner boundary surfaces. An initial sparsely populated grid is usually comprised of low quality, high-aspect ratio triangles. Based off the local density of nodes along the boundaries, a distribution function can be calculated on the estimated distribution of the initial nodes inside the grid.

Elements carry a label which designates if they satisfy a specific set of criterion. If the element has not met the criterion then it is labeled as active, which indicates that its properties can be changed. Conversely, if an element has met the criterion then it is labeled as off and its properties become static. If for some reason the properties of an off element needed to be modified then the element's label is changed to active. Once an off element is completely surrounded by off elements it can no longer be modified.

New points are created off of element faces in what is called advancing-front point placement. A new point is placed in the area normal to an active element. The distance chosen away from the active element is specified so that the resulting new triangle is approximately equilateral. In some cases two adjacent elements may be selected with only a single point being created. This single point is the average of what would have been each elements derived point. Each set of new points must pass all of the rejection tests which include checking to see if the new point is too close to its parent element's points as well as other newly created points. Instead of outright rejecting a new point too close in proximity to another, the two points can be merged with an average of the two positions. In addition to advancing-front point placement, new points may be placed inside of existing cells. These cells are then subdivided into three new cells.

To assure grid quality, the code uses the Lawson edge-swapping algorithm for local reconnection which is repeated until convergence or until a set number of iterations is reached. The final step in the grid creation process is most often some form of coordinate smoothing. Laplacian-like smoothing is a simple, effective, and common method used for final quality improvement. Laplacian-like smoothing is accomplished by taking the average x and y difference between the point of interest and its surround element centroids. A fixed fraction of the average is then added to the original coordinate value for each pass over the point. The fraction for interior points for this method was 0.5 and 0.25 for points adjacent to boundaries.

3.2 Generated Grids

Dr. Dennis provided me with six grids to be used in my research. Each of the six grids is square and is composed of linear triangular elements. The grids vary in size from 107 nodes to as much as 1,694,081 nodes and average between 6.27 and 6.99 non-zeros per row in the

FEM stiffness matrix. The table and figure below give the basic properties of the six grids used for my research.

Table 3.1 Grid Information

Grid	Nodes	Cells	Non-Zeros	Non-Zeros per Row
1	107	176	671	6.27
2	452	826	3006	6.65
3	46,068	91,338	320,878	6.97
4	106,377	211,456	741,921	6.98
5	424,129	845,824	2,964,033	6.99
6	1,694,081	3,383,296	11,848,833	6.99

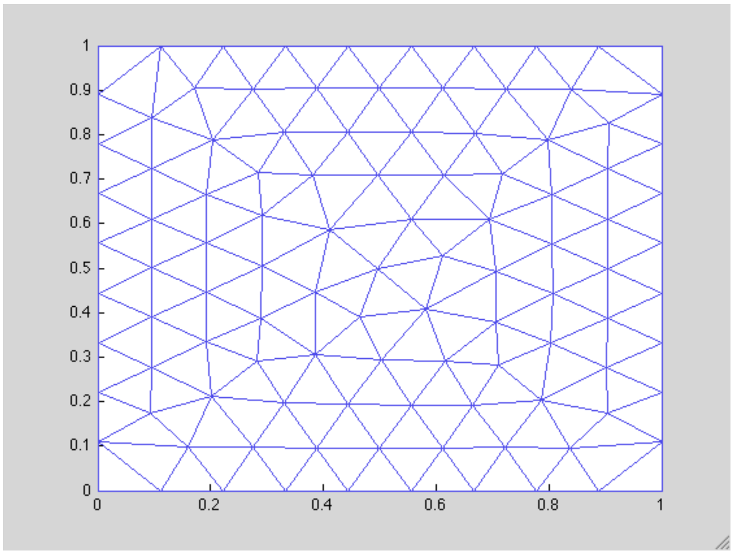

Figure 3.1 Grid 1

CHAPTER 4

SPARSE MATRIX STORAGE

In a typical system of linear equation problems each value in the matrix is explicitly stored in what is called a dense matrix. Dense matrix storage is simple and effective when non-zero values greatly outnumber zero values. However, numerical methods typically contain limited connectivity among nodes, leading to a large amount of zeros per row. The grids in my research never had more than 8 non-zeros per row despite the number of rows (N) reaching up 1.7 million. This leads to highly inefficient storage and unnecessary multiplication and addition of zeros. The logical approach is to develop a way to only store non-zeros and to adjust the code to be able to use the new storage system. Thankfully, several sparse matrix storage schemes are available. The organization and potential patterns in the sparse matrix dictate which storage method is best suited for the problem. If for example a problem existed where both row and column values were given out of ascending order, one could use the Coordinate format (COO). In the COO format both row and column indices are explicitly stored along with non-zero values. The Compressed Sparse Row (CSR) format uses the same data and column arrays as the COO format, but uses a compressed variation of the row array.

For its simplicity and ease of implementation I chose to use the CSR format. The CSR format is a row-major storage scheme, meaning that matrix values are grouped with other values from the same row. In my code, matrix values within a row group are not necessarily in ascending column order. This is acceptable and of little consequence as the CSR storage format can easily accommodate the disorganization. In the CSR format three 1-Dimensional arrays are needed to store the matrix. Non-zeros from the matrix are stored in a data array (data) of length equal to the number of non-zeros, nvals. The column indices of the non-zeros are stored in the second array (col) also of length nvals. A third array (ptr) with a length of N+1 stores a pointer to the first entry for each row, with the last value in the array being the total number of non-zeros in the data array. The number of non-zeros in the Nth row can be found by using ptr[N+1] – ptr[N].

$$A = \begin{bmatrix} 4 & 0 & 0 & 2 \\ 0 & 6 & 7 & 9 \\ 1 & 2 & 0 & 0 \\ 5 & 0 & 4 & 0 \end{bmatrix}$$

$$data = [4\ 2\ 6\ 7\ 9\ 1\ 2\ 5\ 4]$$
$$col = [0\ 3\ 1\ 2\ 3\ 0\ 1\ 0\ 2]$$
$$ptr = [0\ 2\ 5\ 7\ 9]$$

The memory required for storage of a dense matrix is proportional to $\mathcal{O}(N^2)$, while the memory storage necessary for the CSR format is proportional to approximately $\mathcal{O}(15N)$ for the grids used. There is a memory storage benefit for CSR format over dense format once the number of rows is greater than 15. This means that for virtually all practical purposes there will be a benefit in using CSR format. The most computationally intensive procedure in the solver is the Matrix-Vector multiplication. For dense matrices $\mathcal{O}(2N^2)$ computations are needed. For my grids, using CSR format only requires $\mathcal{O}(14N)$ computations. This means dense format requires $\mathcal{O}(N/7)$ times more computations. It is quickly becoming obvious that the slight increase in complexity from using sparse matrix format greatly decreases the memory and computation times needed for matrix work.

CHAPTER 5

ITERATIVE LINEAR SOLVER

There are two ways to solve a system of linear equations. Direct solvers, such as Gaussian elimination, are based on easy to follow steps and can be easily calculated by hand for small number of linear equations. They give answers which are exact as long as computational errors are not introduced. However, the number of calculations to solve a system with N number of equations is on the order of $\mathcal{O}(N^3)$. This means that rate of increase in computation times far outpaces the rate of increase of a linear system of equations. An iterative method is another way to solve a system of linear equations. Depending on the method of storage, iterative method used, and the degree of accuracy desired the number of calculations required could be much less than what is required with Gaussian elimination.

Iterative solvers vary greatly from method to method, but all work on the general principle of taking an initial guess to the problem and iterating the interim solution through a process until the solution is deemed to be accurate enough. There are two types of iterative solvers, the stationary and nonstationary methods. Stationary methods such as the Jacobi method are easy to understand but require a large number of iterations to reach convergence. Nonstationary methods such as the BiCGSTAB method are not as straightforward in concept but converge at a much faster rate [18].

Several iterative solvers were considered in the early phases of programing. The Jacobi method was initially considered because it was easy to understand and program; however initial studies on its implementation showed it to have difficulty reaching convergence. The next method investigated was the Conjugate Gradient Method which had much better convergence characteristics compared to the Jacobi method but would fail to converge for Grid 3 and larger. Finally, I picked the biconjugate gradient stabilized method. Convergence was much less of a problem compared to other methods. Because of its rate of convergence and stability, the BiCGSTAB method is used in many commercial computational solvers, such as Fluent.

The biconjugate gradient stabilized method (BiCGSTAB) is an iterative solver developed by van der Vorst to solve a nonsymmetrical system of linear equations. BiCGSTAB seeks to minimize the residual, which leads to a smoother convergence compared to many other conjugate gradient methods. Van der Vorst [6] outlined the unpreconditioned BiCGSTAB algorithm as seen below:

```
1. $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
2. Choose an arbitrary vector $\hat{\boldsymbol{r}}_0$ such that $(\hat{\boldsymbol{r}}_0, \boldsymbol{r}_0) \neq 0$, e.g., $\hat{\boldsymbol{r}}_0 = \boldsymbol{r}_0$
3. $\rho_0 = \alpha = \omega_0 = 1$
4. $\boldsymbol{v}_0 = \boldsymbol{p}_0 = \boldsymbol{0}$
5. For $i = 1, 2, 3, \ldots$
        a. $\rho_i = (\hat{\boldsymbol{r}}_0, \boldsymbol{r}_{i-1})$
        b. $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$
        c. $\boldsymbol{p}_i = \boldsymbol{r}_{i-1} + \beta(\boldsymbol{p}_{i-1} - \omega_{i-1}\boldsymbol{v}_{i-1})$
        d. $\boldsymbol{v}_i = \boldsymbol{A}\boldsymbol{p}_i$
        e. $\alpha = \rho_i/(\hat{\boldsymbol{r}}_0, \boldsymbol{v}_i)$
        f. $\boldsymbol{s} = \boldsymbol{r}_{i-1} - \alpha\boldsymbol{v}_i$
        g. $\boldsymbol{t} = \boldsymbol{A}\boldsymbol{s}$
        h. $\omega_i = (\boldsymbol{t}, \boldsymbol{s})/(\boldsymbol{t}, \boldsymbol{t})$
        i. $\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \alpha\boldsymbol{p}_i + \omega_i\boldsymbol{s}$
        j. If $\boldsymbol{x}_i$ is accurate enough then quit
        k. $\boldsymbol{r}_i = \boldsymbol{s} - \omega_i\boldsymbol{t}$
```

CHAPTER 6

KERNELS

6.1 Matrix-Vector Multiplication

Matrix-Vector multiplication is a frequently used linear algebra operation and is an important aspect of most iterative methods. Matrix-vector multiplication can be viewed as a series of dot products, with each entry being the dot product between a row in the matrix with the vector. The resulting scalar of each dot product is placed into the corresponding position in the output vector. The Matrix-Vector multiplication is the most computationally intensive portion of my FEM solver. It is used 2 times during each solver cycle. However, a third could be used for using $residual = Ax^k - b$ for the exit condition.

The matrix-vector multiplication used in Carrigan et al [7] utilized a single kernel. The approach is very similar to the CSR kernel introduced in Bell and Garland. However, the matrix used in their computations had a high number of non-zeros per row. Because of a higher number of non-zeros per row, they were able to assign a warp of 32 threads per each row which minimizes thread idle time. It was decided in Carrigan et al [7] that the single kernel approach would be easier to explain and was hence chosen. For this paper, the more computationally efficient method of breaking the matrix-vector multiplication into two kernels was deemed to be appropriate.

The following code segment is a representation of the single kernel approach:

```
while (tid < N) {

y[tid] = 0;
loc = ptr[tid];
len = ptr[tid+1] - loc;

for (i=0;i<len;i++)
        y[tid] += data[loc+i] * x[col[loc+i]];
tid += blockDim.x * gridDim.x;
}
```

In my paper, I will show that the CSR method benefits from splitting the algorithm into two separate kernels. For the two-kernel approach the first kernel executes a single thread for each multiplication for a total of nvals threads. The resulting products are stored on the device memory in an array of length nvals. The second kernel launches a single thread for each row in the matrix for a total of N threads. Each thread serially adds all the output values from the first kernel, which corresponds to the given row.

The following code segment is a representation of the two kernel approach:

Kernel 1:

```
while (tid < nz) {

        c[tid]= data[tid] * x[col[tid]];
tid += blockDim.x * gridDim.x;

}
```

Where nz is equal to the total number of non-zeros in the sparse matrix.

Kernel 2:

```
while (tid < N){

loc=ptr[tid];
len=ptr[tid+1] - loc;
for (i=0;i<len;i++)
        y[tid] += c[loc+i];

tid += blockDim.x * gridDim.x;

}
```

6.2 Dot Product

Dot product, like the matrix-vector multiplication, is a very commonly used mathematical tool. The inputs are two equal length vectors and the output is a single scalar, which is the summation of the product between corresponding entries in each vector. The dot product procedure is used five times for each solver iteration in my code. Initial versions of my code called for the dot product to be handled by the CPU, as it was believed at the time that parallelizing the dot product procedure would be too inefficient. It was quickly discovered that transferring the vectors back and forth between host and device RAM was taking a significant part of the total runtime of the code. Further research shows that others such as Bolz et al. [8] effectively utilized the GPU for both matrix-vector and dot product.

The procedure for dot product computations outlined in Sanders and Kandrot [9] was found to be fast, accurate, easily scalable and fairly easy to understand. Unlike the other kernels outlined in this paper, the single Sanders and Kandrot kernel is separated into two separate parts. The kernel makes use of CUDA's caching system's ability to schedule threads in blocks and performs partial sum reduction in parallel. The first portion of the kernel, seen below, performs the elemental multiplication between the two vectors.

```
while (tid < N) {

temp += a[tid] * b[tid];
tid += blockDim.x * gridDim.x;

   }
cache[cacheIndex] = temp;
```

The kernel calculations are conducted with blocks in mind with each block execution producing a threadsPerBlock number of products. The product from each thread is stored into shared memory until the next step.

Once all the threads in the block have finished executing, a "folding" sum reduction is performed with each "fold", reducing the size of the array in half. The partial sum reduction code is as follows:

```
while (i != 0) {

if (cacheIndex < i)
   cache[cacheIndex] += cache[cacheIndex + i];

 i /= 2;
}
```

Once only a single value remains, that value is stored in the output array of length blockPerGrid. This continues with new blocks until the output array is filled. At this point the dot product procedure is not complete. A small set of values to be summed still exists, but finishing the final summations of the GPU would make poor usage of available CUDA cores. In fact, to finish the sum on the GPU is slightly slower than transferring the remaining values to the CPU for final sum reduction.

6.3 Other Kernels

The total number of floating point operations per each loop of the BiCGSTAB method is on the order of 46N. The dot product and Matrix-vector operations account for 34N of the total 46N operations. Assuming there is a 10x speedup for both dot product and Matrix-vector kernels and ignoring memory transfer times, the overall speedup of the code would only be about 3x. Even with an infinite speedup of the two kernels, only a 3.8x speedup would be realized. Taking this into account makes it obvious that every possible part of the BiCGSTAB method needs to be ported to the GPU using CUDA. In the BiCGSTAB method is also comprised of vector-scalar multiplication, vector-vector addition and therefore, addition kernels were needed for these operations. The kernels for these additional operations are trivial and virtually identical to the equivalent C code. Because of their simple nature and trivial memory access, only one thread is needed for each value in the array for a total number of N threads.

CHAPTER 7

ANALYSIS OF RESULTS

7.1 Application to FEM

The Finite Element Method is a mathematical tool used to approximate the solution to a differential equation over a specified set of coordinates called nodes. The method was originally developed to analyze structures but has since spread to fluid mechanics and electromagnetism.

The equations used to approximate the solution between nodes are called shape functions. Shape functions are general equations which can be as simple as scalars or as complex as multi-ordered polynomials. Higher ordered polynomials give more accurate solutions but require more nodes within a cell. Hence, they require a larger stiffness matrix and will require more calculations and time to calculate the solution. For my algorithm, I chose linear equations over triangular cells, which are defined by the shape function below.

$$N_i = \frac{a_i+b_ix+c_iy}{2A} \text{ [5], for } i = 1,2,3.$$

Manipulation of the 2-D Poisson's equation $k\nabla^2\phi$=f(x,y) results in the local stiffness matrix equation

$$k_{ij} = \int_\Omega \left(k\frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial x} + k\frac{\partial N_i}{\partial y}\frac{\partial N_j}{\partial y}\right) d\Omega \text{ [5].}$$

Plugging in the appropriate derivatives and integrating the above equation over the general area results in the final general local stiffness matrix equation

$$k_{ij} = \frac{t}{4A}\left(kb_ib_j + kc_ic_j\right) \text{ [5].}$$

Where t is the thickness of the plate, A is the area of the triangular cell, k is the thermal conductivity of the plate, b is an array of the differences in y positions between the nodes in the triangle, and c is an array of the differences in the x positions between the nodes in the triangle.

The coordinates for each node in the triangular cell are plugged into the local stiffness matrix equation and each value is added to the appropriate location in the global stiffness matrix using nodal numbering.

FEM, like differential equations, requires boundary or initial conditions in order to be solved. The grid file contains information about which nodes lay on which boundary and assigns them a marker. Whenever a node lays on a boundary, the algorithm zeros out the row associated with that node, then sets the value on the diagonal to one.

After the global stiffness matrix and global forcing vector are assembled, the only remaining step is solve the system of linear equations. The method used to solve the system is up to the coder and user to determine, but for my code I chose to use the BiCGSTAB method.

Initially, my CUDA code used single precision float point numbers but I discovered that for grids 3 and larger that BiCGSTAB either had problems converging smoothly or would simply diverge. I believe this may be an unavoidable consequence stemming from issues I discussed in chapter 8, as well as BiCGSTAB's reliance on accurate calculations. I determined the simplest approach to avoid these convergence issues was to switch to the more accurate double precision floating point numbers.

7.2 Results

This paper mainly focuses on minimizing the time in which it takes to reach a steady state solution but does not neglect to verify that the solution is correct. To calculate the accuracy of the CUDA output, I derived the analytical series as follows:

$$T = 10 + \sum_{m=1}^{5}\sum_{n=1}^{5} \frac{40}{mn\pi^4(m^2+n^2)}(1 - \cos m\pi - \cos n\pi + \cos m\pi \cos n\pi)\sin m\pi x \sin n\pi y$$

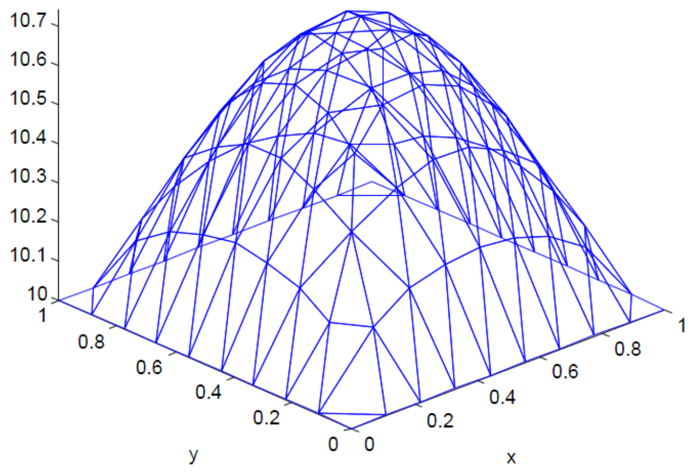

Figure 7.1 Analytical Solution on Grid 1

I used the above equation in MATLAB and evaluated it at each node in the FEM grid. The figure below is a 3-D visualization of Grid 1 with the Z-axis representing percent error between the analytical and CUDA solution. An ℓ_2 norm of the vector r with a value of 1e-6 was used for this example, resulting in a peak percentage error of only 0.103%.

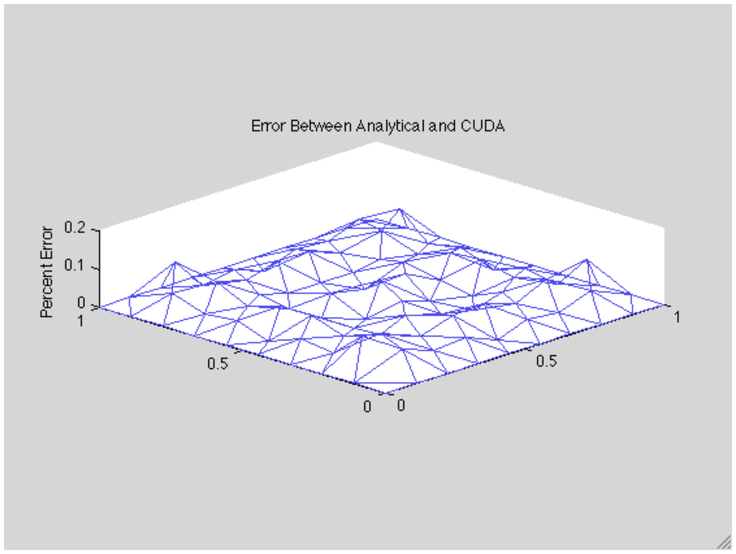

Figure 7.2 Error Between Analytical and CUDA

7.3 Computational Performance

Computational performance enhancement was the main goal of my research. Every choice made had to be scrutinized for its performance impact from every aspect including memory storage, memory transfer, algorithm efficiency, and kernel approach. Each kernel is subject to the strength and weaknesses of the architecture of the GPU. For instance, the sparse matrix-vector multiplication kernels have the advantage of a larger data set but suffer from poor memory coalescence and lack of utilization of the memory cache. However, the dot-product kernel is quite different in that memory cache is highly utilized, but it suffers from a relatively small data set and requires memory transfer between the host and GPU RAM. I found the number of nodes in a grid to be a large factor in the performance speedup. In fact, for the smallest grid no performance speedup was realized for the whole code. Below are plots of grid size versus performance speedup for various portions of the code.

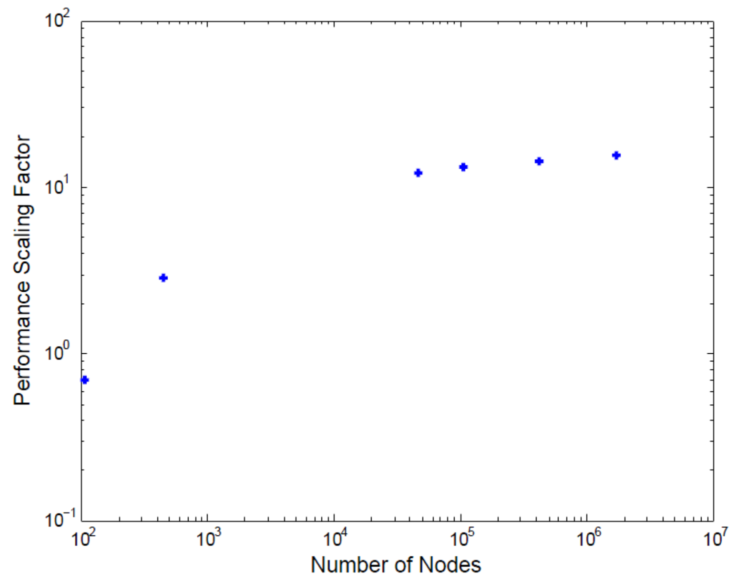

Figure 7.3 SpMV Speedup as a Function of Grid Size

The plot above only displays the speedup of the sparse matrix-vector multiplication kernel relative to a single core. It can be seen in the grids with 46,000 nodes or more that diminishing returns is very prevalent. Hence, it can be concluded that at some point increasing grid size will not improve performance speedup.

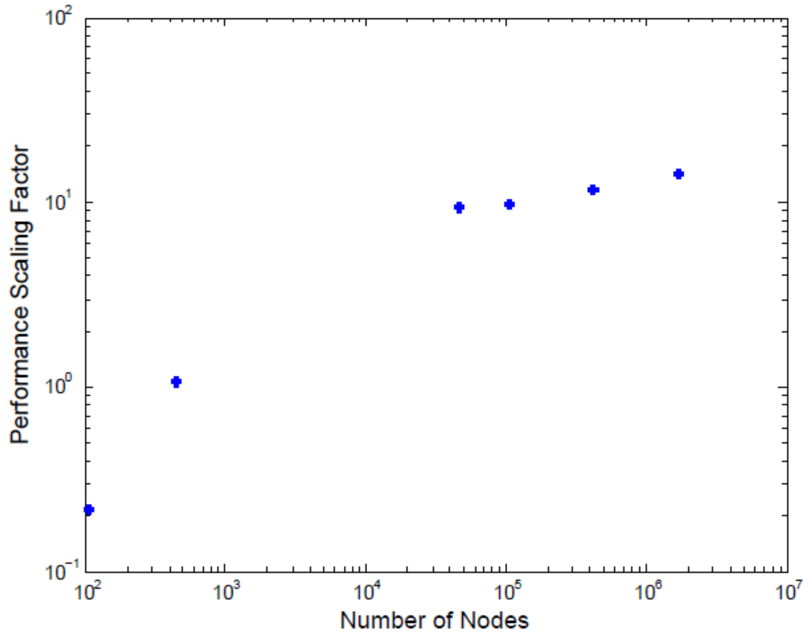

Figure 7.4 BiCGSTAB Speedup as a Function of Grid Size

The plot above includes the entire iterative portion of the code and takes into consideration all the kernels used in the code. Diminishing returns is still present, however, it is not as pronounced as it was for the SpMV multiplication kernel.

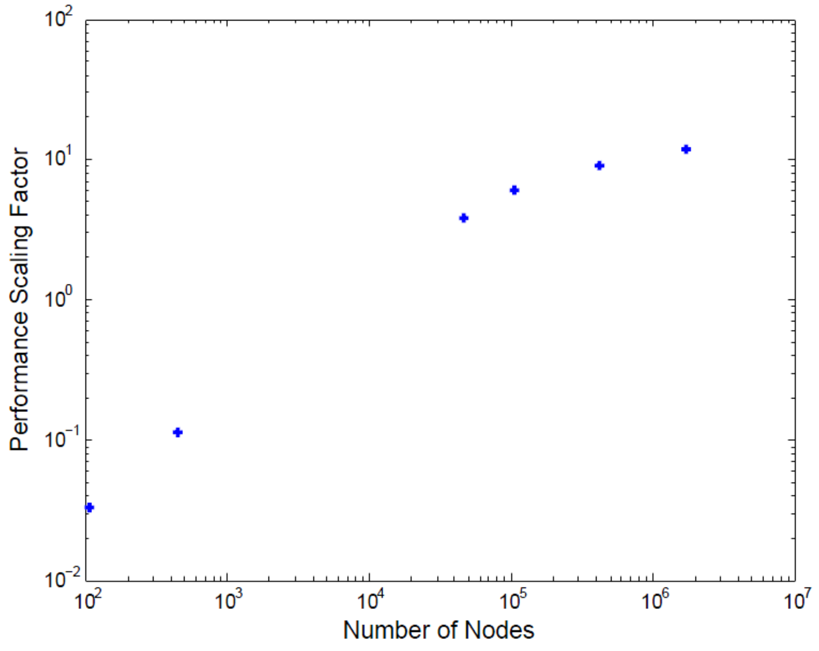

Figure 7.5 Entire Code Speedup as a Function of Grid Size

The plot above illustrates the performance speedup for the entire code including the time needed to read the grid file into host RAM. It can be seen that now both the two smallest grids fail to realize a performance speedup and are in fact much slower than the equivalent CPU based code. This is to be expected as memory transfer times for smaller grids from host to GPU RAM accounts for a significant portion of the total run time.

CHAPTER 8

OPTIMIZATION

In an attempt to better improve the computational acceleration, I performed an optimization study. The blocksPerGrid and threadsPerBlock variables were used as the design variables for the optimization. While computations are easily parallelized, each step in the BiCGSTAB method more or less must be accomplished serially. This allows each kernel's optimization to be independently conducted, meaning only two variables need to be investigated at a time. Before picking an optimization technique I examined the sensitivity and general behavior and discovered that execution time was very sensitive to both variables and resulted in high amplitude oscillations. According to Deb [10], iterative methods typically rely on sensitivity analysis to determine if an adequate minimization has been achieved. With the large amplitude high frequency time values I sampled, I determined the best optimization method would be a "brute force" approach. The possible total values for threadsPerBlock and blocksPerGrid are 1024 and 65,535 respectively. This leads to over 65 million possible combinations of the two variables. Grids 6 and 5 take approximately 31 and 4 seconds respectively to run the BiCGSTAB portion each time. This would take too long to run so I only studied grid 4. At approximately 1 second for each run this would result in over 2 years just to optimize just grid 4. Thankfully, a few assumptions can be made to lower the number of trials. First, I assumed blocksPerGrid would be between 1 and 1024. Second, I limited each variable to only be multiples of 8. This is a fair assumption to make in regards to blocksPerGrid because the location of the peaks and troughs are multiples of 8. With the assumptions made, the time to run through the remaining combinations was between 4.5 and 5 hours.

Table 8.1 Optimization Results

Kernel	blocksPerGrid	threadsPerBlock
DataProd	736	384
DataAdd	40	64
Dot	736	384
SRCalc	736	128
PCalc	128	272
XCalc	296	192

Comparing the values obtained from the optimization with an arbitrary value of 512 for both threadsPerBlock and BlocksPerGrid yields a 18.4% decrease in run time for grid 4.

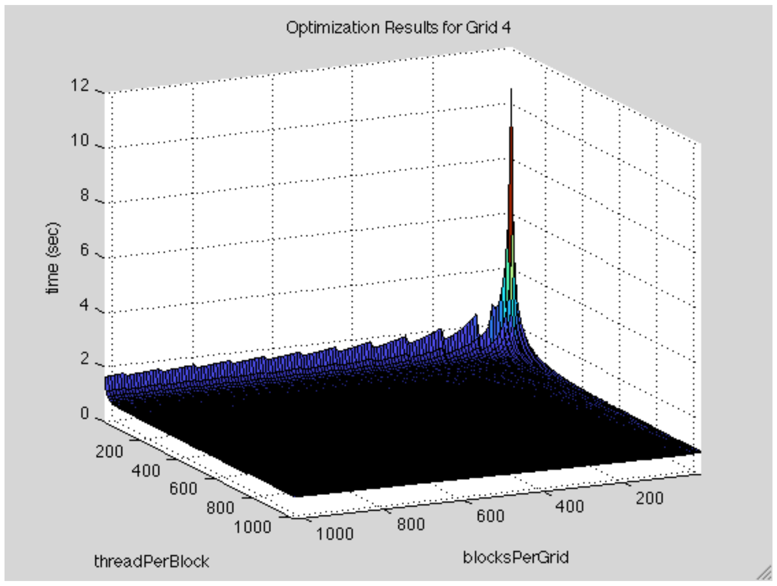

Figure 8.1 Optimization Results, Full Range

Above is a surf plot of the optimization results for the DataProd kernel. It can be seen that for small values of blocksPerGrid and threadPerBlock that the GPU runs very inefficiently leading to very long run times. A saw tooth pattern can be seen along blocksPerGrid for low values of threadsPerBlock. Times appear smooth on this scale for most of the plot. As shown in the plot below, this is only because of the high time values near the origin.

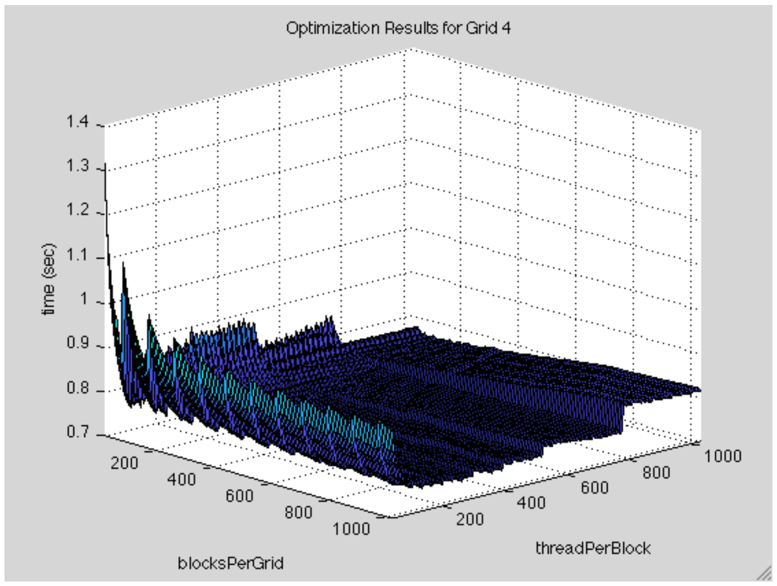

Figure 8.2 Optimization Results, Medium Range

The above surf plot leaves out the first three values of both blocksPerGrid and threadsPerBlock. This allows for more detail to be seen, where before it appeared to be relatively flat. For larger values of blocksPerGrid, it can be seen that results increase in a step manner as you move along threadsPerBlock. The plot below limits the field of interest even further.

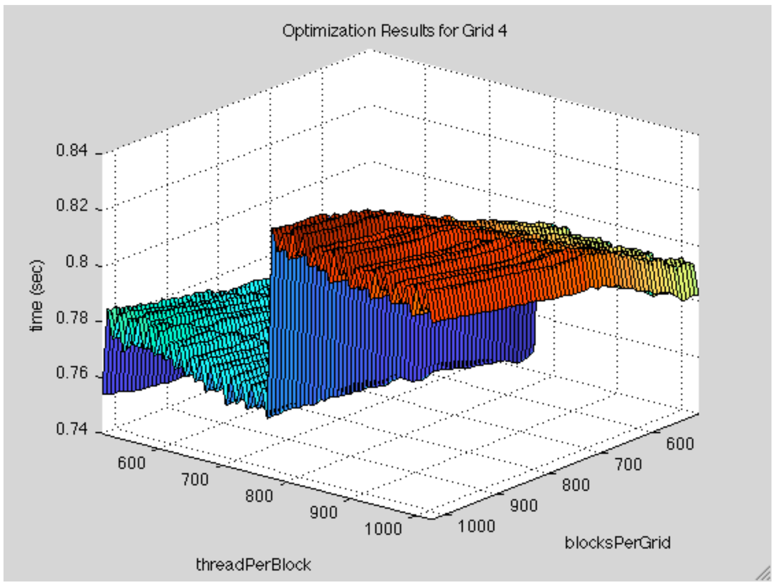

Figure 8.3 Optimization Results, Narrow Range

The above plot only includes time values for when the two variables are above 512. Both the stepping motion and noise in the results are better visualized. However, while results along threadsPerBlock show a large amount of noise and discontinuity, results along blocksPerGrid are relatively smooth. Below are cross-section plots intersecting the point of minimum run time.

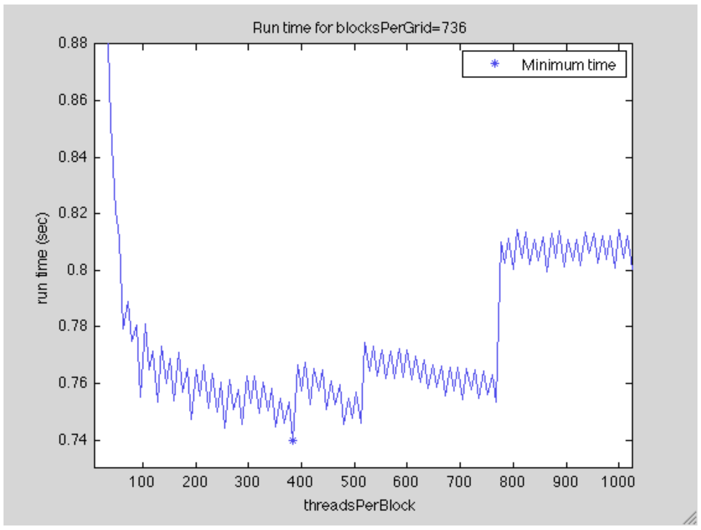

Figure 8.4 Optimization Results at blocksPerGrid = 736

The above plot shows threadsPerBlock versus run time for a fixed values of blocksPergrid = 736. The results for this plot are very noisy and contain several discontinuity jumps, making an iterative optimization study hard to conduct. ThreadsPerBlock values surrounding the minimum point only vary by about 3.5% at most.

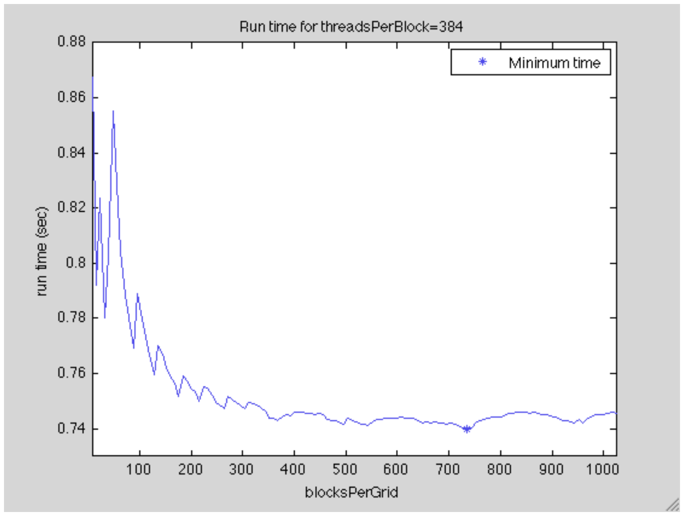

Figure 8.5 Optimization Results at threadsPerBlock = 384

The above plot shows blocksPerGrid versus run time for a fixed threadsPerBlock. It can be seen that for values of blocksPerGrid over 400 the results are relatively smooth. In fact, the runtimes for values of blocksPerGrid between 400 and 1024 are at most about 0.8% greater than that of the optimized minimum. Naively, one might come to the conclusion that only an optimization for threadsPerBlock for a fix blocksPerGrid would be necessary; however, the best threadsPerBlock changes for each fixed value of blocksPerGrid, forcing both variables to be optimized simultaneously.

CHAPTER 9

CONVERGENCE ANALYSIS

Single and double precision floating-point numbers are storage methods which computers use to store real non-integer numbers. Double precision holds 64 bits of information while single precision holds 32 bits. They both are not without their disadvantages. Careful consideration must be taken to make sure a mathematical operation between two floating-point numbers accurately represents the correct value. For instance, if you are summing a large series of values such as in a dot-product, values of small magnitude lose significant digits when added to the rolling sum of a much larger magnitude. Instead of serially adding the values you can get a more accurate result if a reduction is used. This way the values added during each step are relatively close to one another in value. Also, using double precision instead of single precision will greatly improve the serial addition results but with double precision comes greater storage and computational requirements. Because of the limitations of single precision, grids larger than grid 3 diverge with the BiCGSTAB method but converge with double precision. Below are plots of ℓ_2 norm of the residual versus number of iterations for grids 1, 3 and 5.

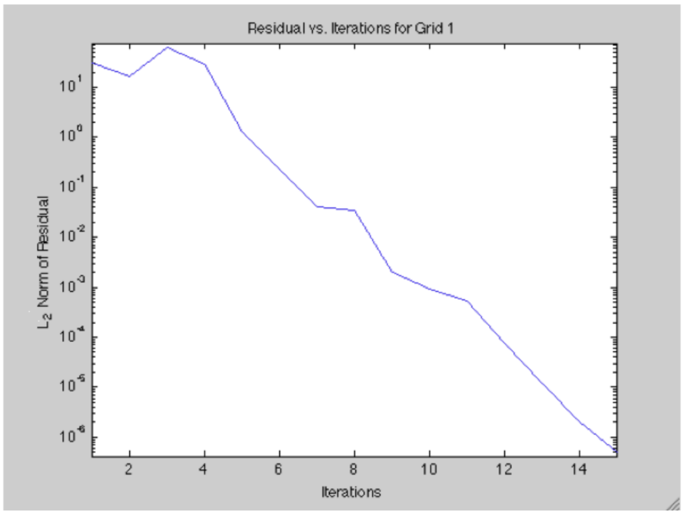

Figure 9.1 Grid 1 Convergence

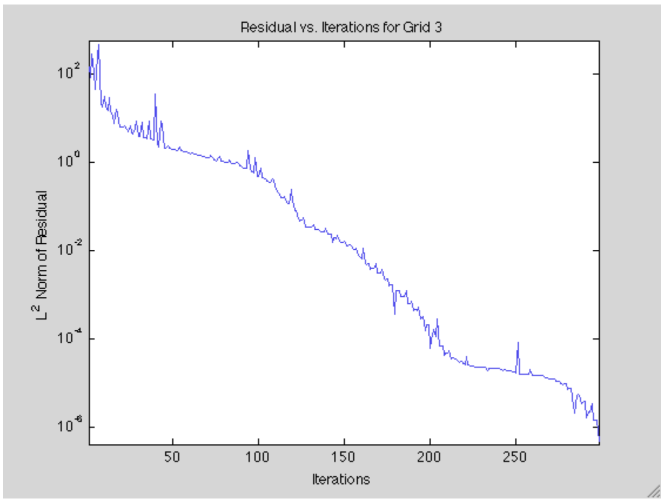

Figure 9.2 Grid 3 Convergence

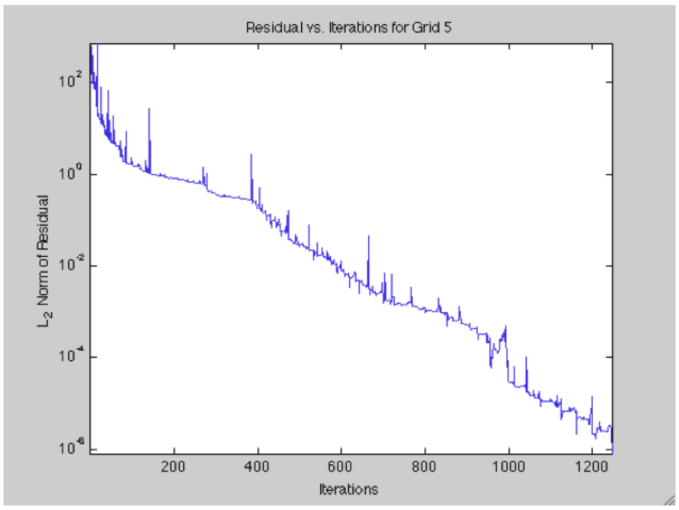

Figure 9.3 Grid 5 Convergence

There are several methods and combinations of methods which can be used to calculate the exit condition. Two popular methods I compared were ℓ_2 and ℓ_∞ norms [11].

$$\|x\|_2 = \left(\sum_{i=1}^{n} |x_i^2|\right)^{1/2} \qquad \ell_2\ norm$$

$$\|x\|_\infty = \max_{1\le i\le n} |x_i| \qquad \ell_\infty\ norm$$

For the graphs below I used the first equation on the residual vector from line k of the BiCGSTAB method and I used the second equation on the vector $v = A\tilde{x} - b$.

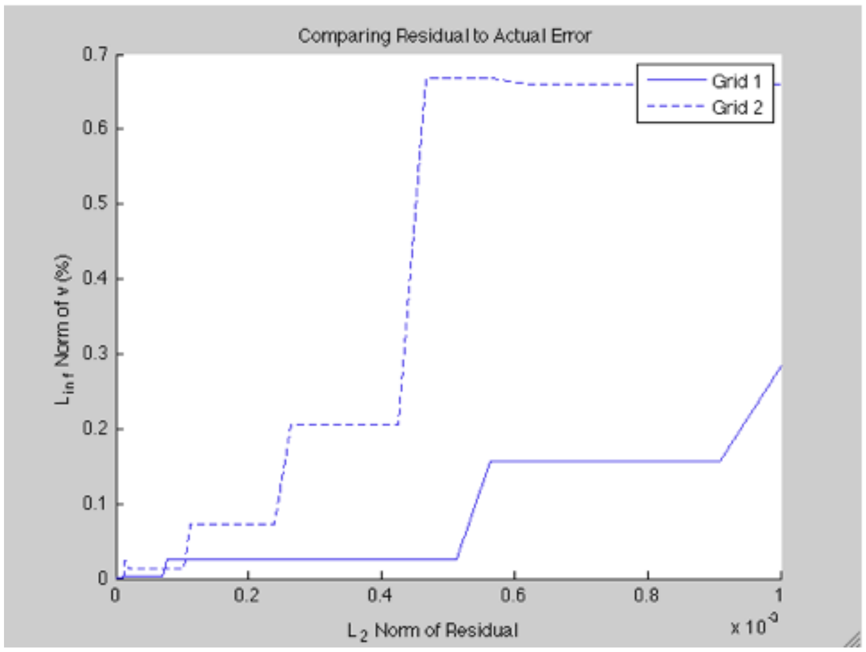

Figure 9.4 Comparing Norms for Grids 1 and 2

It can be seen from the first two grids that you can expect an error of less than 0.7% for any exit condition smaller than 10^{-3}. The stepping of the data is a result of multiple exit conditions being satisfied for the same number of iterations. There is a general trend that the larger the exit condition value, the greater the error in the approximation.

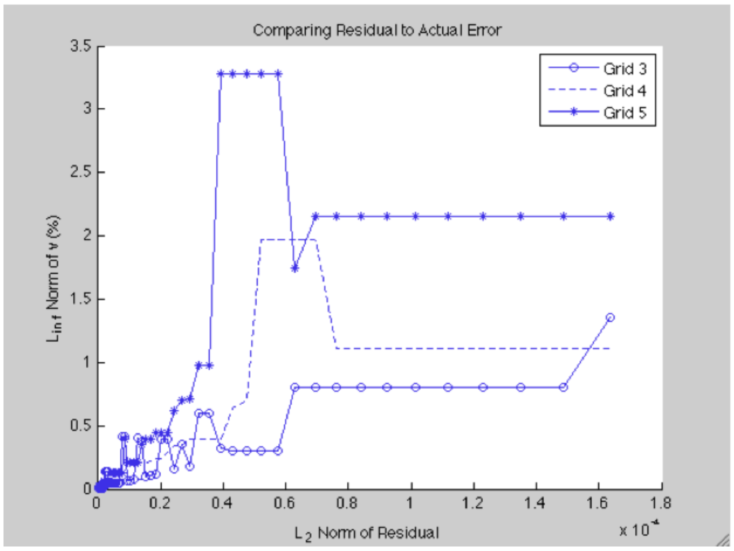

Figure 9.5 Comparing Norms for Grids 3, 4 and 5

For grids 3, 4 and 5 much more consideration must be taken when choosing the exit condition value for ℓ_2 norm. As long as the ℓ_2 norm chosen is below about $0.4*10^{-4}$, it can be expected to have an ℓ_∞value of less than 1%. For grids with a large number of nodes it may be more meaningful and beneficial to use ℓ_∞ for the exit condition. When grids become very large the values of the residual must be very small in order to satisfy a small ℓ_2 norm value. This can potentially cause issues with single precision.

CHAPTER 10

CONCLUSIONS AND RECOMMENDATIONS

The results and research presented in this paper demonstrate the benefits and disadvantages to using GPUs for general-purpose programing, specifically when solving finite element problems. Galerkin discretization, sparse matrix storage, iterative solver, and massively parallel computing are the methods I used in this project to solve steady-state Poisson's equation with equal boundary conditions and a uniform source term over a unit plate. I have shown that an overall speedup of 11.8 times can be achieved for sparse matrix-vector multiplication without any special consideration of memory coalescing or cache utilization. Furthermore, once all the remaining portions of BiCGSTAB have been ported to CUDA, it is shown to increase speedup performance up to 14.4 times. The whole code still experiences a speedup of 11.8 times for grid 6 even when loading the grid file, local matrix calculations, sparse matrix setup, boundary enforcement, and memory transfer to the GPU are taken into consideration.

The purpose of this paper is to show the potential of the integration of GPUs into more complex fluid dynamic and structural analysis codes. For instance, turbulent flow problems can now afford to have finer grids with smaller time steps to better capture realistic flow characteristics without the need of expensive domain decomposition [12].

According to Bell and Garland [13,14], the CSR storage method suffers from poor memory coalescing and leads to poor utilization of the GPU. Different sparse storage systems could be expected to allow for better use of cache as well as improved memory coalescing and bandwidth. Cache-oblivious methods such as Zig-zag CRS and Bi-directional Incremental CSR, as described by Yzelman and Bisseling [15,16], perform well despite the lack of cache optimization and may prove to be efficient for the particular sparse nature of my grids.

While inspecting figure 4.1 I noticed that at each corner there is a cell with all three of its nodes laying on a boundary. This causes a slight but noticeable error at each corner. I would recommend that the grid generation software have an exception built in so that each corner is split into two separate cells.

REFERENCES

[1] NVIDIA, NVIDIA CUDA C Programming Guide. Nov. 2011. 31 Mar. 2012 <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf>.

[2] Kirk, David, and Wen-mei Hwu. Programming Massively Parallel Processors. Burlington: Morgan Kaufmann Publishers, 2010.

[3] NVIDIA, NVIDIA Fermi Compute Architecture Whitepaper. 2009. 31 Mar. 2012 <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>.

[4] Marcum, David and Nigel Weatherill. "Unstructured Grid Generation Using Iterative Point Insertion and Local Reconnection." AIAA Journal 33.9 (1995): 1619-1625.

[5] Huebner, Kenneth, et al. The Finite Element Method for Engineers, John Wiley & Sons, 1995.

[6] van der Vorst, Hendrik. "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems." SIAM 13.2 (1992): 631-644.

[7] Carrigan, Travis, et al. "Using GPU-Based Computing to Solve Large Sparse Systems of Linear Equations." ASME DETC2011-48452 (2011).

[8] Bolz, Jeff, et al. "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid." ACM SIGGRAPH, (2003): 917-924.

[9] Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming. Ann Arbor: Addison-Wesley, 2011.

[10] Kalyanmoy, Deb. Optimization for Engineering Design Algorithms and Examples. New Delhi: Prentice-Hall of India, 2004.

[11] Cheney, Ward, and David Kincaid. Numerical Mathematics and Computing. Belmont: Brooks Cole, 2007.

[12] Corrigan, Andrew, et al. "Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware." 19th AIAA Computational Fluid Dynamics Conference AIAA-2009-4001 (2009).

[13] Bell, Nathan, and Michael Garland. "Efficient Sparse Matrix- Vector Multiplication on CUDA." NVIDIA Technical Report Dec. 2008: NVR-2008- 004.

[14] Bell, Nathan, and Michael Garland. "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors." Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. Portland. 17 Nov. 2009. < http://www.nvidia.com/docs/IO/77944/sc09-spmv-throughput.pdf>.

[15] Yzelman, A., and Rob Bisseling. "A cache-oblivious sparse matrix-vector multiplication scheme based on the Hilbert curve." (2010). <http://people.cs.kuleuven.be/~albert-jan.yzelman/PDFs/yzelman11c-pp.pdf>.

[16] Yzelman, A., and Rob Bisseling. "Cache-Oblivious Sparse Matrix-Vector Multiplication by Using Sparse Matrix Partitioning Methods." SIAM 31.4 (2009): 3128-3154.

[17] Hill, Mark. "Amdahl's Law in the Multicore Era." IEEE: Computer 41.7 (2008): 33-38.

[18] Barrett, R., et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. Philadelphia: SIAM, 1994.

[19] Drexel University. Department of Physics. C Language Tutorial. <http://www.physics.drexel.edu/courses/Comp_Phys/General/C_basics/>.

[20] England, J. N. "A System for Interactive Moeling of Physical Curved Surface Objects." ACM SIGGRAPH Computer Graphics 12.3 (1978)

BIOGRAPHICAL INFORMATION

Jacob Watt was born in Denton, Texas on October 13th, 1984. In High School he was a member of the team that won two consecutive UIL Academics Science State Championships. Jacob received his Bachelor of Science in Astronautical Engineer from Purdue University in 2007. After returning back to Texas, Jacob made the decision to expand his knowledge further by pursuing a Masters degree from the University of Texas at Arlington. During his first semester at UTA, Jacob took a Computational Fluid Dynamics course through Dr. Dennis and decided he wished to pursue a concentration in CFD. During his second semester at UTA, Dr. Dennis encouraged Jacob to conduct research into programming CFD codes in CUDA.

Jacob is currently a contractor for GE Aviation. He does CFD analysis and film cooling design on aircraft combustors.