Graphics Processing Unit Acceleration Of Computational Electromagnetic Methods

Matthew Joseph Inman
University of Mississippi

GRAPHICS PROCESSING UNIT ACCELERATION OF COMPUTATIONAL ELECTROMAGNETIC METHODS

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Mississippi

by

Matthew Inman

August 2013

ABSTRACT

The use of Graphical Processing Units (GPU's) for scientific applications has been evolving and expanding for the decade. GPU's provide an alternative to the CPU in the creation and execution of the numerical codes that are often relied upon in to perform simulations in computational electromagnetics. While originally designed purely to display graphics on the users monitor, GPU's today are essentially powerful floating point co-processors that can be programmed not only to render complex graphics, but also perform the complex mathematical calculations often encountered in scientific computing.

Currently the GPU's being produced often contain hundreds of separate cores able to access large amounts of high-speed dedicated memory. By utilizing the power offered by such a specialized processor, it is possible to drastically speed up the calculations required in computational electromagnetics. This increase in speed allows for the use of GPU based simulations in a variety of situations that the computational time has heretofore been a limiting factor in, such as in educational courses.

Many situations in teaching electromagnetics often rely upon simple examples of problems due to their complexity. The use of GPU based simulations will be shown to allow demonstrations of more advanced problems than previously allowed by adapting the methods for use on the GPU. Modules will be developed for a wide variety of teaching situations utilizing the speed of the GPU to demonstrate various techniques and ideas previously unrealizable.

DEDICATION

To my friends and family. None of this would be possible without their support.

ACKNOWLEDGMENTS

I would like personally acknowledge the efforts Dr. Atef Z. Elsherbeni and the late Dr. Charles E. Smith for their personal guidance and support throughout the years. It is with their example that I continue my career.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

BACKGROUND INFORMATION ON GRAPHICAL PROCESSING UNITS

1.1 Introduction

Since the widespread adoption of computers, research has relied upon the power of the central processing unit (CPU) to perform the wide variety of computational tasks needed. Over the years great progress has been made in harnessing the power of the CPU by the introduction of faster clock speeds, larger caches, faster memory, multiple processors, and even multiple cores in a single chip. This, however, has also been accompanied by the ever-expanding needs of computer users to do a wide variety of tasks from browsing the Internet to watching video and playing games. Due to these needs of the general computer the instruction set of the average commercial processor has expanded well past 300 separate instructions in addition to the core instructions of the processor. The CPU has been forced to be a *Jack-Of-All-Trades* for computing tasks, allowing it to do a wide variety of tasks but not specializing in any particular area.

Conversely the graphical processing unit (GPU) is designed to be very narrow in nature in that they only need to perform a relatively few operations. The video card was designed with only one purpose originally, to process instructions and data necessary to provide graphics to the user. Over the past decade, advancements in the design of GPU's have been occurring at a much greater pace than with CPU's due to the narrow nature in which it was intended to be used. The current generational rate for graphics processors has been on the order of 12 months, whereas with CPU's it has been 18 months. This has led to the development of very powerful processing units for computer graphics. The current generation GPU's are running at approximately 1300 MHz with a 512 bit data bus and memory bandwidth approaching 160 GB/sec. While GPU clock speed seems slow compared to modern Pentium CPU's.

1.2 The GPU as a Programmable Processor

GPU's are essentially very specialized processors that incorporate many simultaneous instruction pipelines coupled with a memory bandwidth an order of magnitude faster than modern system memory as seen in Tables 1 and 2. In fact, the number of transistors in the latest GPU's are more than 3 times the amount used in a modern Quad-Core CPU. This leads to an ability to perform the specialized instructions the GPU was designed for an order of magnitude or faster than just using a CPU. While the majority of silicon in CPU's is dedicated to performing non-computational tasks like branch-prediction, out-of-order-execution, and cache operations, the majority of transistors on the GPU are dedicated to their computational tasks. It must be noted while the GPU was originally designed specifically for rendering graphics, not for performing computational electromagnetics, companies have been adding in features allowing these GPU’s much greater ability to perform general purpose scientific computing.

The first uses of the GPU in general purpose computing occurred roughly around the year

2000 when the needs for 3D graphics began to grow into a commonplace occurrence. This led to the major manufactures of the video cards expanding the GPU past a simple floating point processor by adding multiple cores, specialized functions for vector operations, and high speed memory. Originally, in order to utilize these cards one had to be versed in either graphics programming or in the assembly language of the processor on the card. The earliest work was completely based of assembly language programming of the cards to perform the scientific computing routines desired. This was neither arbitrary nor advantageous to the average user.

The steep learning curve required to program the cards led to several early languages such as *Sh* and *Brook* being introduced to facilitate the creation of programs that could be executed directly on the graphics cards themselves. These early languages used a combination of extensions to common programming languages such as *C/C++* and libraries like *OpenGL* and *DirectX* to interface with the cards. While these languages allowed for programs to be more easily written, they suffered from the limitations of the graphics libraries. Often arrays could never be over 4096x4096 due to the fact the graphics libraries never assumed it would never need to display on a screen bigger than 4096x4096. These earlier languages did, however, spark a movement for more widespread adoption of GPU based computing.

In 2007 NVIDIA launched their alternative to these languages named *CUDA* (Compute Unified Device Architecture), which combined a new programming interface to the graphics cards with the incorporation of new hardware inside the cards to allow them to perform better as computing processors. Every successive generation of their cards since the release of *CUDA* has included improvements and additions such as an increase in the number of processing cores and support for double-precision numbers. Since these cards have been designed from the ground up as both video cards and general computational engines, *CUDA* has allowed for mainstream adoption of the video card as a general purpose computing device.

Memory Controller	Memory Type	Memory Bandwidth
Intel X58 (Core i7)	PC3-16000 DDR3-SDRAM (triple channel)	48.0 GB/s
Intel X58 (Core i7)	PC3-12800 DDR3-SDRAM (triple channel)	38.4 GB/s
Intel X58 (Core i7)	PC3-8500 DDR3-SDRAM (dual channel)	17.0 GB/s
Intel 975 (Core Duo)	PC2-6400 DDR2-SDRAM (dual channel)	12.8 GB/s
Intel 975 (Core Duo)	PC2-5300 DDR2-SDRAM (dual channel)	10.6 GB/s
Intel 975 (Core Duo)	PC2-4200 DDR2-SDRAM (dual channel)	8.4 GB/s

Table 1. Common Memory Bandwidths of CPU Systems

	Number of Cores	Memory	Memory Bandwidth
Consumer Class			
GeForce 280/285	240 @ 1.35-1.5 GHz	1 GB	141 GB/s
GeForce 295	2 x 240 (Dual GPU) @ 1.24 GHz	1.792 GB (896 GB Per GPU)	223.8 GB/s
Business Class			
Quadro FX 4800	192	1.5 GB	76.8 GB/s
Quadro FX 5800	240	4 GB	102 GB/s
CUDA Tesla Class			
Tesla C1060	240 @ 1.3 GHz	4 GB	102 GB/s
Tesla S1070	4 x 240 @ 1.3 GHz	16 GB (4 GB Per GPU)	408 GB/s

Table 2. Common Memory Bandwidths of GPU Systems

1.3 The Hardware of the GPU

First and foremost the GPU is designed to display graphics on a computer monitor. The use of the GPU as a general computing device for scientific applications occurs chiefly due to the fact that the majority of calculations used in displaying pictures, graphics, and 3D rendering are analogous to the majority of mathematical operations required in scientific computing. Applying texture elements on a 3D object is the same as adding two arrays of data together. Drawing 3D objects only requires the processor to perform simple matrix calculations. This analogous nature of the GPU allows common routines can be adapted to run on the GPU if the hardware differences are understood.

Figures 1-3, show the hardware flow chart of an Intel Core Duo CPU, a NVIDIA G80 series GPU, and an AMD Radeon 6900 series GPU. The major difference between the current design of CPU's and GPU's lie in how the dedicated hardware is designed to operate. An Intel Core Duo processor currently contains a little over 300 million transistors. Much of this hardware is dedicated to program scheduling and flow control. The processors are able to perform a wide variety of various instructions and much of the hardware is dedicated to these various tasks. Little of the transistor space is used to for actual arithmetic processing. The GPU design is almost diametrically opposite to that of the CPU. In the current NVIDIA G80 processors there are over 1400 million transistors, mainly dedicated to actual arithmetic processing. Since there are only very few instructions possible on the GPU, much more space can be dedicated to systems to perform math instead of flow control or system functions. This is shown by the major differences in the flow diagrams of the CPU and GPU, mainly the number of ALU units (Shown as ALU blocks in the CPU and SP blocks in the GPU).

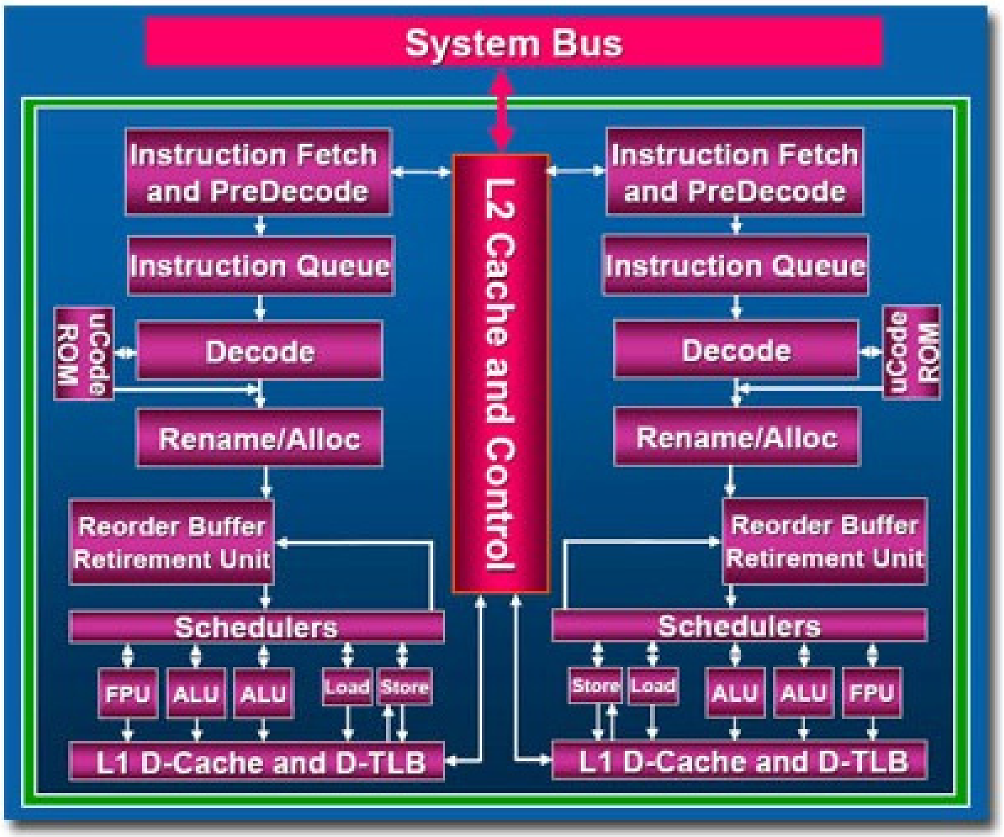


Figure 1. Flow Diagram of Intel Pentium Processor (© Intel)

In the realm of graphics programming, there are two different types of data that are operated upon: geometry and texture maps. Geometry data will usually relate the three-dimensional shapes (vertices) of various objects to the GPU. Texture maps, on the other hand, are two-dimensional arrays that relate surface characteristics of the object to the GPU. These surface characteristics may be color, reflectivity, roughness, etc. Geometry objects may be composed of several texture maps combined (i.e. the ball may be red, highly reflective, bumpy, etc). In order to apply various texture maps to the geometries in many differing ways, the GPU must be able to perform a variety of arithmetic functions on the texture maps. For example it may need to add, subtract, multiply, or divide across the texture maps, which is being done using vector math on these textures.

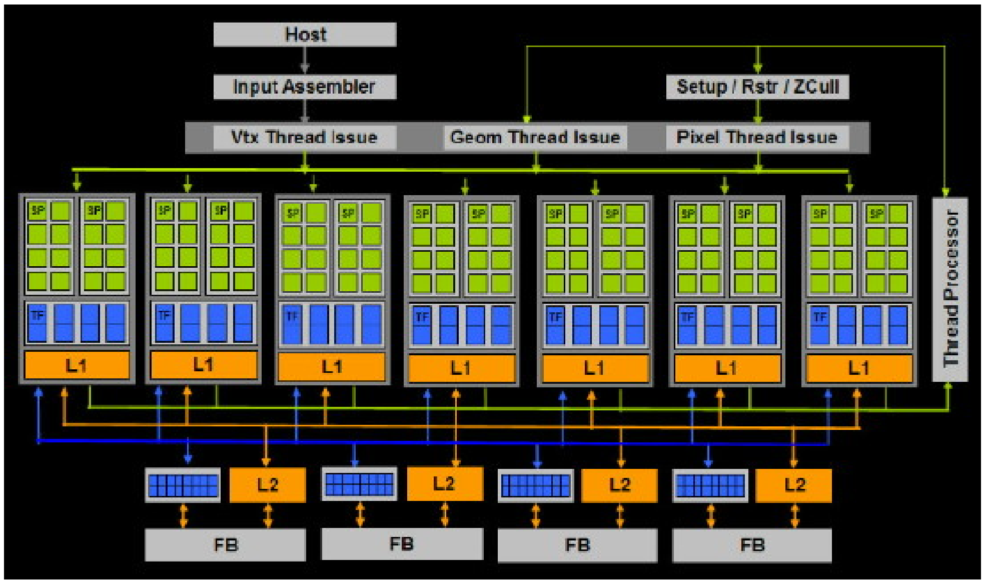


Figure 2. Flow Diagram of NVIDIA G80 GPU (© NVIDIA)

The math among texture maps occurs in a section of the GPU named the "fragment processor" otherwise known as the "pixel shader" or more currently the "stream processor". In modern cards there are upwards of 240 of these processors or more in parallel depending on the video card. Each one of these processors is fully programmable and has separate and dedicated connections into the main GPU cache and often grouped with a separate local cache as well. The purpose of the stream processor is to apply the mathematics across the texture maps to create a generalized vector processor. In the GPU there are many processors running in parallel, each processor runs the exact same program as the others with each operating on different points. Since there are many stream processors it is important for the cache to operate as efficiently as possible otherwise a fetch stall will occur. A fetch stall happens when the cache does not contain the data necessary for the operation; the stream processor then stalls until the cache has been updated with the necessary information. Mathematical operations with sequential elements (such as simple vector addition) will perform the fastest as the methodology to retrieve the sequential elements of the program is fairly straightforward. Operations such as vector or matrix multiplication suffer some performance penalty as the necessary matrix elements for the operation are much more random in nature. This randomness causes the cache to operate at a slower rate and can cause a greater chance of program stalls. However, if the program is crafted carefully, these stalls can be mitigated and a better performance gains can be realized.

For the majority of texture processes, the math only involves operations across the same element of the various matrices. In this light, the "stream processors" operate on data that is streamed into it. In other words, since every element in the texture maps are only used once and in the same order across all maps, the GPU can stream the entire array from memory instead of having to randomly access the individual elements. When data is accessed in this sequential manner, as opposed to randomly accessing elements, data can be processed at a maximal rate. As the number of random accesses that must be performed is increased, the larger a performance hit will occur. Current GPU memory speeds are an order of magnitudes faster than CPU memory speeds so the performance hit is minimal compared to the speedup that can be gained.

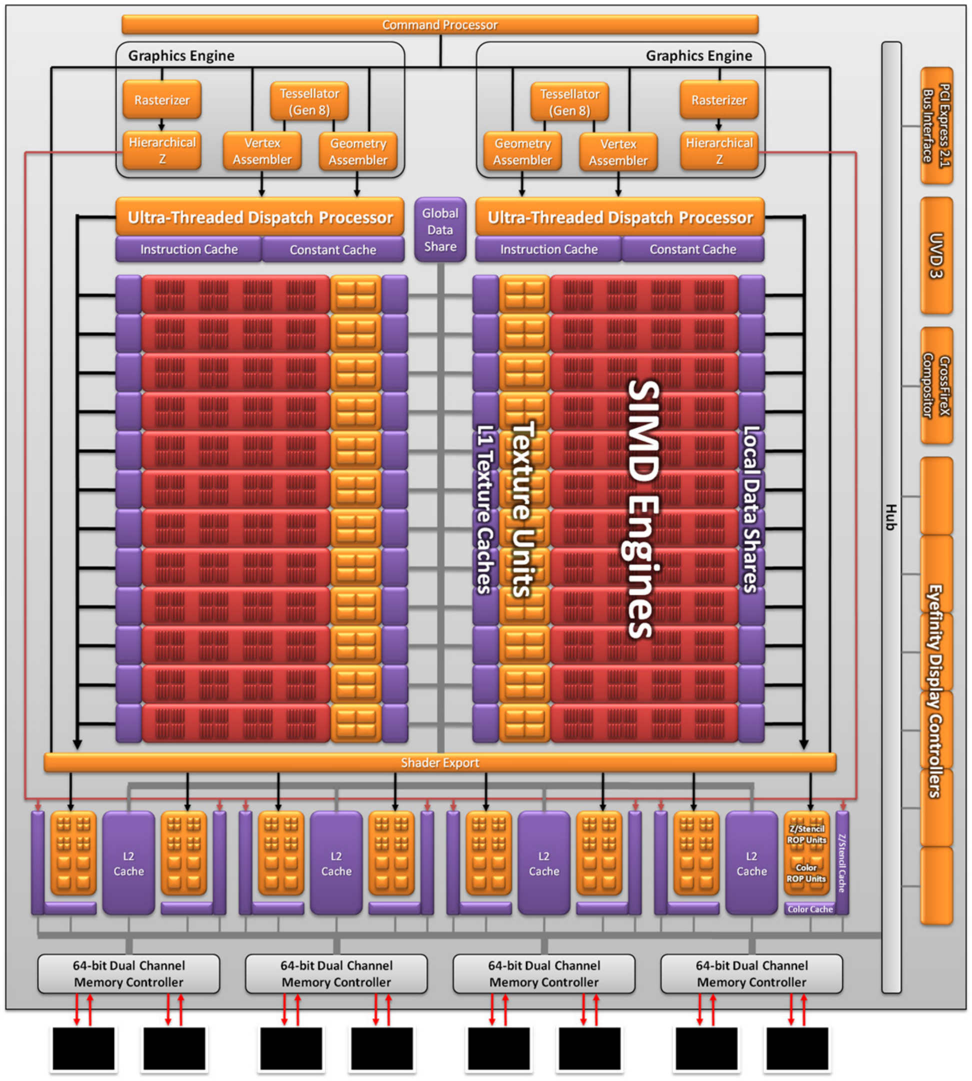


Figure 3. Flow Diagram of AMD Radeon Cayman GPU (© AMD)

Since GPU's are natively meant to process images for the screen, there are several caveats that need to be clear when attempting to program for them. The first being that all arrays stored inside the graphics card are two-dimensional, even if the interface allows them to be defined otherwise. Implementing code which the structure might be more than two-dimensional will need to use one of the various methods available for storing and retrieving the desired data. The second caveat is what can be programmed inside the GPU. The GPU only has limited ability for flow-control decisions; this means that If-Then-Else statements are not easily implemented in hardware. *CUDA* allows for the use of flow control operators but overuse of these can drastically affect execution speeds of programs. The GPU will operate the fastest when the programs are equivalent to operators which are applied to all the data that is sent to it. Decisions on the content of the data should be left to the CPU.

1.4 The GPU as a tool for teaching applications

In order to create usable modules that can be easily integrated in teaching applications the basics of how GPU's are programmed must first be discussed. Chapter two will explain how GPU's are programmed and how normal programming integrates into the GPU system. The use of programs in classes to demonstrate ideas and applications can take many forms so the areas where its use would be most appropriate will need to be identified. Most simulation techniques are either matrix or iterative solutions. Since the GPU runs programs differently than the CPU, Chapters 3 and 4 will examine how different types of computational electromagnetic techniques would best be solved by the GPU. Chapter 5 will demonstrate several modules that have already been developed for the GPU that show differing ways GPU based solvers can be used in teaching situations. Finally Chapter 6 will present conclusions about already completed work and detail future work to be done.

CHAPTER II

PROGRAMMING METHODOLOGY

2.1 Introduction to programming the GPU with CUDA

On the surface, most GPU based programs appear to be very similar to their standard CPU based counterparts. There are, however, two key concept differences between them; parallel processing and kernel construction. In essence the GPU acts as a massive parallel computing system within the actual computer. Accordingly data must be transferred to the GPU, the operations on the data must be performed, and finally the data transferred back to the CPU. In order to perform the operations needed on the data *kernels* must be written to program the GPU with the requested instructions.

The first step in utilizing the GPU is to create arrays on the GPU to store the data being used. This is accomplished in *CUDA* in the same method a *C/C++* using an function analgous to the **malloc** function. The GPU **malloc** simply reserves space in the video card memory so that it may be filled by then transferring data from the CPU memory into the GPU memory. The custom *kernels* can then be called to perform the necessary operations on the data. Once the *kernels* have finished, the data can be transferred back to the CPU for any post processing.

The operations of initializing the memory and transferring the data between the GPU and CPU are trivial in construction and will not be discussed further. The *kernels*, however, can be quite complex to create and call. Small changes in their construction can lead to major differences in their execution.

2.2 Kernels

Kernels by their definition are the custom routines programmed on the GPU to perform the mathematics on a specific set of data streamed into the processors. In *CUDA* a kernel is defined by a specialized set of commands that appear to be similar to creating a normal function in *C/C+*. For example, listing 1 is a *CUDA* kernel that adds two one-dimensional vectors of data together.

```
__global__ void gpu_sum(float *a,float *b,float *c, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) c[idx] = a[idx] + b[idx];
}
```

Listing 1. Example CUDA Kernel Code

This simple code is identified as a *CUDA* kernel by the "__global__" operator. It contains 4 arguments to call it; the input vectors "a" and "b", the output array "c", and an integer "N" that specifies the length of the of the vectors. The first line of the kernel defines the current index point of the array the processor is operating upon. By default the kernels are called with operators that tell the GPU the limits of the array being calculated and how the computational domain is divided up among the separate processors. The kernel is applied to all the processors in the GPU so each processor must know what point in the computational domain it is currently being asked to calculated. The second line checks to make sure the current index point is within the bounds of the problem then adds the current element of “a” and “b” together and saves it in “c”. The “if” statement is necessary since all processors are programmed to run the same code and might be assigned to work on a element that is out of bounds of the data. For instance if our vector length is only 200 elements and we have 240 processors trying to execute this code then 40 of the processors will be assigned to operate on memory locations that are out of bounds. Listing 2 shows the *C/C++* equivalent of the kernel.

```
for (int i=0; i<N; i++) {
     c[i]=a[i] + b[i];
}
```

Listing 2. Example C/C++ Code

While this kernel is an extremely simple example of how one can be constructed, it must be noted that on the CPU this addition is happening one element at a time, while on the GPU, the additions are split amongst many parallel processors. Operations such as this example are commonly known as extreme cases that can scale their almost linearly with the number of processors available (assuming memory access is not an issue).

Likewise there are cases where creating a kernel for a simple function becomes incredibly complex. The most common example is that of the "reduce-sum" function. This function simply tries to sum all the elements of an array together. Such functions are quite common in matrix solving routines in which every row or column of the matrix must be summed separately. Listing 3 shows the *C/C++* version of a simple reduce-sum operation.

```
for (int i=0; i<N; i++) {
        sum=sum+a[i];
 }
```

Listing 3. C/C++ Reduce-Sum Code

If a kernel would be written to try to create the same function as the *C/C++* code, several interesting questions must be asked. How many processors are available? How to split the work up evenly? How to store the intermediary results? In many ways it is not possible to generalize this simple function in a parallel manner. Many varied codes are available in *CUDA* that attempt to solve this problem and each requiring several pages of code.

2.3 In Place Modifications

The term "In Place Modifications" refers to the event when the kernel is programmed to both read and write to the same memory location, such as in the operation "`a[i]=a[i]+1`". While normally on CPU programming this is not an issue, in the situation of the GPU, with many separate processors all sharing global memory, situations where there are memory conflicts may arise. Depending upon the complexity of the kernel operations and the type of GPU being used, the onboard memory controller may have trouble keeping up with read/write access to the same memory locations. Figure 4 shows the results for a sample kernel operating upon an array of 3000 elements by adding 1 to each element 1000 times over. In this figure it can be seen that while many elements of the array do result in 1000, many do not. In this specific case the memory controller could not keep up with the numerous requests for in place modifications and resulted in a number of dropped read/writes to memory. Such situations can be corrected in one of two ways, either by adding a small pause between successive calls or by "ping-ponging" between separate arrays. "Ping-ponging" refers to using two different arrays and switching between them, letting one be the "old" set of data to read from and one being the "new" set to write to.

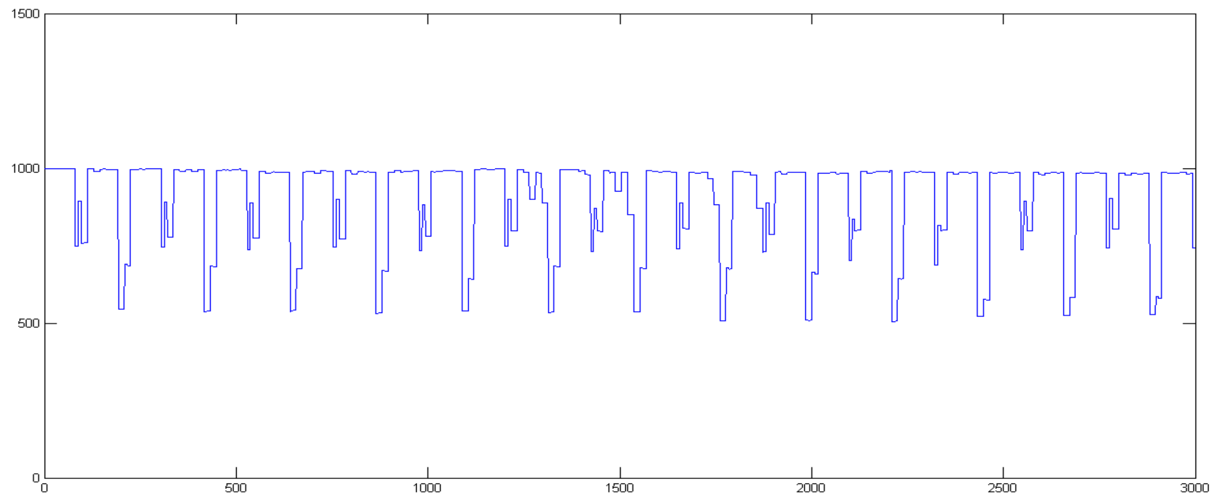


Figure 4. In-Place Modification Error

2.4 Efficient Data Collection

Most common iterative simulations are constructed to extract data from the simulation as it runs. For example in FDTD the data to be extracted usually are field components around structures to find voltages and currents as the simulation runs. In standard CPU based simulations this data extraction is simple as the program simply accesses the arrays being used in the simulation and copies the data elsewhere to be processed. In the GPU, however, it becomes a more complicated. The data needs to be extracted from the GPU and copied back to the CPU before it can be processed. Furthermore this data needs to be extracted before the next time step is run.

The major obstacle in GPU simulations is the need for the code to be self-contained on the GPU for as much as possible in order to achieve the fastest possible speed. The slowest part of any GPU program is in the transferring of data back and for with CPU. Generally the only time data is transferred in a GPU program is at the beginning to set up the problem and at the end to send the results back. In order to transfer data back and forth between the CPU and GPU everything is stopped and sent, as there is bandwidth issues between the interface bus (PCI, AGP, PCI-Express) and memory bus speeds on the two systems, even small data transfers can take large amounts of time compared with the actual time needed to run a single time step of the GPU FDTD code.

An example in FDTD might be simply extracting a single field point from the computational domain at every time step. In this simple example, the code is called every time step and a single value is to be extracted from the simulation and saved for later on the CPU. While this is easily implemented, the effects of copying even a single piece of data every time step has a large time penalty. Every time a write or read function is called to transfer data from the CPU or GPU, the system must pause to set up the data transfer. The majority of time required in these transfers occurs in setting up the transfer itself, therefore the more times these functions are called, the longer the programs will take.

To illustrate this point, a simple 1D FDTD code was written for the GPU. This code has a domain size of 3000 cells, a point source in the middle, a dielectric slab, and CPML absorbing boundaries. First the code was run with no data extraction and then run with copying the data back to the CPU at every time step. This test was run several times with 20000 time steps to get the average run times for each case. On average without any data being extracted the simulation time was 2.9 seconds. As opposed to the case where data was being copied to the CPU every time step where the average simulation time was 6.8 seconds. In this case it can be seen that copying data back to the CPU at every time step led to the simulation run time more than doubling. This increase was for just copying a single point back to the CPU, time penalties will increase the more data needs to be copied back and forth. If there are multiple points to be copied from various textures the time penalties can become quite costly.

As an alternative to copying the data back to the CPU at every time step it is possible to collect the data in the GPU itself and copy it back at a less often rate. This does increase the complexity of the code, however it will increase the efficiency. In the simple 1D example code, a temporary array was created to collect the sample points until there are ready to be transferred. By only copying between the GPU and CPU once every 1000 time steps, the amount of overhead time required to copy can be limited. In the 1D FDTD code this ran multiple times with an average run time of 3.0 seconds. Since the copy function was only called 20 times compared with 20000 the time penalty was mostly negated.

CHAPTER III

MATRIX SOLVING ON THE GPU

3.1 Introduction

Computational electromagnetic simulations generally fall into one of two classes; Matrix solutions and iterative solutions. Common simulation techniques such as Method of Moments (MOM), Finite Elements (FEM), and Finite-Difference Frequency-Domain (FDFD) all rely on solving matrix equations. Certain techniques require solving large systems of dense matrices with others require spare matrix solvers. Depending on the type of simulation being ran a wide variety of numerical methods are available for solving each matrix.

3.2 Sparse Matrix Solvers

The first type of matrix solver to be examined is case of sparse matrix systems. These problems are identified by the fact that the majority of elements in the system are populated by zeros. In many cases less than 5% of the matrix is filled with non-zero elements. Because of the large number of zero elements many various techniques may be used on the GPU to both reduce the storage requirements of the matrix (as GPU memory is limited) and speed up the solution time.

The most common techniques involved in solving spare matrix systems are conjugate gradient methods. Exploration of these types of problems for implementation fitness on the GPU has only recently begun.

3.3 Dense Matrix Solvers

The second type of matrix solvers generally used in CEM simulations are dense matrix solvers. Dense matrices are commonly found in simulations such as Method of Moments and even simple problems can often lead to complex matrices whose size can order in the thousands. In order to accurately and quickly solve these simulations, especially cases where there are many of right hand sides to be calculated, an appropriate solution method must be chosen.

Implementation of solvers for these dense matrix problems on the GPU generally occurs in two basic areas; Matrix filling and matrix solving. Both of these processes show good applicability for integration on the GPU. There are a wide variety of various methods for solving any dense matrix systems, each with certain advantages and disadvantages such on complexity and convergence issues. One of the more common techniques is that of LU Decomposition, where a matrix is "decomposed" into an upper and lower triangular matrix. Implementation of a GPU based LU solver will be detailed in the next section.

In the realm of GPU implementations LU decomposition offers many advantages over other decomposition, inversion, and direct solution solvers. For a large number of right hand sides, direct solution solvers become unwieldy to implement, as each right hand side requires its own solution. Inversion methods can allow for the solving at will after the matrix has been inverted but often require large computational runtimes and can suffer from instability as the order of the matrix becomes too large. Decomposition methods offer a good compromise between full inversion and direct solution. LU decomposition in particular lends itself well to implementation on the GPU.

Using the CUDA interface, many of the computations required for LU decomposition can be offloaded to the GPU. While LU decomposition on the GPU has previously been demonstrated to outperform the CPU, the past work has been limited to only solving real matrices. For LU decomposition to be of use in computational electromagnetics, GPU implementation for complex matrices must be available.

3.3.1 Complex Double-Precision LU Decomposition

The LU decomposition has been previously demonstrated on the GPU using CUDA and other programming techniques for single precision real matrices. Published result produced speed gains approaching an order of magnitude over common CPU's. These solvers mixed a combination of CPU Basic Linear Algebra Subprograms (BLAS) calls, CUDA CUBLAS (NVIDIA's GPU based BLAS libraries) calls, and CUDA kernel. The BLAS libraries contain highly tuned functions commonly used in many programs to perform basic linear algebra. The published LU solvers were facilitated by the complete and mature development of CUBLAS libraries for single precision real data types. These solvers showed a speed increase of 6 to 12 times (relative to various hardware). However, the restriction of single precision real data types limits its usefulness for CEM simulations. Many common CEM problems require the solver to be available for any combination of single precision, double precision, real, and complex data.

The development of solvers that support data other than a real single precision on the CUDA/GPU platform presents several unique challenges to be addressed. These challenges occur from the status of the CUBLAS libraries. The CUBLAS libraries (previous to release 3.0) only supported complete BLAS routines in single precision real and only very limited support for single and double precision complex. In the utilized version 2.0 of CUBLAS, only 2 out of 13 level 1 BLAS routines, 1 out of 16 level 2 BLAS routines, and 2 out of 6 level 3 BLAS routines were supported. The CUBLAS version 3.0, expands support for all BLAS routines in more data types.

With the release of CUBLAS 3.0 it is now possible to perform the LU decomposition directly on the GPU without the aid of any CPU calls. However, this does not mean that the CUBLAS functions outperform their CPU based counterparts. Certain linear algebra functions still perform significantly faster (such as factorization) on the CPU compared to the GPU's as utilized. The algorithm presented here was carefully profiled to determine when and which parts of the LU decomposition routine can be solved on the GPU with maximum efficiency.

The real single precision solver presented here follows the published methodology of utilizing both the CPU and the GPU and the established algorithms for parallel computing systems. The code has been programmed and tuned using these methods. In order to extend this solver for other data types, some of the CUBLAS calls have been replaced with custom developed kernels (GPU functions).

In the solvers presented here, the “*trsm” function which is a standard BLAS routine used to solve a triangular matrix, has been offloaded to the CPU. The transpose functions have been developed in CUDA to support all types of data (complex and real in single and double precisions). With this added support for the various data types, the developed GPU code was tuned for various block sizes which determines how much data gets transferred, at a time, between the GPU and CPU. Offloading the “*trsm” function back to the CPU also presents problems in maintaining data consistency. The transfer of data between CUBLAS on GPU and Intel MKL BLAS on CPU is simple when working with single (float) or double precision real numbers. However, for complex data, MKL BLAS and CUBLAS have different data types and data structures to represent the numbers. In order to accomplish consistent data transfer, the MKL BLAS has been modified so that its data structure is compatible with CUBLAS data types. This modification allowed the free exchange of data between CUBLAS on the GPU and MKL BLAS on the CPU for complex numbers.

The custom routines in CUDA for transposition and pivoting, were developed to support all combinations of data types. Depending on the data type needed, the additional data overhead requires smaller blocks of the matrix to be transferred at a single time (as a double precision complex matrix has 4 times the data as a single precision real matrix). The transpose routines make use of local cache memory inside the GPU in order to make this process as efficient as possible.

Table 3 details the various functions used for the developed CPU+GPU based LU decomposition and where they are performed. The basic algorithm iterates through the various block columns of the matrix and performs the decomposition. Each block is first transposed and the L/U matrices are updated on the GPU. The block is transferred to the computer system and factorization takes place on the CPU. The block then streams through the GPU for pivoting and back to the CPU. The block is then inverted and the L matrix is solved. The update for the U matrix is performed on the GPU, then the data is transferred back and the final U solve is done on the CPU. Applicable code is detailed in appedicies A,B, and C.

Transpose Block	GPU (CUDA Kernel)
Matrix Multiply	GPU (CUBLAS)
Factorization	CPU (MKL BLAS)
Pivot	GPU (CUDA Kernel)
Triangular Matrix Solve	CPU (MKL BLAS)

Table 3: Functions required for LU decomposition

While the construction of LU Decomposition solvers on the GPU has been well documented before, they have remained solely in the domain of single-precision real values. While many real world applications fit nicely into these limitations, most engineering problems require the use of complex values. The next step in creating a useable LU Decomposition routine was to extend published methodologies to support complex values in both single and double precision.

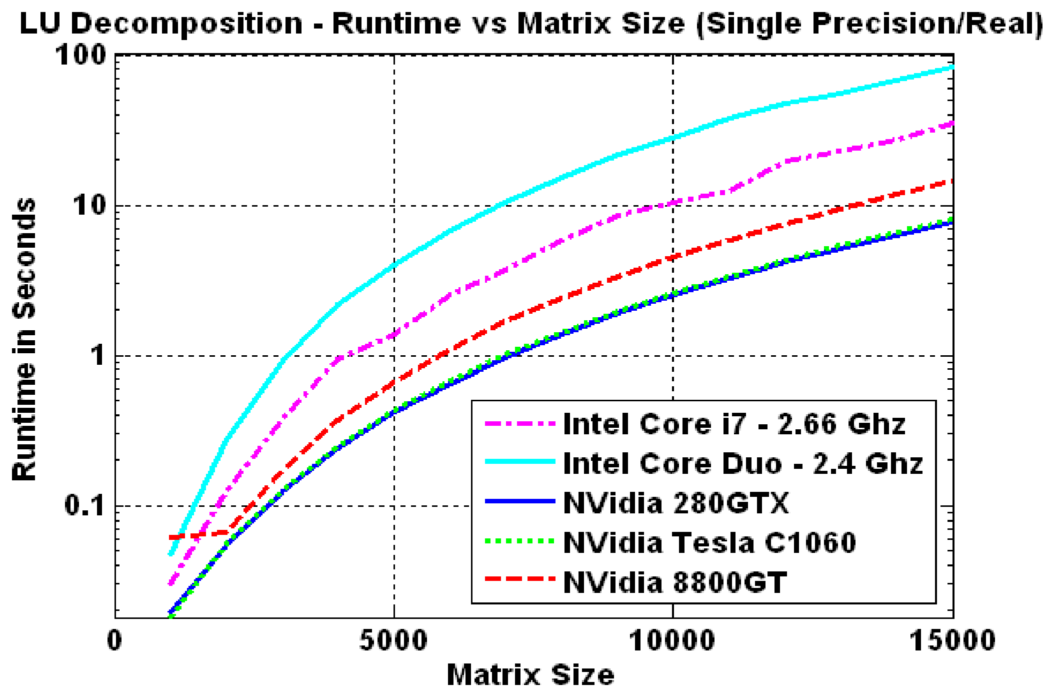


Figure 5. Runtime Speeds for Single Precision Real Value LU Decomposition

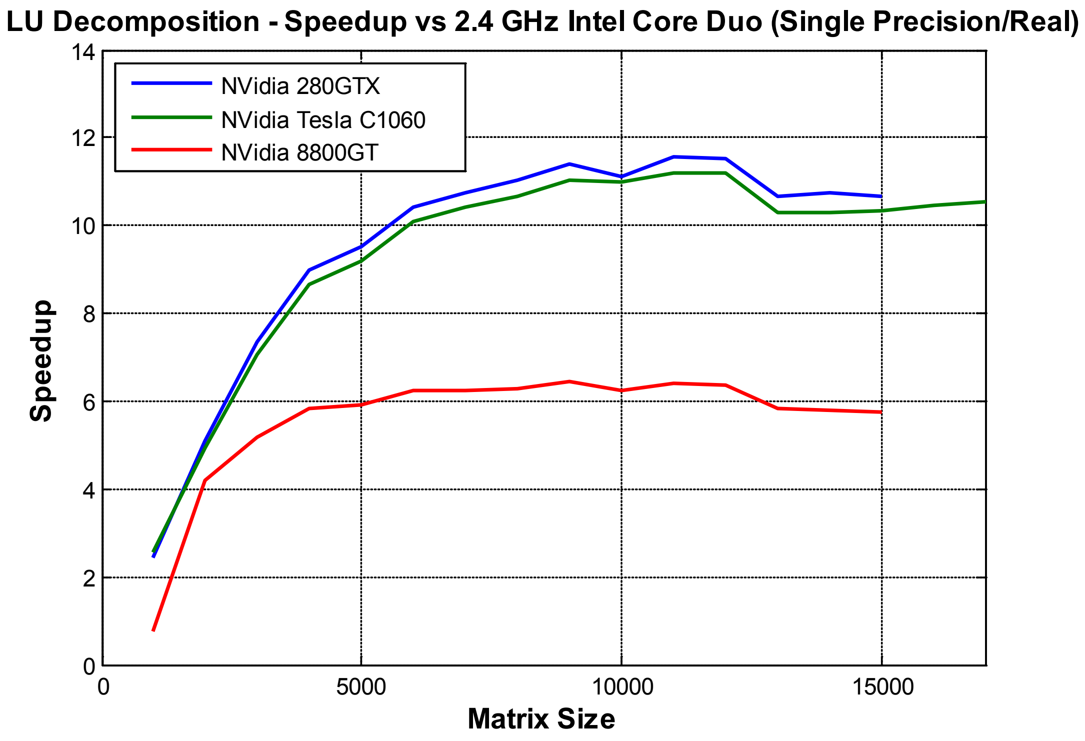


Figure 6. Speedup Factors for Single Precision Real Value LU Decomposition

In the real double precision cases, the CPU+GPU implementation achieved a speed gain of seven times over the CPU only based counterpart. Interestingly, even though twice the amount of data is required to be moved for a double precision case and known inefficiencies of the GPU processing double precision data, the CPU+GPU case only increased runtime by 90%. This can be explained by examining the memory access patterns in processing double precision data. In algorithms such as LU decomposition, data access to the memory of the CPU and GPU are not optimal for the fastest transfer. The addition of double precision data in these cases actually increase the efficiency of memory access since larger blocks of linear memory is being read at a single time. The addition of double precision arithmetic for these cases did not account for any noticeable increase in processing time. This is due to the fact that in these cases the arithmetic is fairly simple. The calculations were completed before the next block of data has arrived from memory even with the overhead of double precision calculations.

Depending upon the size of the matrix complex double-precision support must be available. While many of the subroutines used in LU decomposition can be run on the GPU faster than the CPU, some portions of the code are still more appropriate to run on the CPU. Maintaining data integrity between CPU and GPU complex double-precision data types must be preserved. The inclusion of double-precision calculations will also be examined from a memory standpoint in optimizing the local cache memory in the GPU for the fastest execution times.

Double precision complex development on the CUDA platform presents several unique challenges to be addressed. These challenges occur from the incomplete development of the

CUBLAS libraries (BLAS libraries being optimized functions used commonly in linear algebra and CUBLAS being a standard CUDA implementation of them). Currently the CUBLAS libraries only support complete BLAS routines in single precision real and only very limited support for single and double precision complex. In the current version of CUBLAS only 2 out of 13 level 1 BLAS routines, 1 out of 16 level 2 BLAS routines, and 2 out of 6 level 3 BLAS routines are supported (Levels referring to the complexity of the routines). In the development of a single precision code for LU decomposition, the CUBLAS libraries can be extensively used. For double precision code with support for complex numbers, the CUBLAS libraries must be supplemented with custom CUDA BLAS kernels and CPU based BLAS routines.

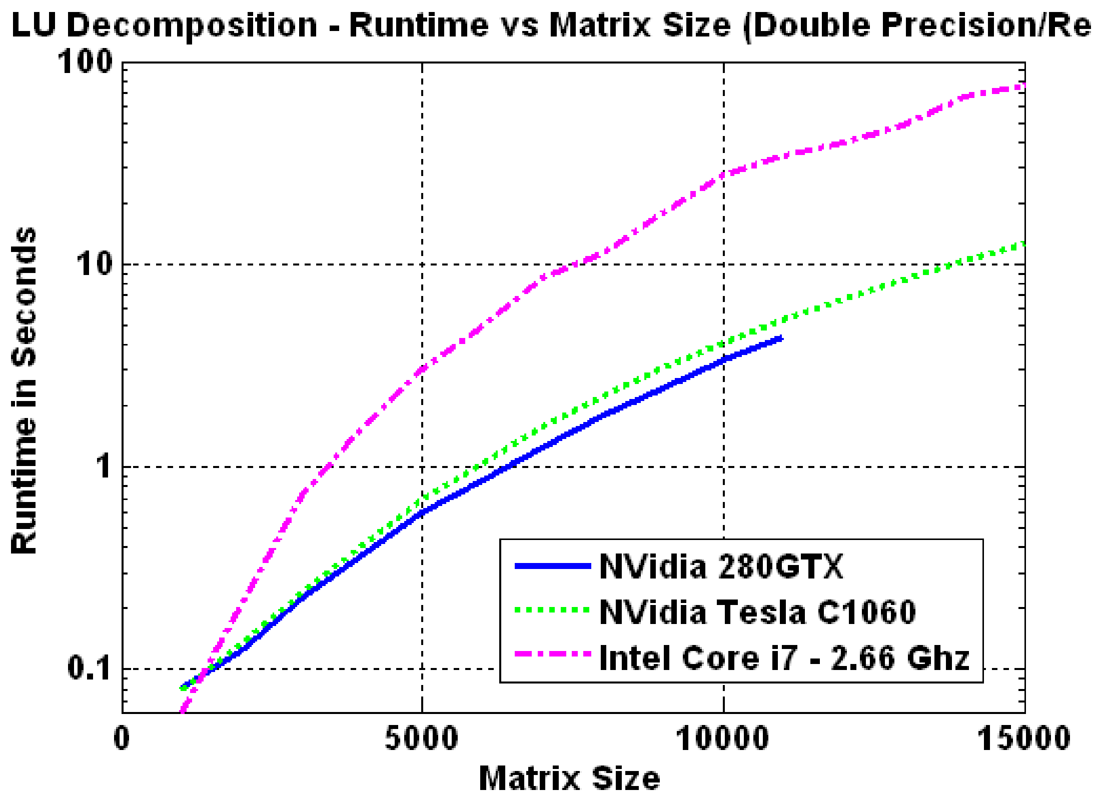


Figure 7. Speedup Factors for Double Precision Real Value LU Decomposition

In order to compensate for the lack of several appropriate BLAS routines in CUBLAS, the “Zsrtsm” function has been offloaded to the CPU, while the transpose functions has been written in CUDA with support for double precision complex numbers. Offloading the “Zstrm” function back to the CPU also presents problems in maintaining data consistency. When working with single or double precision real numbers, transferring data between CUBLAS and Intel MKL BLAS (the CPU BLAS used here) is trivial as these routines operate with the same data types (float or double). The complex MKL BLAS and CUBLAS have different data types and data structures to represent the data. In order to accomplish consistent data transfer the MKL BLAS has been modified so that its data structure is compatible with CUBLAS data types. To use the MKL BLAS functions the CUBLAS data types must be forced recast into MKL BLAS.

The custom routines written in CUDA for transposition were written to support the complex double precision numbers. The added data overhead requires smaller blocks of the matrix to be transferred at a single time (as 1 element of a double precision complex matrix has 4 times the data as a single precision real matrix). The transpose routines make use of local cache memory inside the GPU in order to make this process as fast as possible. At this point these routines have only begun to be optimized for speed, as the memory required for complex double precision as well as the memory layout makes this process difficult.

The addition of double-precision complex support for a GPU based LU decomposition solver has allowed a moderate speed gain of 2 to be achieved as seen in Figures 6 and 9. These figures relate how the addition of double-precision complex numbers lowers the possible speed gains of the GPU. This low number (in comparison to single-precision speeds) is first due primarily to the immature double-precision hardware on the GPU itself. Future generations of GPU's are expected to greatly increase the double-precision speed.

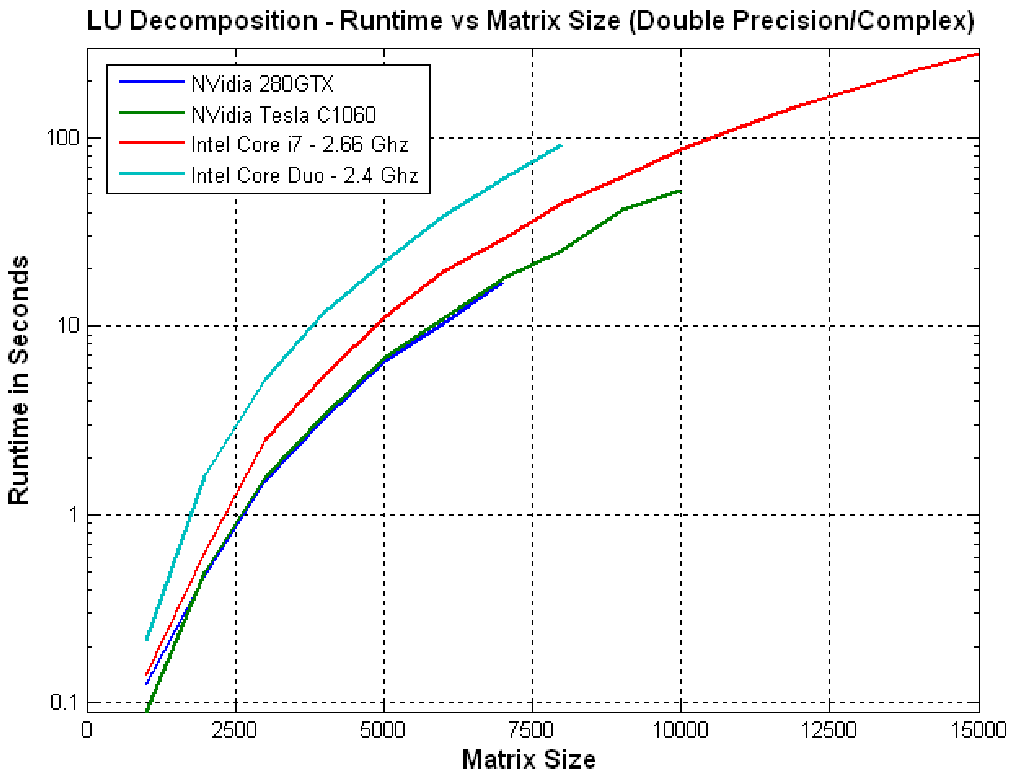


Figure 8. Runtimes for both CPU and GPU implementation of Complex Double-Precision LU Decomposition

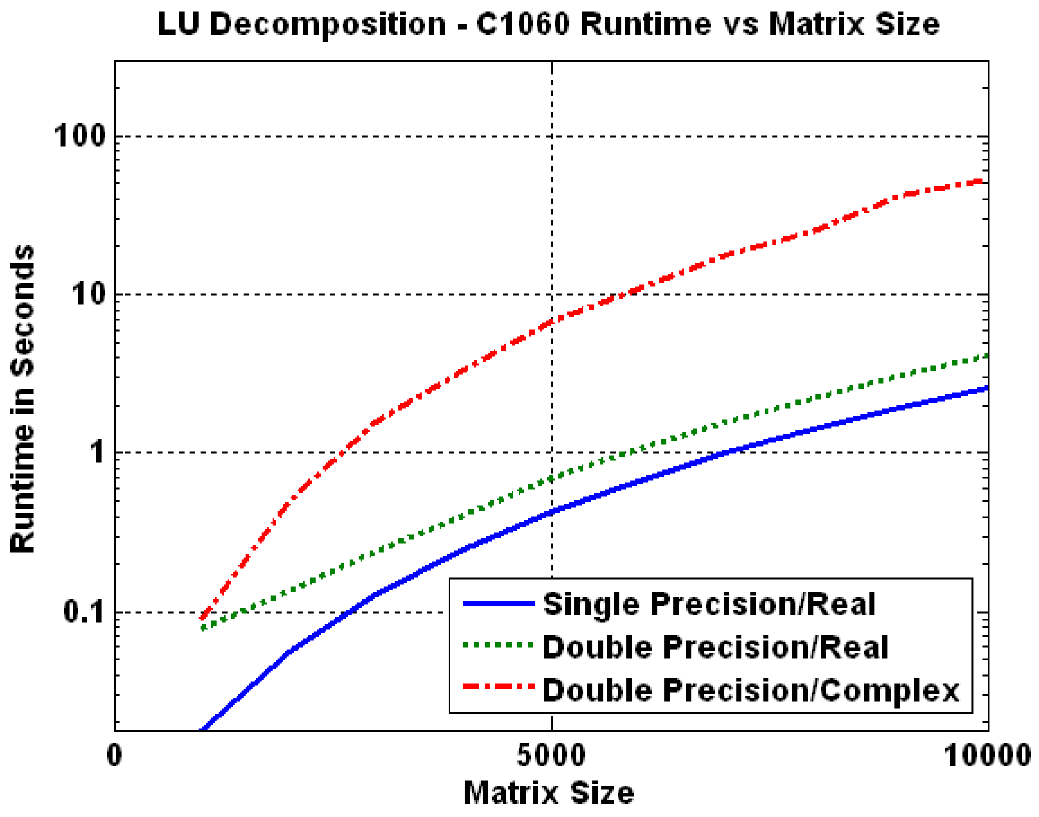


Figure 9. Runtime of various GPU based implementations of LU Solvers

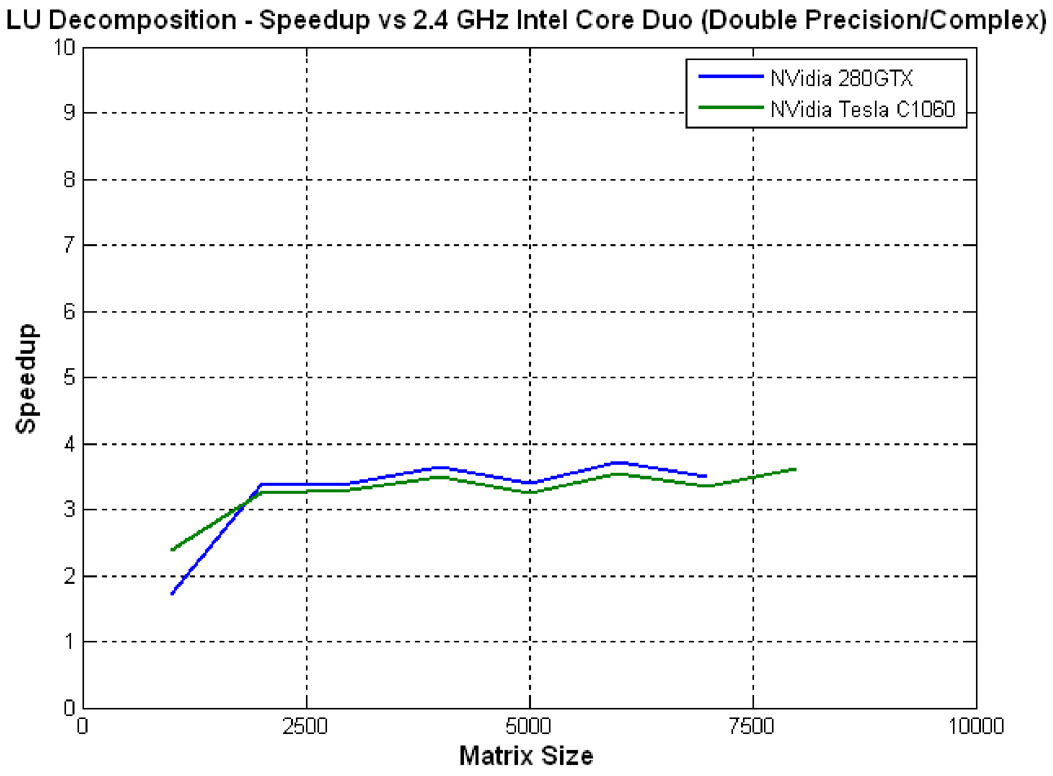


Figure 10. Speedup factors of GPU based over CPU based implementations of Complex Double-Precision LU Decomposition

3.4 Complex Double-Precision Method-of-Moments Results

To show the use of the GPU based solver, a well known sample problem was chosen. In this sample, the current along a wire antenna of length L (0.1m) and radius A (0.1mm) that is excited by a magnetic frill model will be calculated as shown in figure 1. This simulation will be calculated using sinusoidal basis functions and mid-point integration.

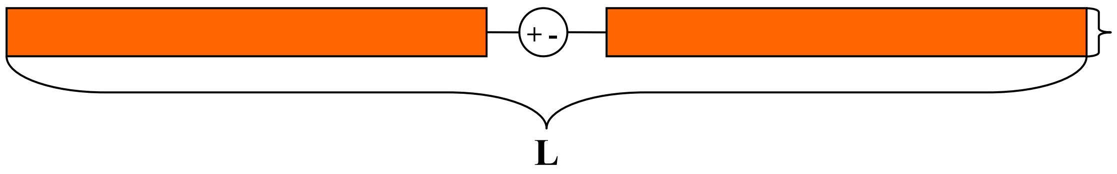


Figure 11. Sample wire antenna configuration

The sample problem was chosen in order to validate the simulations against existing codes and for its simplicity in integration into the GPU solver codes. Because of its nature it is simple to change the discretization of problem and examine the solution times as a function of the subsequent matrix size.

The GPU code was run against the reference codes to ensure proper operation. Figures 12 and 13 shows the current along the wire in both codes for a sample discretization of 1024 segments. The results show very good agreement with only very minor differences in the magnitude of the current. These differences (less than 0.1%) can be attributed to minor differences in how the numbers were stored and calculated in the various programs and the use of the GPU in the simulation. The errors in the phase calculations were even smaller by several degrees of magnitude which means the differences were most likely due to the differences in how the GPU and CPU handles rounding.

Figure 14 shows the various solution times for different matrix sizes. These solution times were measure using the same program operating in either CPU only mode (using Intel MKL BLAS for the calculations) or in CPU+GPU mode (Using NVIDIA CUBLAS to operate on the majority of the simulation). These are the simulation only run times and do not include matrix fill times. The results shown are for several different configurations of systems and graphics cards as noted on the figures. From these results it can be seen that as the matrix size increases the GPU codes run approximately twice as fast as the CPU only codes run compared to a quad core 2.6 GHz Intel i7 machine and approximately 4 times faster compared to a 2.4 GHz Intel Core Duo. Since the GPU only has begun to support double precision calculations recently this slow down can be attributed to the relative immaturity of the GPU hardware in this aspect.

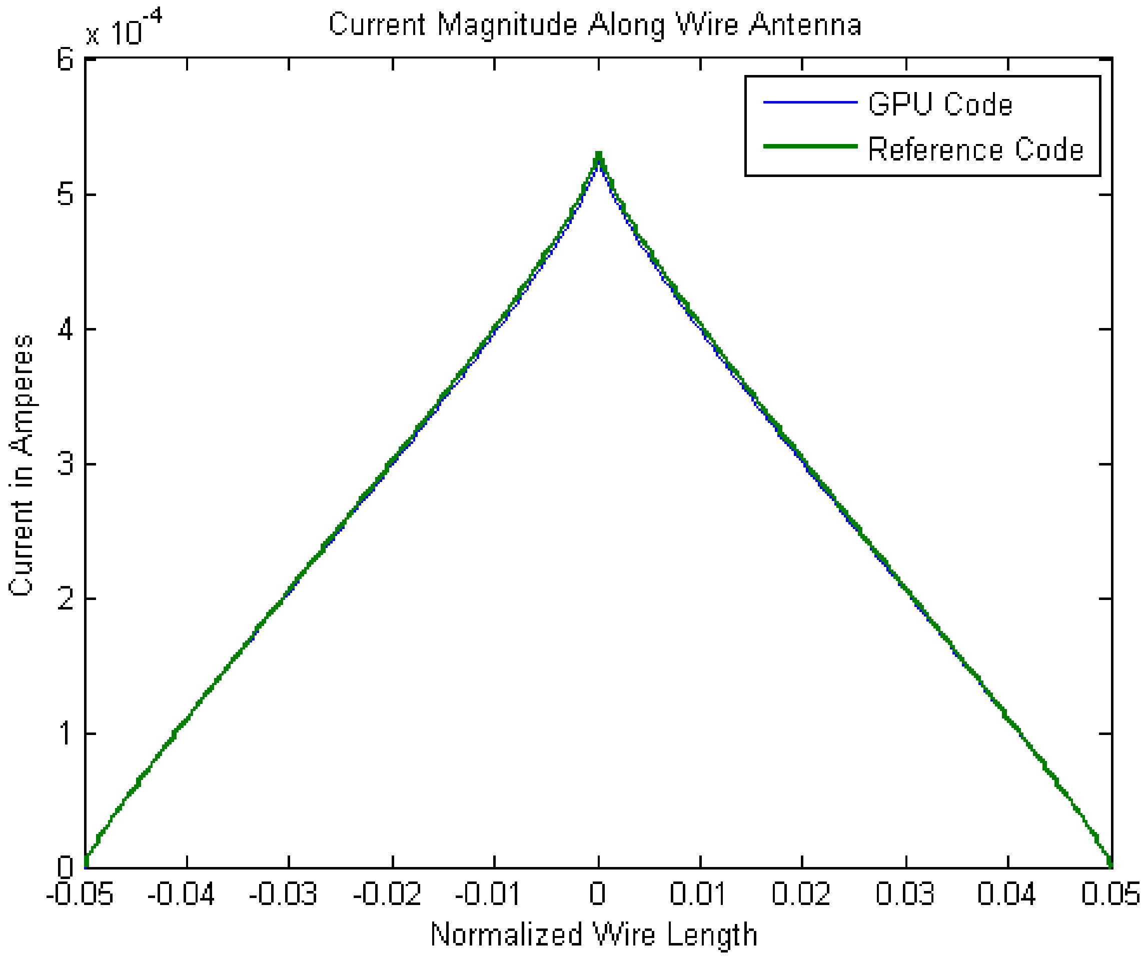


Figure 12. GPU and Reference results for current

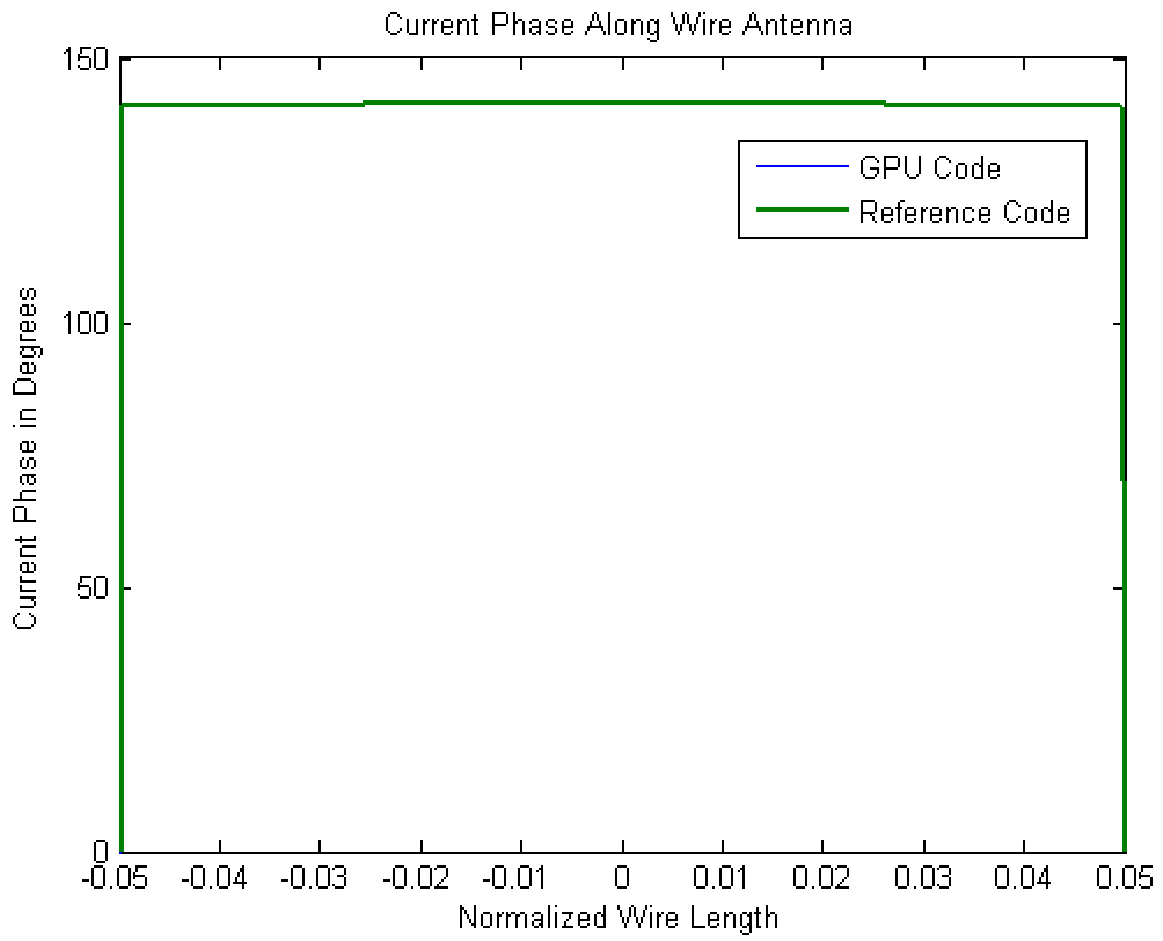


Figure 13. GPU and Reference results for current phase

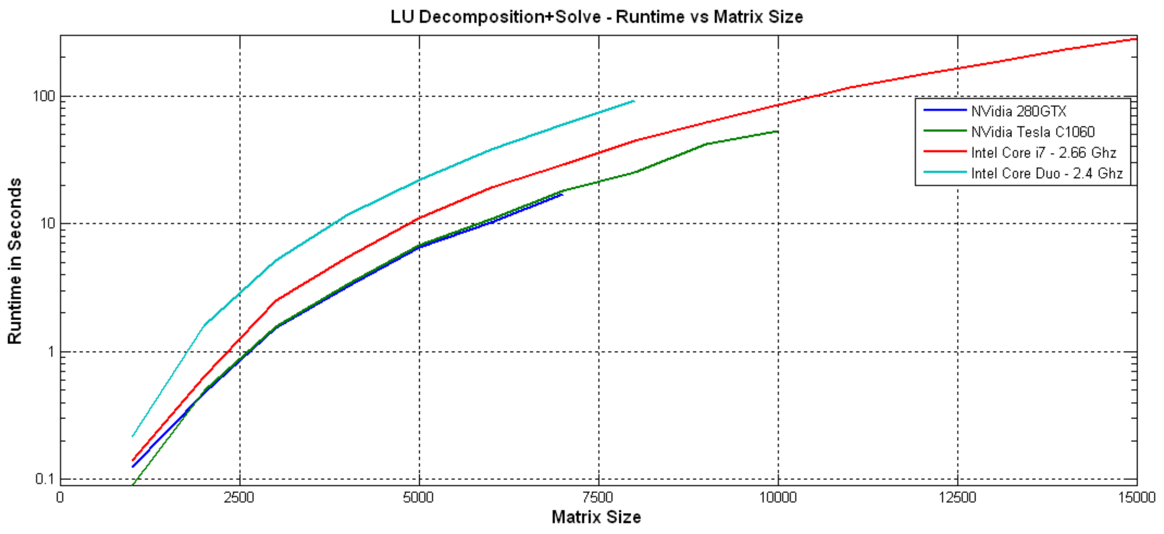


Figure 14. CPU and GPU solution times for various matrix sizes

CHAPTER IV

ITERATIVE METHODS ON THE GPU

4.1 Introduction

The second class of problems commonly occurring in computational elecromagnetics are iterative methods such as the Finite-Difference Time-Domain method (FDTD). FDTD attempts to solve Maxwell's field equations in the time domain by applying their partial derivative form across a structural domain in a time-marching fashion. Iterative techniques such as FDTD are highly suited for GPU based applications, as they generally require applying a set of standard equations over the entire domain. Since there is no decision-making or branching involving, and the memory accesses are fairly sequential, iterative techniques often show the largest speed gains.

4.2 Finite-Difference Time-Domain

Finite-Difference Time-Domain (FDTD) solvers for electromagnetic simulations have been around for many years, however, it have been the recent advances in computer technology that has seen the technique gain wide use. As computers become more powerful and systems with larger memory are introduced, applications for FDTD increase as well. FDTD is increasingly used to simulate larger and larger problems, which require more memory and faster processing in order to complete the simulation in reasonable amounts of time. Even with current generation computers the FDTD method is still limited in the speed for a simulation to be performed. Most of the research into the FDTD method has been on introducing new formulations for solving problems and more efficient absorbing boundary conditions. In the past years, researchers have begun to explore different ways to implement FDTD solvers in alternative methods in order to gain increases in speed. Basing FDTD on graphics processors has been shown to offer orders of magnitude speed increases in simple vector operations required in the method.

4.3 Finite-Difference Time-Domain Domain Tiling and Absorbing Layers

Integrating a fully functional FDTD simulator on the GPU presents several obstacles that need to be overcome in order to operate efficiently. The first of these problems is how to efficiently store and access what is essentially a three-dimensional domain inside a two-dimensional storage space. Figure 15. details the most common method of translating into two-dimensional space, tiling. Here it can be seen that the three-dimensional space has been "sliced" across the z-dimension and tiled into the two-dimensional array. In this system, accessing neighboring elements in "x" or "y" is preserved, while accessing neighboring elements in "z" require movement in "y" up or down the slices. This preserves much of the sequential memory accesses needed to gain the most speed possible. This tiling, however, presents a few problems in programming other functions commonly needed in FDTD such as absorbing boundaries. In most CPU based implementations, while not trivial, are fairly easily implemented. On the GPU their implementation can be much more complex. Because these absorbing boundaries and their constituent equations are usually applied only in certain areas of the computational domain, how to apply them properly while maintaining speed becomes a key issue. Two separate methods are commonly used in GPU programming of common boundary types such as CPML. The first being using if-then operators in the code to test if the equations are to be applied and the second to apply the equations over the entire domain. While the second method offers certain advantages to maintaining speed gains it does require storing both data and coefficients for the entire domain and negatively impacts memory usage. The second method also offers ease of implementation since the equations can be applied uniformly on the entire domain (with only the areas inside the boundaries having none-zero coefficients).

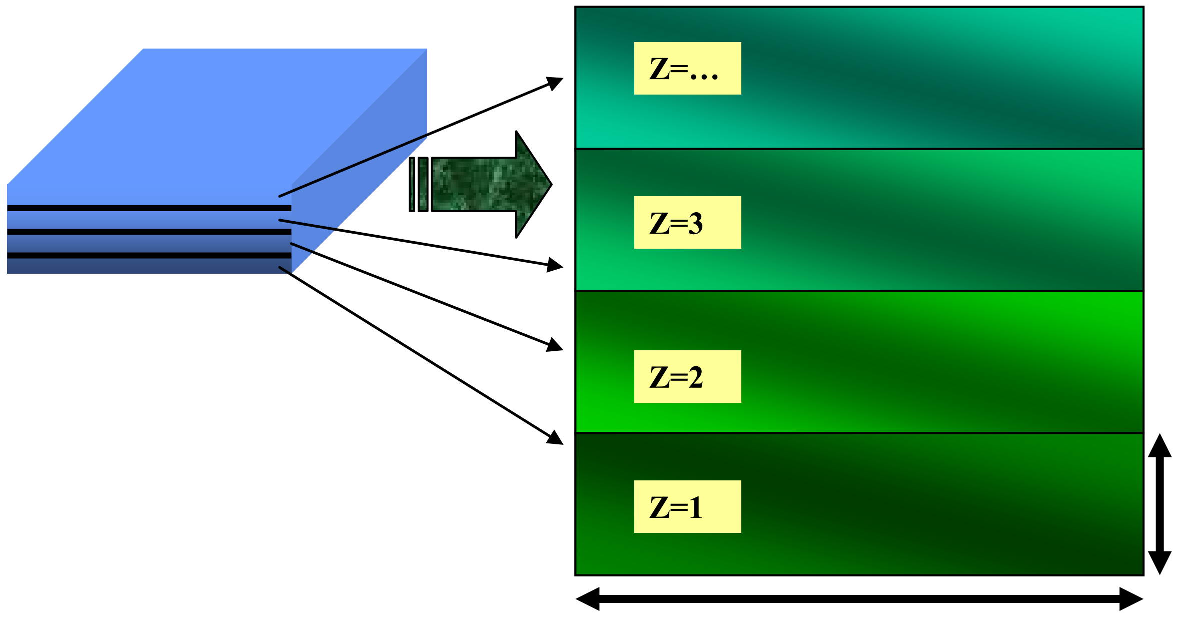


Figure 15. 3D Domain Tiling

Since the domain is tiled it is useful to consider exactly where the absorbing boundaries are being applied. Figure 16. shows a sample domain and where a CPML absorbing boundary is present. It can be seen that while he “x” and z” boundaries are very well defined, the “y” boundaries are more scattered throughout the domain. It is from this fact that using if-then statements to determine where to apply the boundaries becomes quite complex to the GPU if that method is used.

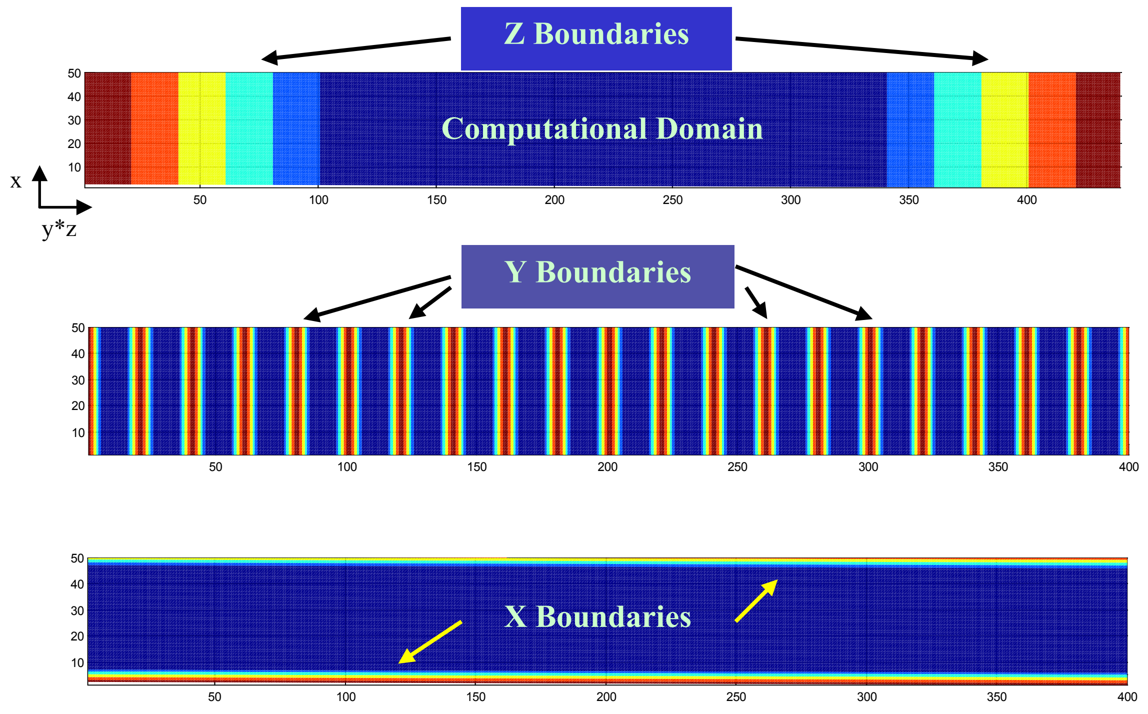


Figure 16. PML Boundaries in a sample computational domain

The second obstacle to be overcome is the need for efficient extraction of field components necessary in FDTD simulations. Normally these field components are used for calculating surface voltages and currents and require the extraction of many various points in the domain at every time step. Efficient extraction methods were covered previously in chapter 2.

A FDTD code was implemented on a GPU (appendix D) to test the speed gains possible and to validate the results against existing codes. In this example as shown in Figure 17, a simple printed dipole antenna was constructed with the parameters shown. The voltage and current from the source were extracted and analyzed for comparison.

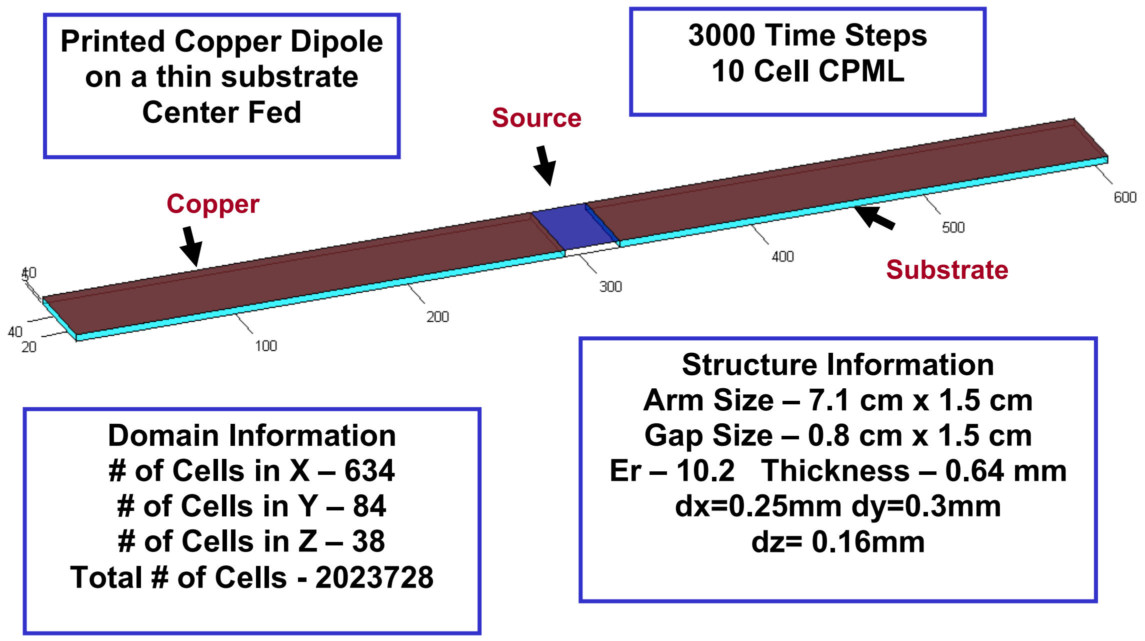


Figure 17. Sample GPU FDTD Domain of Printed Dipole Antenna

This simulation was run using both CPU and GPU based FDTD simulators. On the CPU based program the simulation was completed in 23.2 minutes while on the GPU the simulation was completed in 55.2 second. These simulation times resulted in a speedup factor of approximately 25 times for the GPU implementation of FDTD. Comparison of the output data showed good agreement between the two implementations.

A second test case was implemented to test the efficacy of the CMPL boundaries using this method. In this case as shown in Figure 18, a point source is used to excite the domain while an object that may be either PEC or dielectric is placed to cause complex reflections. Several observation points were used to study the magnitude of the reflections as shown in Figures 19-21.

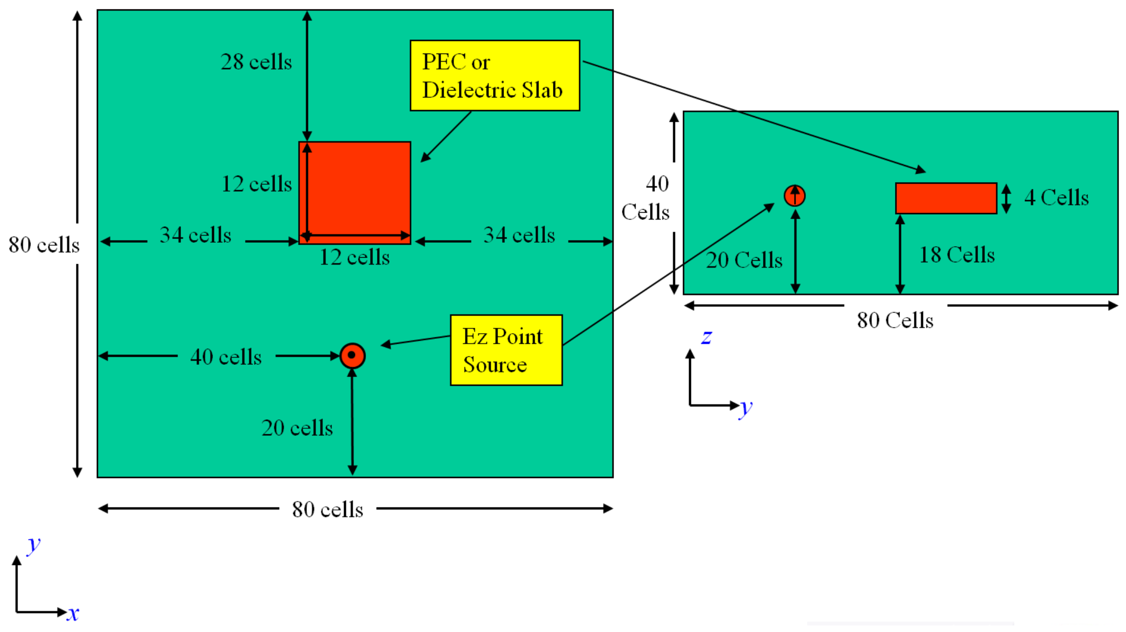


Figure 18. CMPL Test Case 2

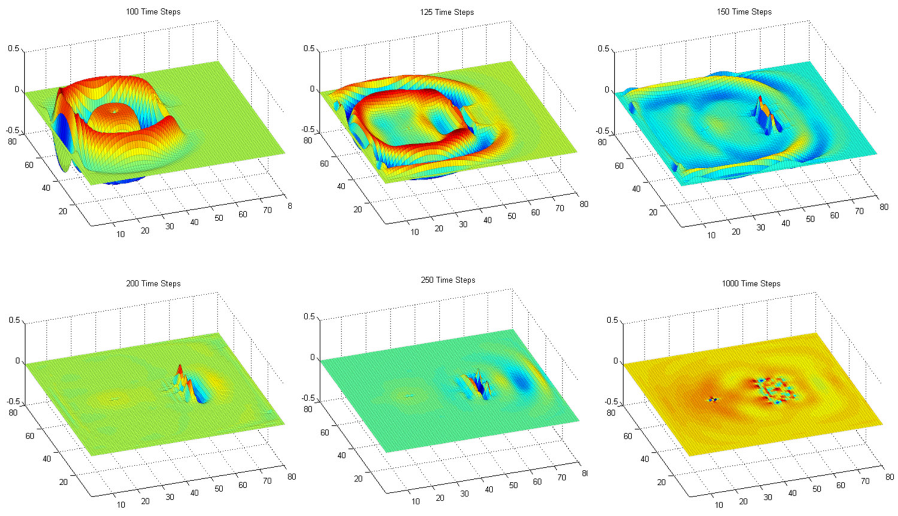


Figure 19. E_z field component at various time steps for dielectric block of ε_r=10.2

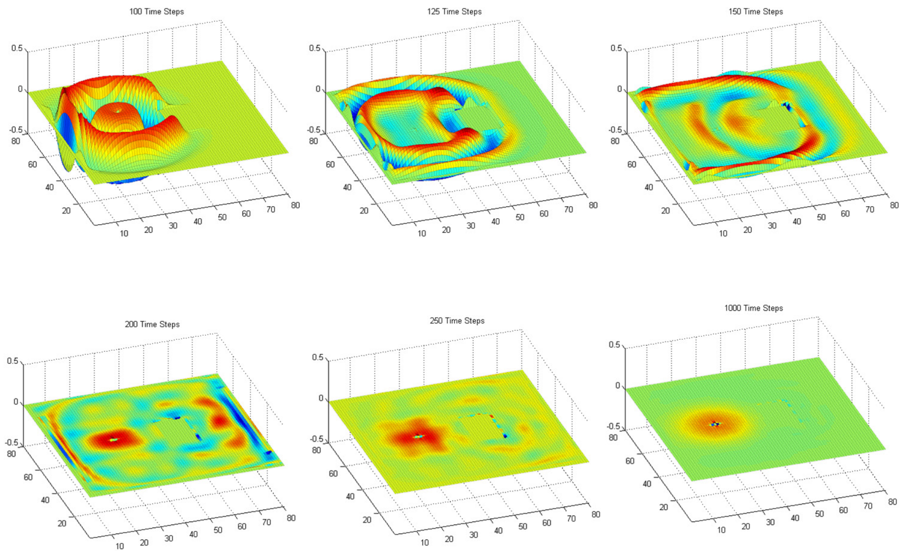


Figure 20. E_z field component at various time steps for PEC block

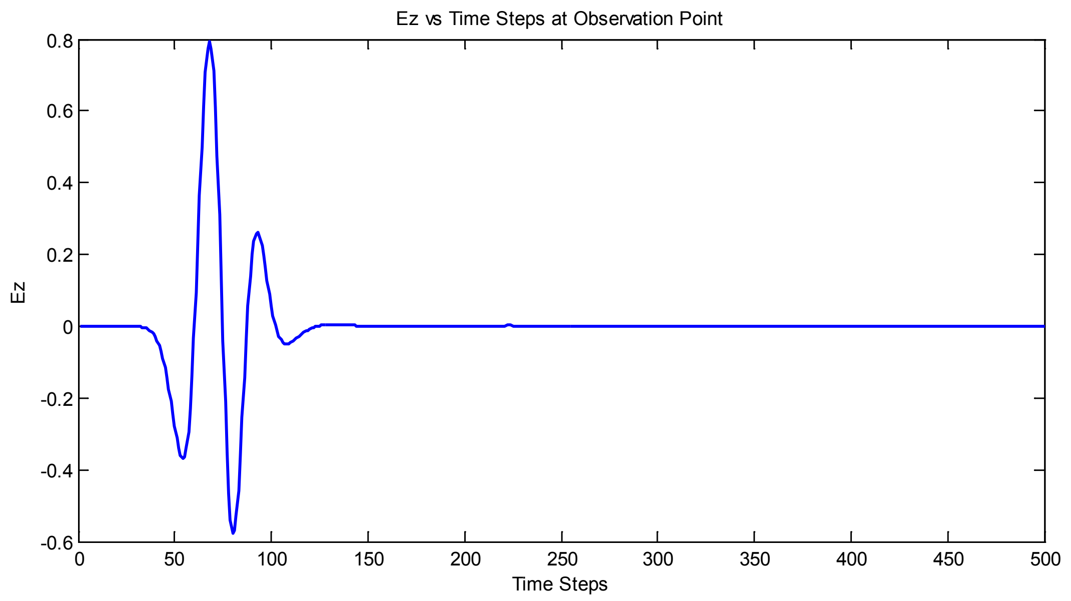
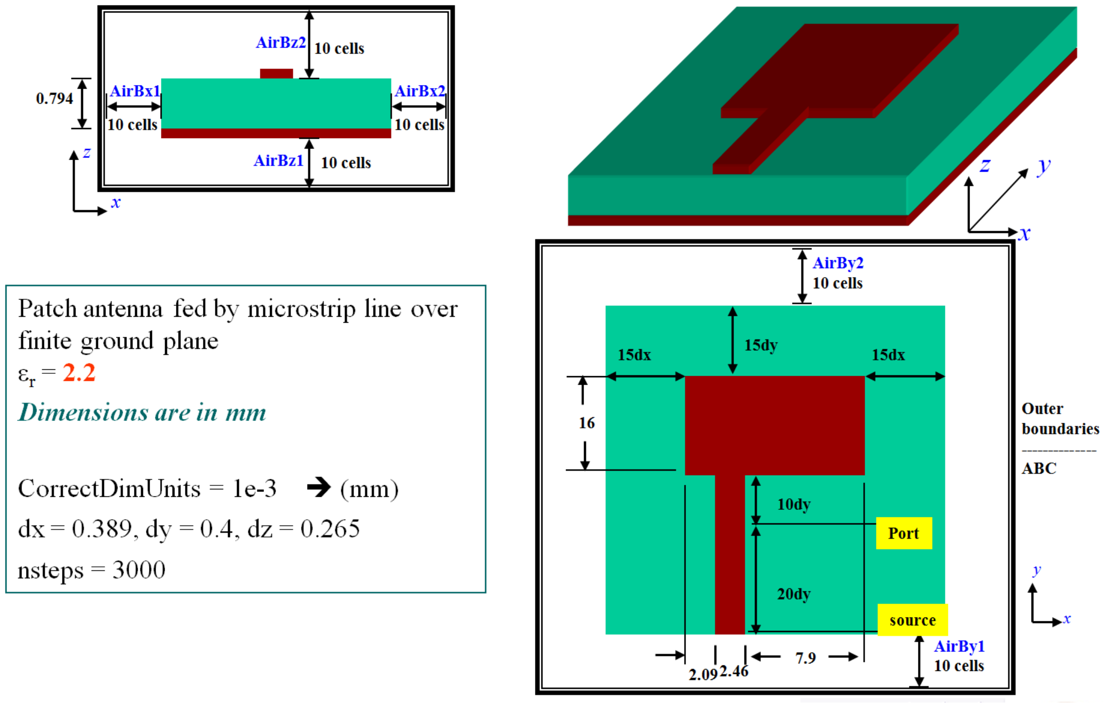


Figure 21. E_z field component at observation point

4.4 Finite-Difference Time-Domain Verification Cases

In order to verify the proper operation of the GPU FDTD code with boundary layers, several verification cases were simulated. These verification cases were taken from the well-known paper "Application of the Three-Dimensional Finite-Difference Time-Domain Method to the Analysis of Planar Microstrip Circuits".

The first case considered is that of a simple microstrip antenna as shown in Figure 22. This antenna is fed at the base of the microstrip feed line and the results are observed at a port located closer to the patch itself. Results showing excellent agreement are shown in Figures 23-25.



Patch antenna fed by microstrip line over finite ground plane

$\varepsilon_r = 2.2$

Dimensions are in mm

CorrectDimUnits = 1e-3 ➔ (mm)

dx = 0.389, dy = 0.4, dz = 0.265

nsteps = 3000

Figure 22. Microstrip patch antenna validation case

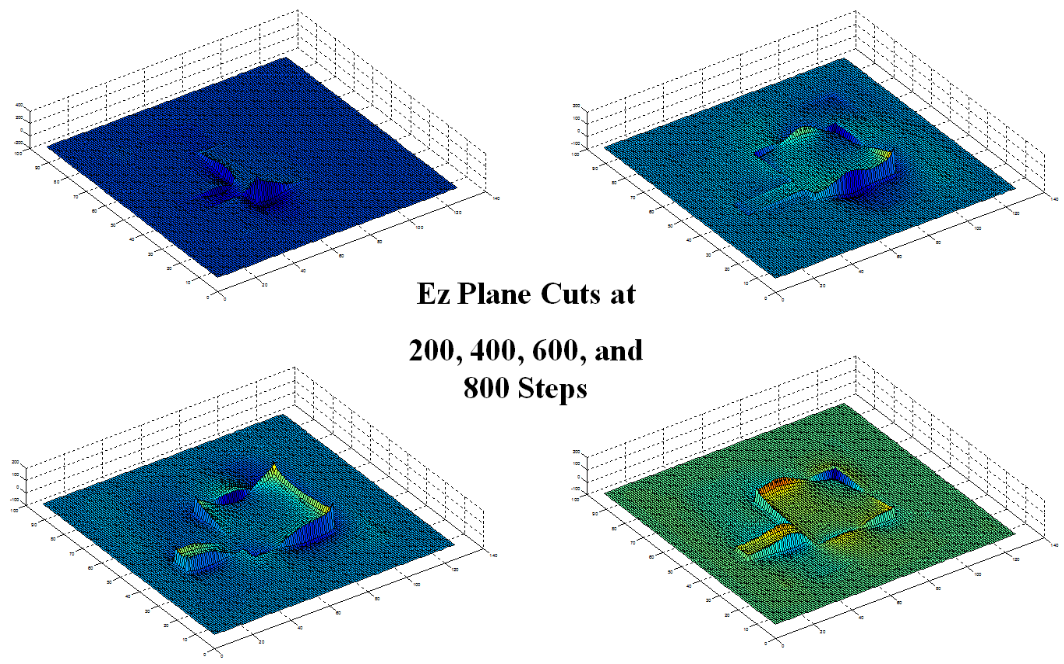


Figure 23. E_z Plane Cuts at various time steps for microstrip validation case

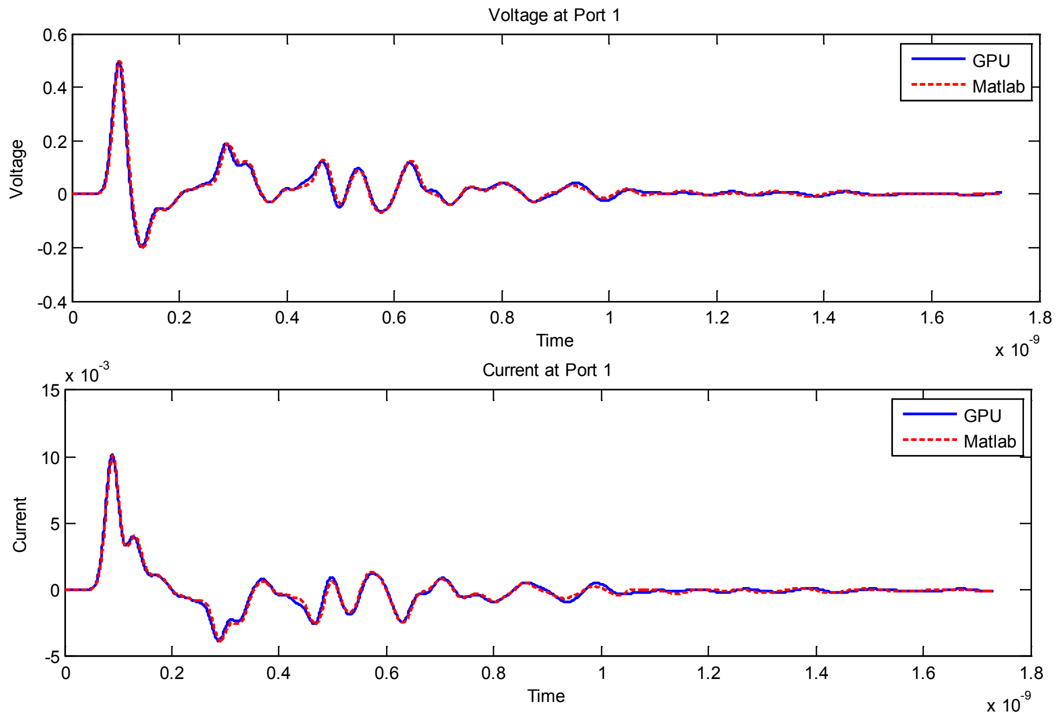


Figure 24. Microstrip patch voltages and currents at the observation port for GPU and CPU codes

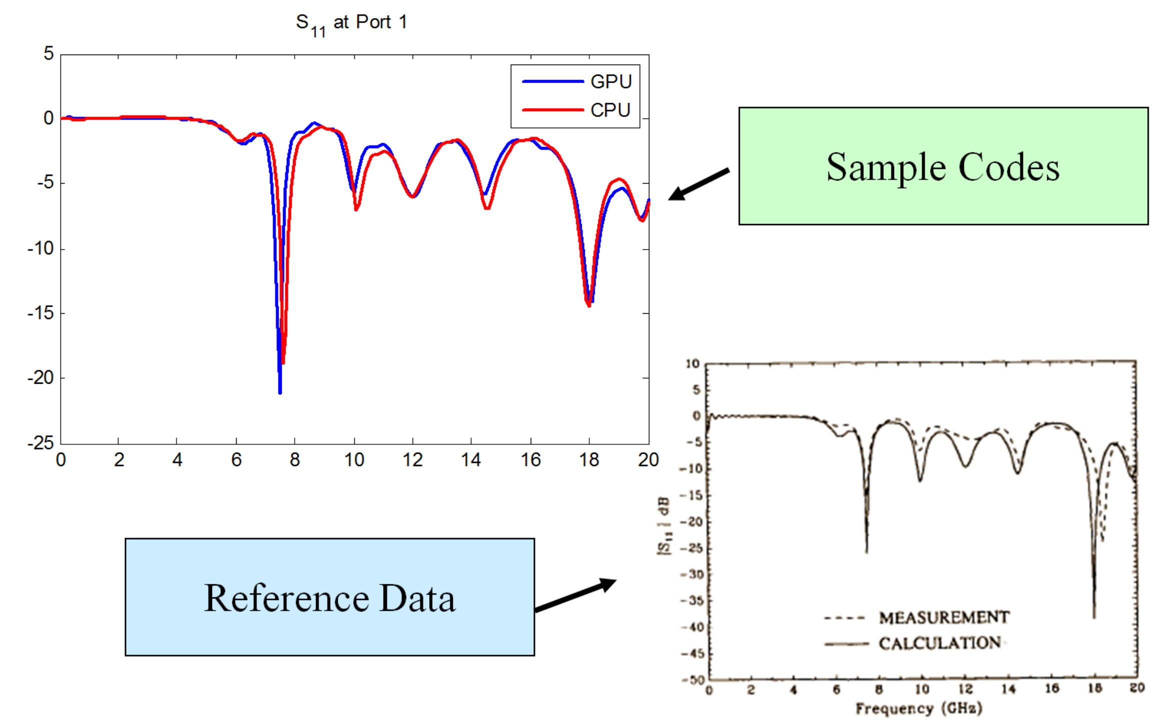
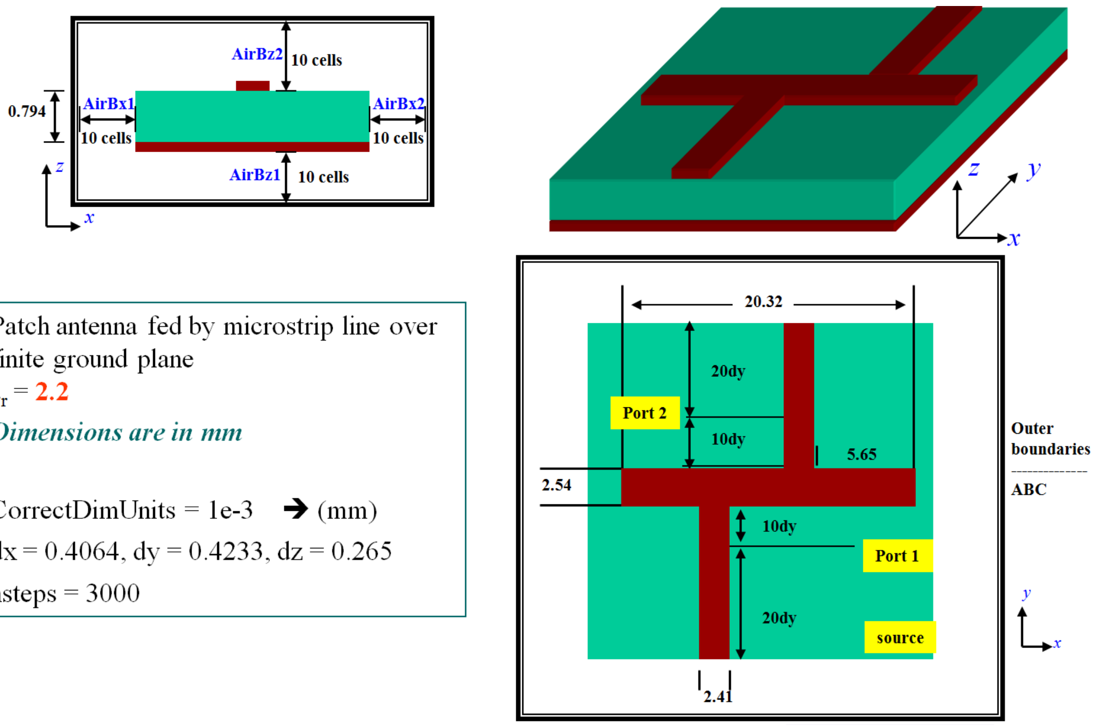


Figure 25. Microstrip patch return loss comparison between GPU, CPU, and reference data

The second case considered is that of a simple microstrip filter as shown in Figure 26. This filter contains two ports to observe both the reflected and transmitted components of the source which is located at the edge of the filter below port 1. Results showing excellent agreement are shown in Figures 27 and 28.



Patch antenna fed by microstrip line over finite ground plane

ε_r = **2.2**

Dimensions are in mm

CorrectDimUnits = 1e-3 ➔ (mm)

dx = 0.4064, dy = 0.4233, dz = 0.265

nsteps = 3000

Figure 26. Microstrip filter validation case

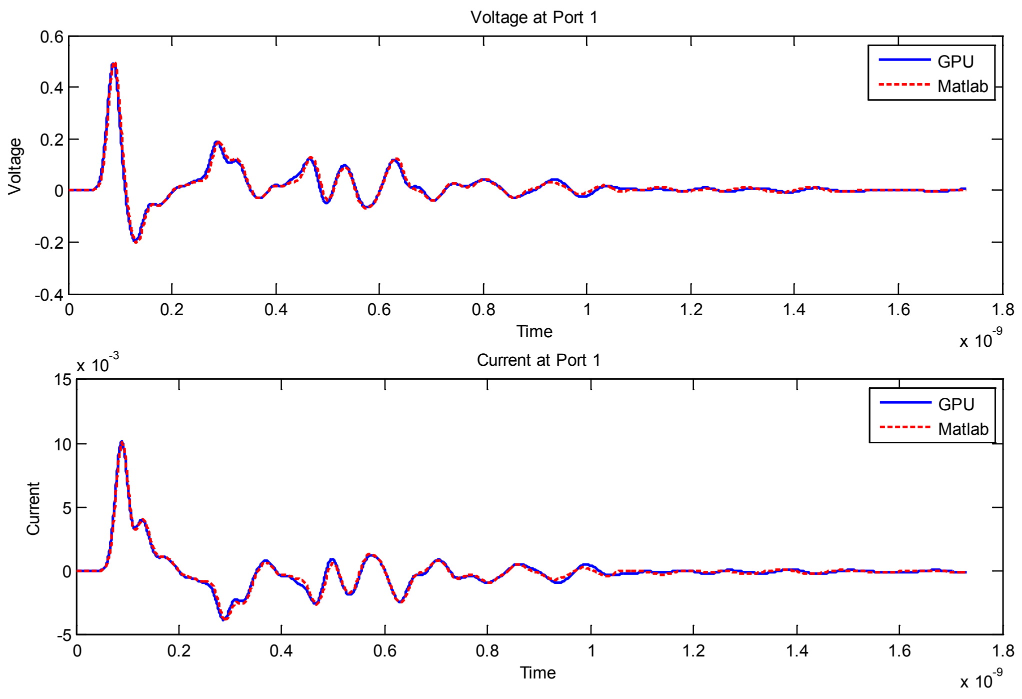


Figure 27. Microstrip filter voltages and currents at the observation port for GPU and CPU codes

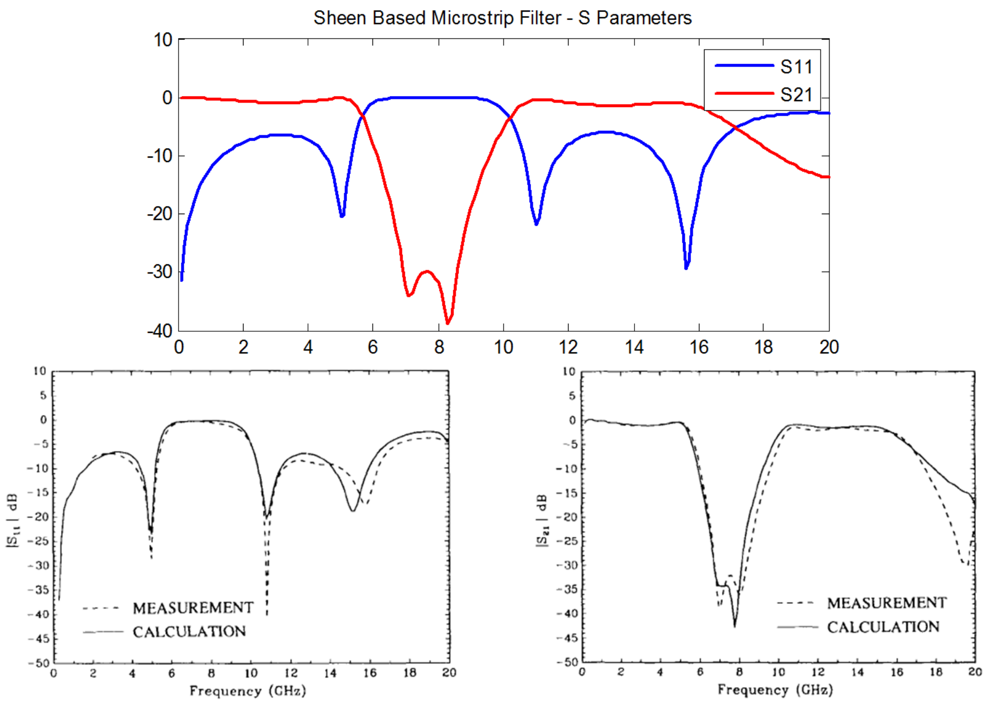


Figure 28. Microstrip filter transmission and return losses comparison

4.5 Finite-Difference Time-Domain Multiple GPU Improvements

One common feature that all GPU's share with CPU's is that at the lowest level, all memory is considered linear. No matter what the method of allocating or accessing the memory, what occurs on the memory chip is the access of a linear memory address space. In 3D FDTD simulations, accessing a cell with a particular (X,Y,Z) coordinate requires either the system or the programmer to translate between the position of the cell in the computational domain and its memory location. CUDA currently offers two distinct types of memory allocation for general device access. These two processes either allocate a straight block of memory defined by the number of elements requested or allocating it padded so that the memory aligns with the hardware requirements. In either case, no matter if the simulation domain is a 1, 2, or 3D array, a linear block of memory will be allocated.

In the case where no padding is requested (cudaMalloc), it is up to the programmer to design how the translation between the linear address and the computational location in the domain is handled. Figure 29 shows a simple situation of a 2x2x2 (NX, NY, NZ=2) domain and how its individual elements are mapped into the linear memory space. This figure shows each of the 8 allocated locations in memory, their locations in the computational domain, and the translation between the two. The case where padded memory is used (cudaMallocPitch/cudaMalloc3D) differs only slightly from the non-padded situation. In padded memory, translation can be automatically provided by CUDA, but the memory required will increase since each "row" might be padded to meet the alignment requirements for coalescing the data.

Neither case presents a clear advantage programming the GPU for FDTD, insofar as they can be functionally equivalent inside the CUDA kernel. These two methods are detailed here so that performance differences between how the FDTD domain is laid out may be explained. All the data presented in this paper will follow this generalized memory layout in that neighboring elements in the X direction will be successive in memory, neighboring elements in the Y direction will be located in memory ±NX away in memory, while neighboring elements in the Z direction will be located ±(NX*NY) in memory.

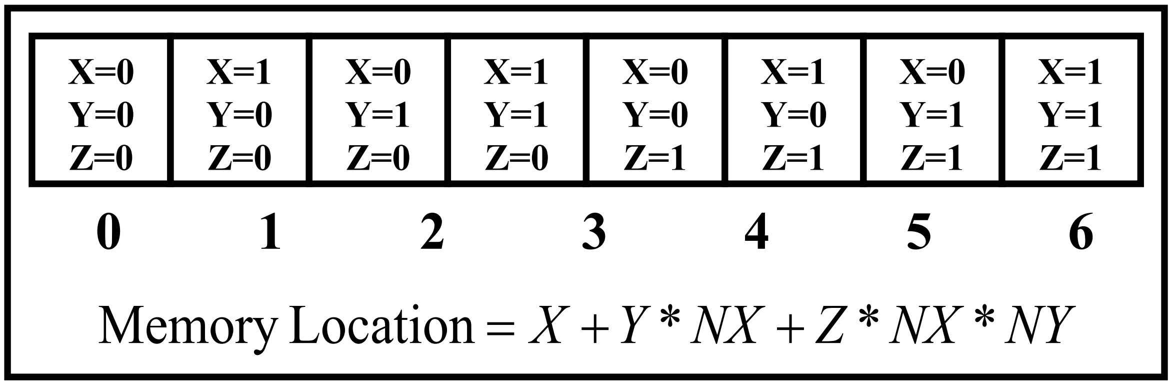


Figure 29. Sample GPU Memory Layout

Previous research has detailed how various ways of configuring the same simulation in memory has impacted the performance in a generic GPU simulation. As an example of this phenomenon, a simple simulation was created with two sides of the domain set for 100 cells and the third varied from 100 to 700 cells. Figure 30 shows the results of this simulation for all 3 cases of either NX, NY, or NZ varied while the other two set at 100 cells. All of the data for this paper was generated on a 2.6 GHz Core i7 machine utilizing two NVIDIA Tesla C1060 GPU's utilizing multiple threads to control the GPU's. The results in this figure show the average time required for one update time step (update E then update H with PEC boundaries). The figure shows that the simulation time is most dependent upon the extent of both the X and Y directions, while the extent of the Z direction has the least effect on simulation time.

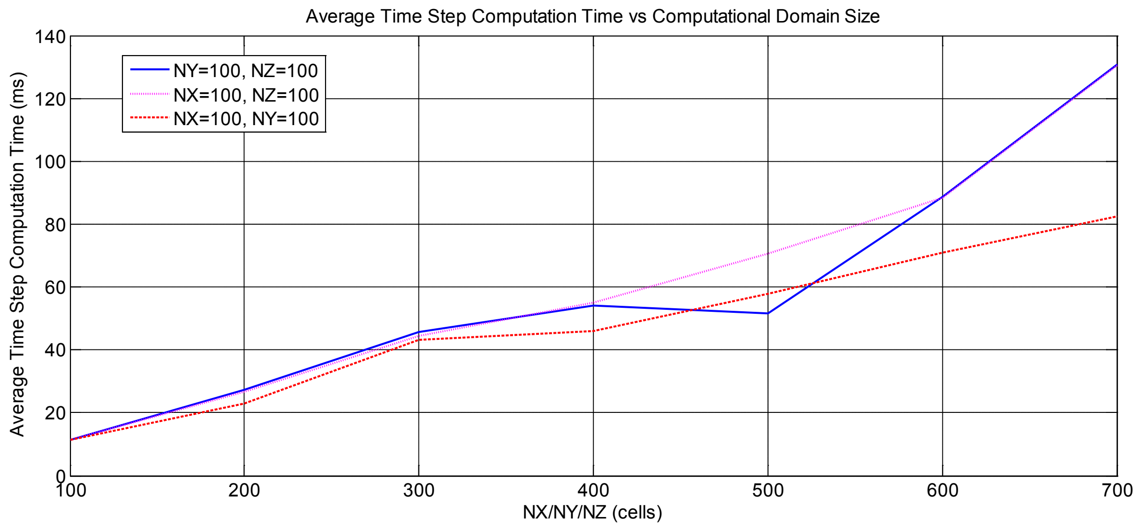


Figure 30. Computation time vs. domain size

These results follow logically if the memory layout is taken into account. FDTD by definition is more a memory intensive technique than a computationally intensive technique. Faster memory rates in any computational device will produce in kind increases in computational speed. Therefore, the most efficient way to optimize any FDTD routine is by making memory access as efficient as possible. This is most noticeable inside a GPU based simulation as the memory access speeds are highly dependent upon the pattern of memory access. The fastest memory transfer speeds are achieved when the memory being accessed is nearest to linear as possible. As the memory locations being accessed grow farther apart, the data transfer rates slow considerably. By expanding the extent in the X direction, memory calls to neighboring cells in the Y direction grow linearly more apart, while memory calls to neighboring cells in the Z direction grow geometrically more distant. When the Y extent is expanded, calls to neighboring cells in X are unaffected while calls to neighboring cells in Z grow linearly more distant. If the Z extent is expanded, there is no difference in the spacing between any memory calls for neighboring cells of any axis. This pattern can be observed in Figure 2 with the line representing an expanding Z extent having the smallest effect on simulation performance.

The large differences in computational performance depending on the orientation of the computational domain increase in significance in multiple GPU simulations. In these simulations the total computational domain in divided up in various sections with each assigned to its own GPU. This follows normal parallelization routines as "ghost cells" are required for certain field components along the dividing boundaries. These "ghost cells" are included in the computation domain for each GPU, but are not updated as they technically belong to a different region. After each full update step, the “ghost cells” must be updated from the other GPU’s and this requires downloading data from each GPU and exchanging this boundary information with other GPU’s and re-uploading the new “ghost cell” data. The easiest way in CUDA to implement this data is to download the field component containing the “ghost cells”, update them, then re-upload the entire field component back to the GPU. Figures 31 and 32 show the effects on performance when an entire single field component is downloaded, updated, and re-uploaded between time steps. This figure shows that in the simplest of cases where only a single field component needs to be updated, this procedure can account for up to 12% of the total computational time. This percentage will grow in more complex simulations as more field components will contain “ghost cells” needing updating. In more complex situations, this data transfer can account for almost 50% of the total simulation time if whole field components are transferred.

In comparison, Figures 33 and 34 show the performance penalties when instead of copying the entire field component, only the section of interest (a single slice along one of the axes) is copied. While CUDA does provide several methods for copying only a portion of an array stored on the GPU, these procedure can often be convoluted when applied to 3D arrays. For comparison, instead of using these methods a separate procedure was developed. This involved using a separate kernel to copy the area of interest in the desired field component to a temporary array in the GPU, then transferring just that data back to the CPU. This process is then reversed in uploading new “ghost cell” data back to the GPU. The addition of invoking these new kernels inside the update steps was negligible to the total runtime (< 0.1%). Figure 4 shows the effects of just copying a single required “ghost” cell data to the CPU and back to the GPU and shows little variance on transfer sizes or orientation. By limiting the data transfers to just the required elements for the exchanging of data, the time required has been minimized. Even in more complex simulations where multiple field components are to be transferred, copying and updating only the cells required limits the performance penalties to under 3%.

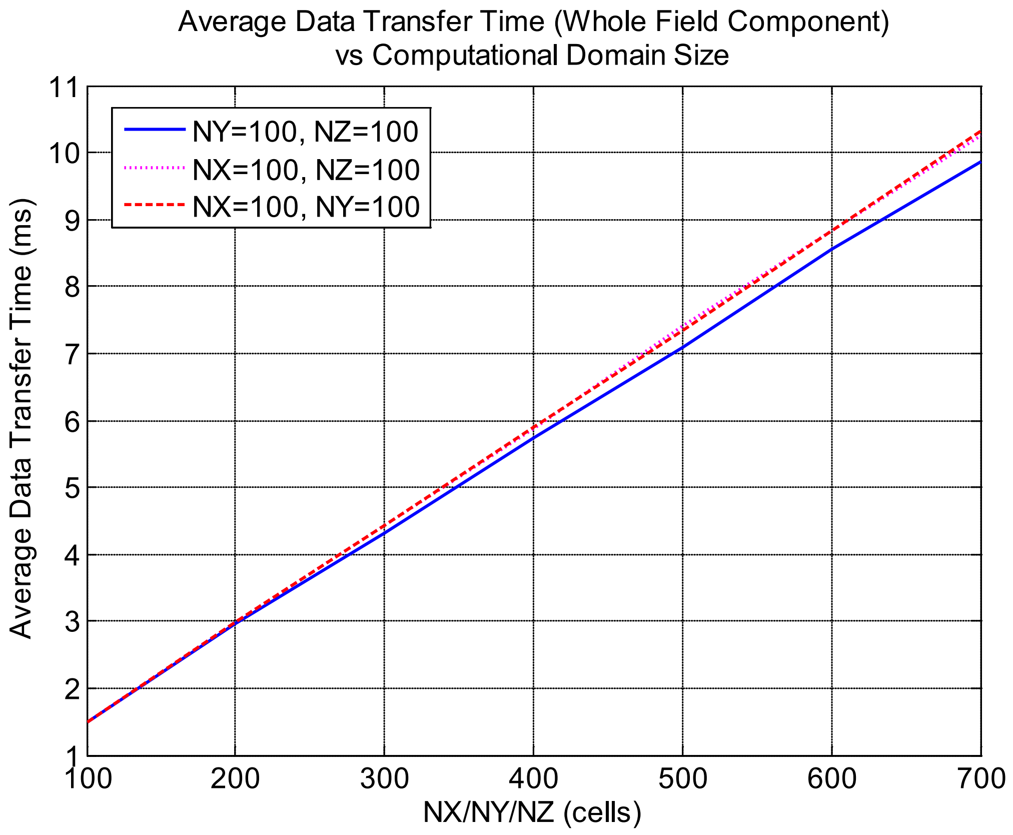


Figure 31. Data transfer times vs computation domain size for whole field components

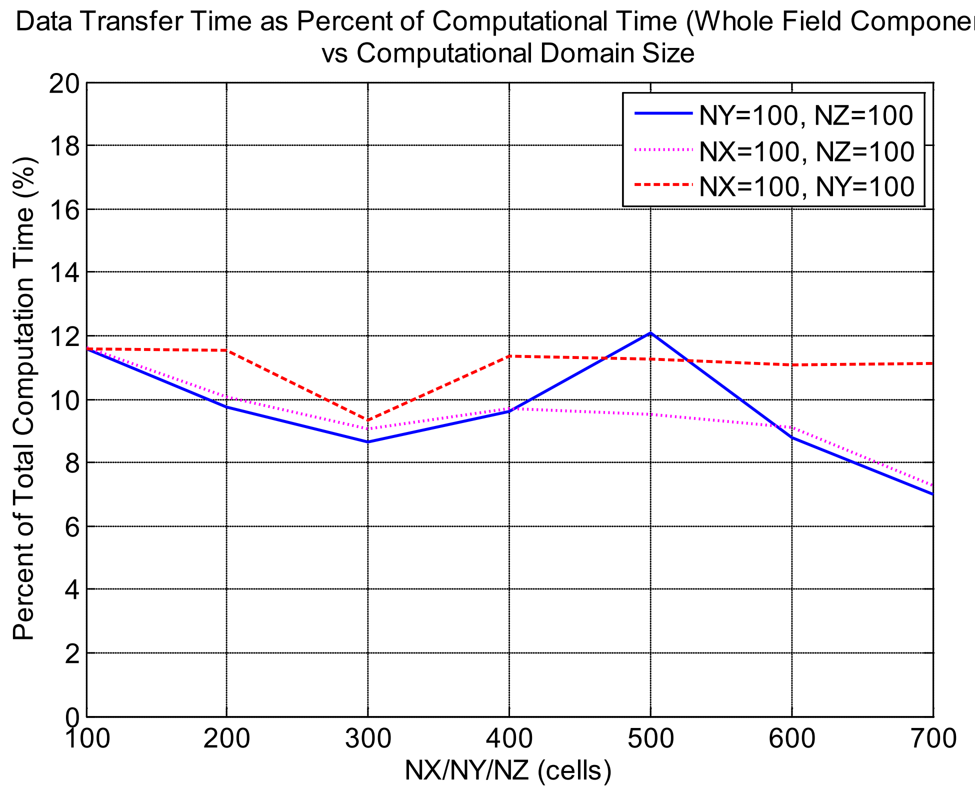


Figure 32. Data transfer times vs computation domain size as a percentage of total computational time for whole field components

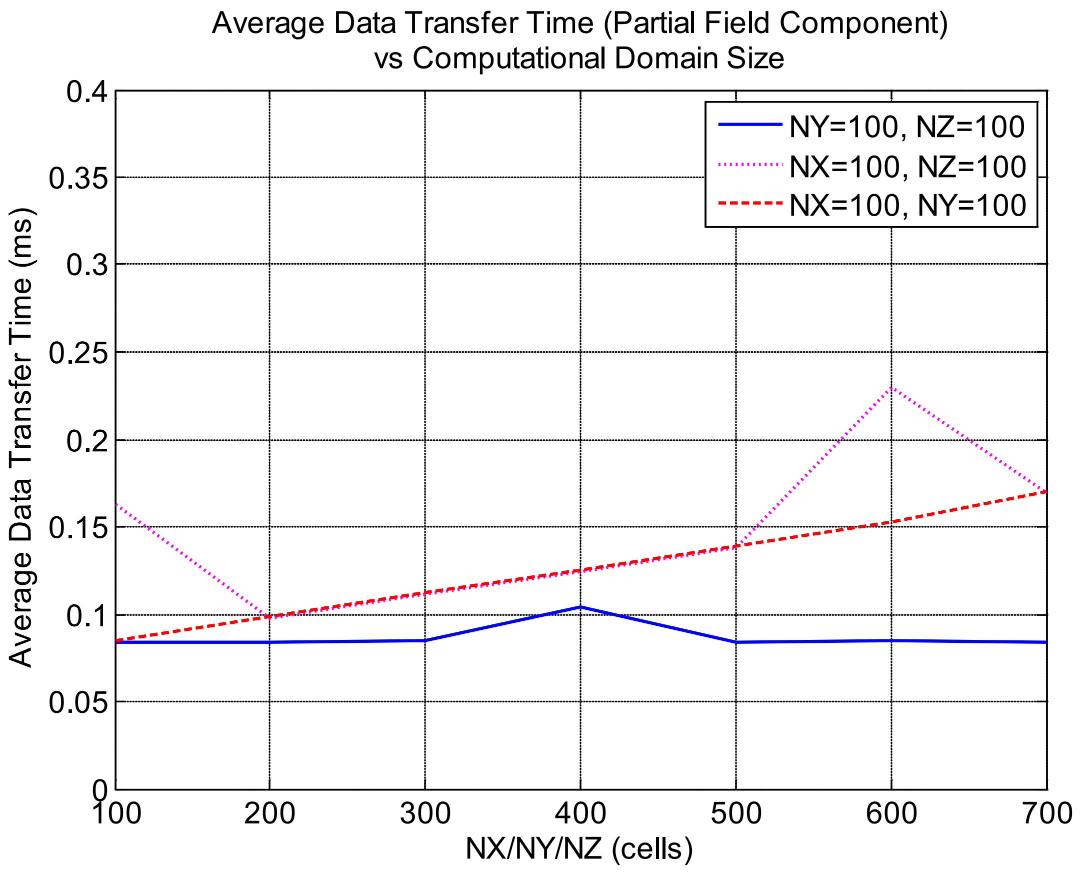


Figure 33. Data transfer times vs computation domain size for partial field components

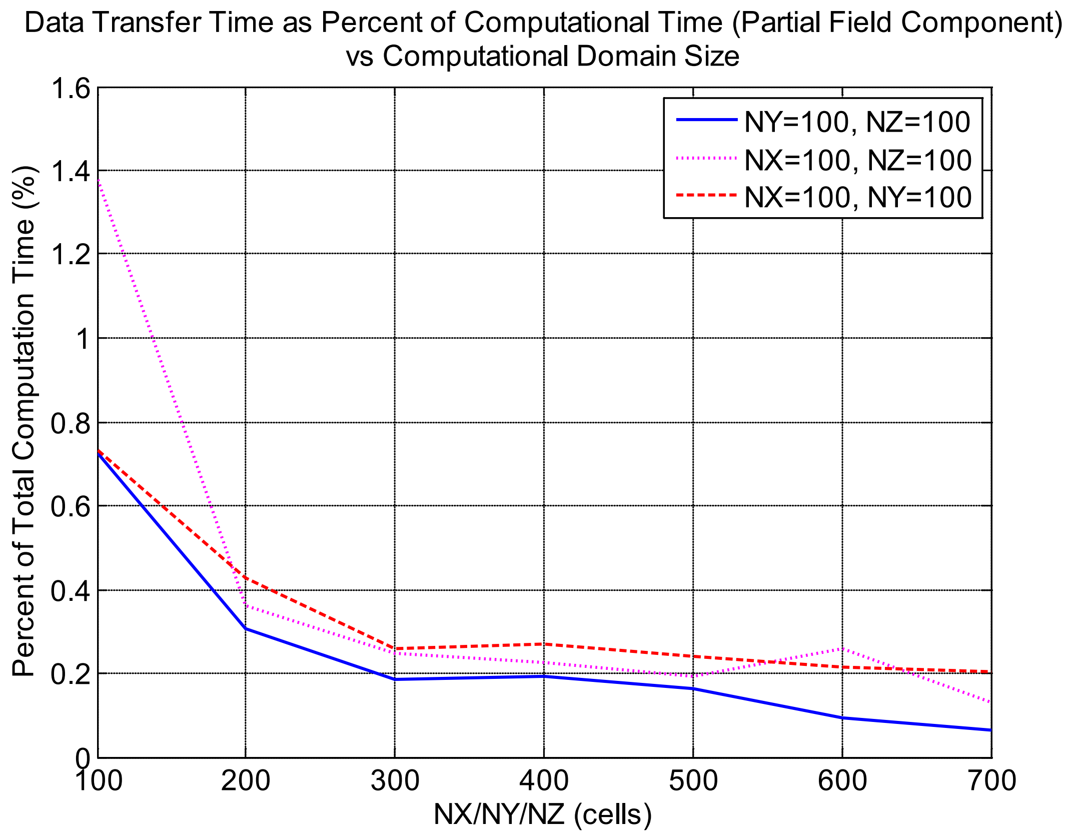


Figure 34. Data transfer times vs computation domain size as a percentage of total computational time for partial field components

Differences in domain orientation on each GPU can have dramatic effects on simulation performance. When dividing up a computational domain for multiple GPU simulations, allowing the Z extent to be the largest can lead up to a 40% increase in per time-step performance over other orientations. Enabling the ghost cell transfers between the different GPU's to be limited to just the areas of interest can minimize the effects of the transfer to almost negligible levels in a single computer system with multi GPU setup. While in more complicated simulations, the transfer times will account for a larger portion of the total simulation time, the effects can be minimized if the domain is decomposed intelligently. Code for these tests may be found in appendices E and F.

4.6 Conclusion

The GPU based FDTD simulator has shown it can accomplish many of the same simulations as their standard CPU based counterparts in a fraction of the time. This will allow it to be used to great effect in many situations where results can be calculated in a matter of seconds instead of several minutes. Such a short simulation time increases its utility in the context of time limited courses for demonstration purposed.

CHAPTER V

TEACHING APPLICATIONS

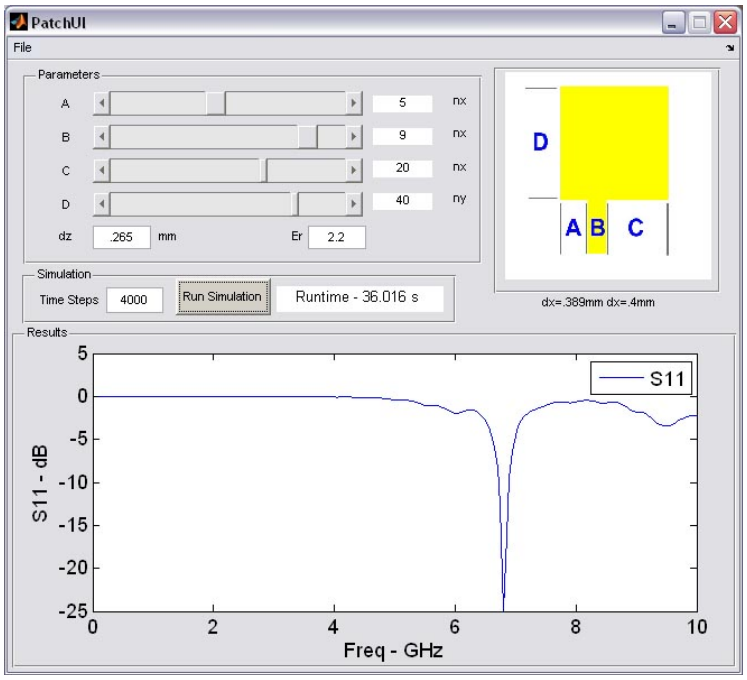
5.1 Introduction

GPU based computational electromagnetics has been shown to significantly increase the execution speed of many various techniques. Both iterative and matrix based simulations have been performed on the GPU with speeds gains ranging from 2 for a double-precision complex LU decomposition, up to over 20 times for a Finite-Difference Frequency-Domain simulation. With the varied techniques gaining speed and choosing the right examples, it is possible to construct GPU based simulations that can be executed in a matter of seconds instead of the minutes or hours before.

In order to be of any use in teaching situations these solvers must fit two distinct criteria; An easy to use interface, and be able to be executed in a reasonable amount of time in a classroom environment. With the basic framework having been detailed, the use of these GPU based solvers can be applied to a wide variety of problems such as antennas, filters, and other electromagnetic devices. The speed gains provided by the GPU based codes open the technique to widespread use in optimization and parameter exploration.

5.2 FDTD Teaching Programs

Previously to now, the running of a simulation using the FDTD method, for example, would take anywhere from a few minutes to many hours or more. As has been show in numerous papers, the execution time for these FDTD simulations can be decreased by up to 25-30 times compared to their CPU based counterparts. The use of the GPU in conjunction with a simple Matlab based graphical user interface can be used in teaching situations. These graphical user interfaces will allow the background GPU code to be connected with an easy to use interface that allows the user to set parameters on a simple simulation and perform the simulation. With the interface, it is possible to vary the constitute parameters of any simulation to show how these parameters effect the operation of the simulation.



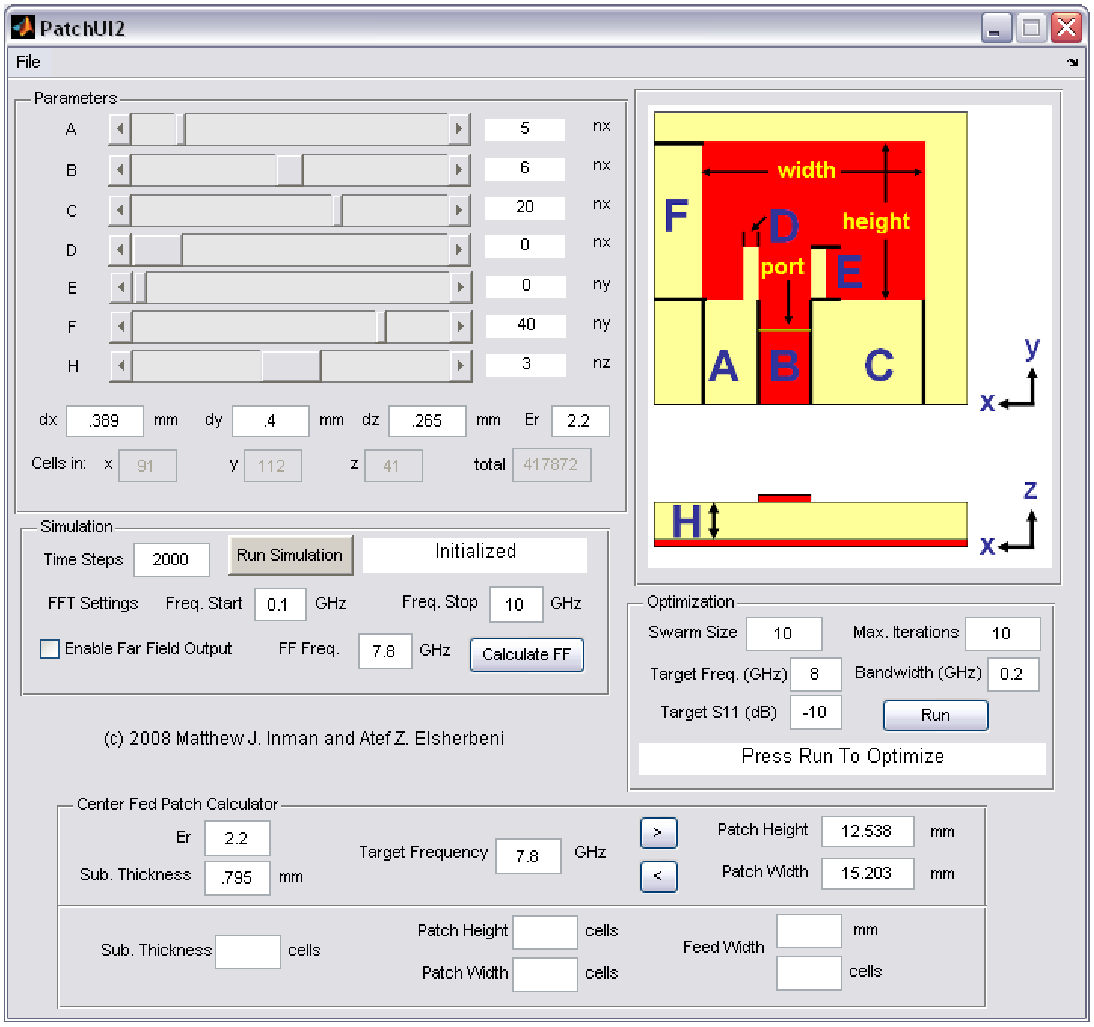
A 5 nx
B 9 nx
C 20 nx
D 40 ny
dz .265 mm Er 2.2
Time Steps 4000
Runtime - 36.016 s
dx=.389mm dx=.4mm

Figure 35. Sample Matlab GUI Interface For Patch Antenna

Figure 35 shows a sample interface created for a GPU based FDTD simulator of a microstrip patch antenna. In this example the user has control over 4 different parameters of this particular antenna. In this case it is a simple microstrip patch antenna. The user may adjust the parameters marked by “A”, “B”, “C”, or “D” to adjust the size of the antenna and the feed lines location and the results be displayed interactively on the bottom. Such a simulation on the CPU would normally take several minutes even for this simple problem, on the GPU however, the simulation time is only a few seconds (11 seconds on the GPU vs 3.5 minutes on the CPU).

Often, especially in cases such as this, while teaching a course covering just such an antenna the instructor will often have to resort to using broad approximations and teaching general rules for how such an antenna would work. Demonstrating how the constituent parameters effect the operation of the device would have been prohibitive if it took several minutes of runtime to calculate a new set of data. With the adoption of the GPU based solvers, since the run time has been reduced significantly, it is possible to demonstrate how each parameter can effect the overall operation of the device. Figure 36 shows a more advanced version of the same application that allows the user to not only simulate given a set of parameters, but also allows synthesis for a starting point of new simulations and optimization options as well. Results from these applications can be had in a matter of seconds rather than minutes with standard CPU based simulations.

In this program a simple particle swarm optimization (PSO) code was implemented on the Matlab portion that drives the simulator to test various combinations of parameters trying to meet the requested output performance. Since the simulation times have been reduced significantly over the CPU simulators, such optimizations can easily be shown even in classroom situations. In cases where simulations only take a matter of seconds, hundreds of executions can easily be performed in the course of a class period.



Parameters

A 5 nx
B 6 nx
C 20 nx
D 0 nx
E 0 ny
F 40 ny
H 3 nz

dx .389 mm dy .4 mm dz .265 mm Er 2.2

Cells in: x 91 y 112 z 41 total 417872

Simulation

Time Steps 2000 Initialized

FFT Settings Freq. Start 0.1 GHz Freq. Stop 10 GHz

☐ Enable Far Field Output FF Freq. 7.8 GHz

(c) 2008 Matthew J. Inman and Atef Z. Elsherbeni

Optimization

Swarm Size 10 Max. Iterations 10

Target Freq. (GHz) 8 Bandwidth (GHz) 0.2

Target S11 (dB) -10

Press Run To Optimize

Center Fed Patch Calculator

Er 2.2

Sub. Thickness .795 mm

Target Frequency 7.8 GHz

Patch Height 12.538 mm

Patch Width 15.203 mm

Sub. Thickness cells

Patch Height cells

Patch Width cells

Feed Width mm

cells

Figure 36. Advanced Matlab GUI Interface For Patch Antenna

In the cases where an optimization might be run, the interface can take the parameters given for the type of optimization and target results and automatically run the simulations given these parameters. In a sample case of this microstrip patch antenna, the optimization system was given the target of -10dB return loss at 7.5 GHz with a swarm size of 10 particles and run over 20 iterations. The GUI will then launch the appropriate simulations, slightly modifying the parameters of the antenna itself from run to run. In this case of 200 separate runs, the total simulation took under 30 minutes with the results shown in figures 37-39.

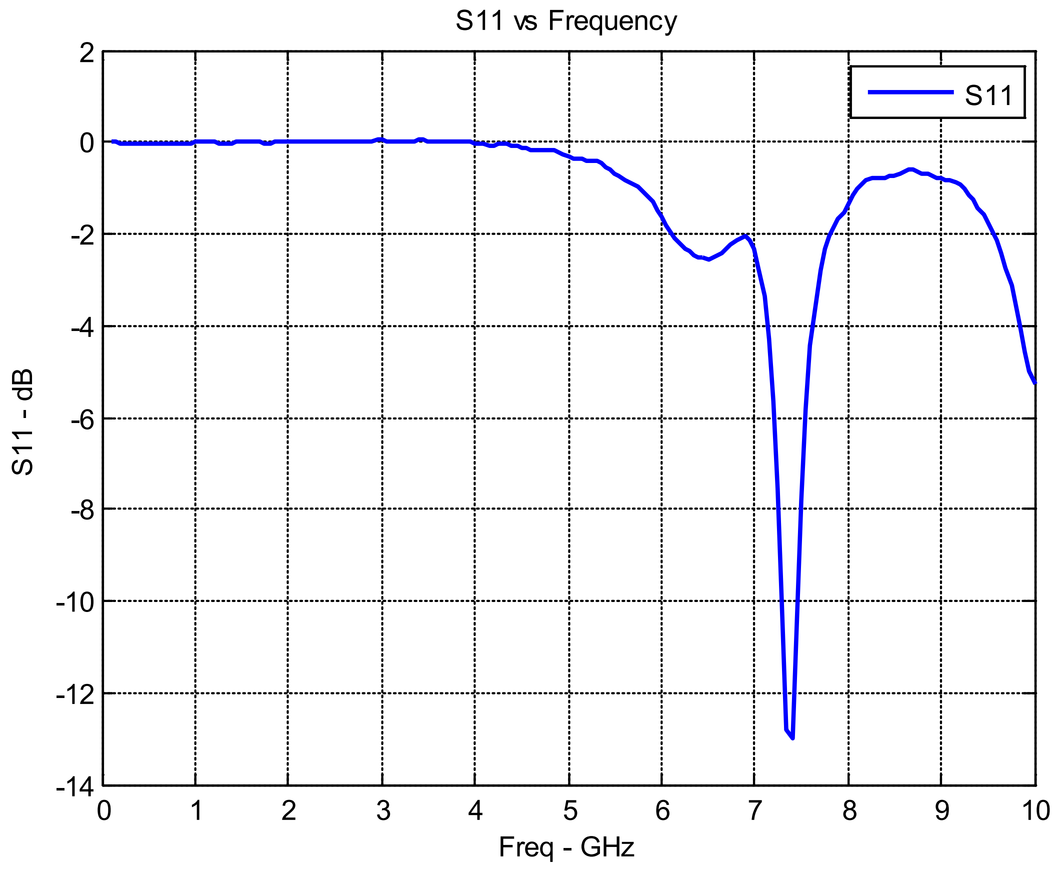


Figure 37. PSO final result from Matlab GUI

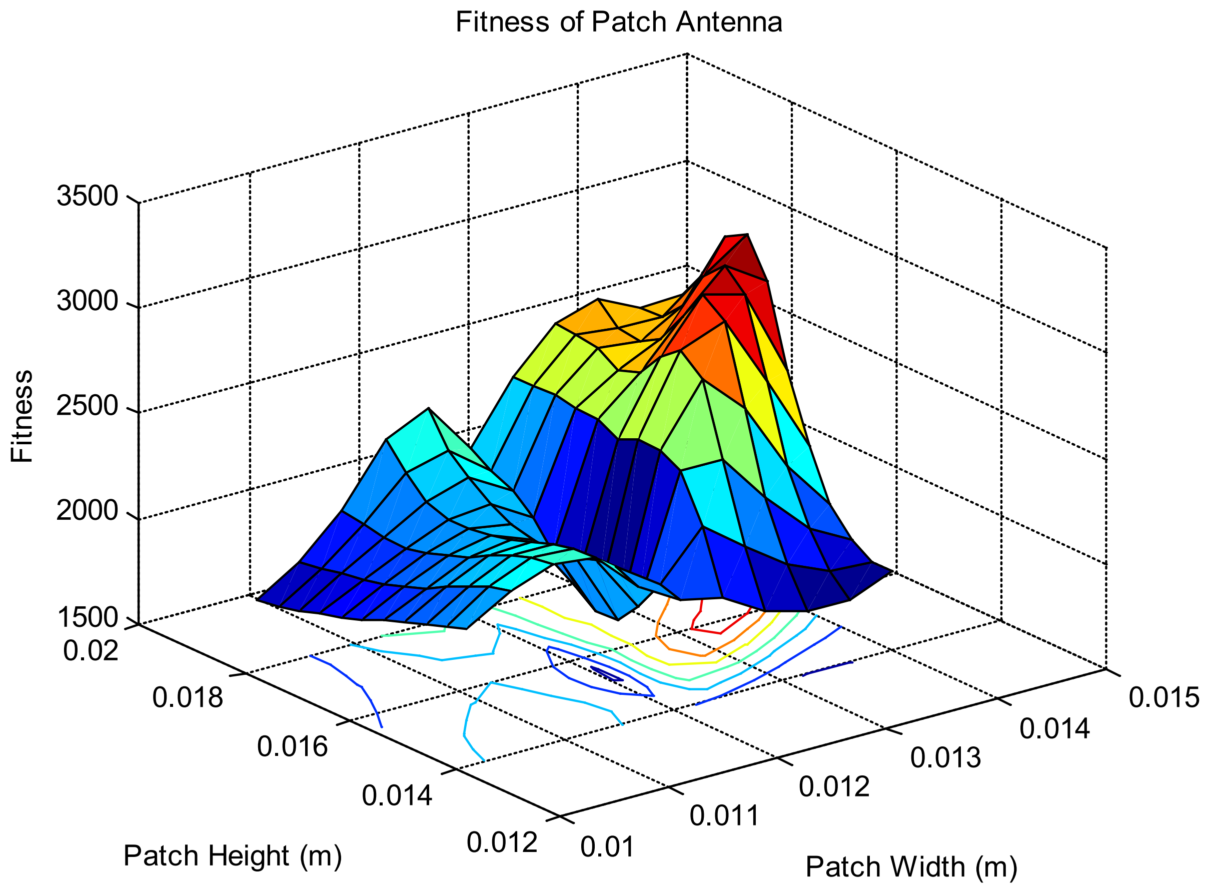


Figure 38. PSO fitness data as the simulator explores the domain

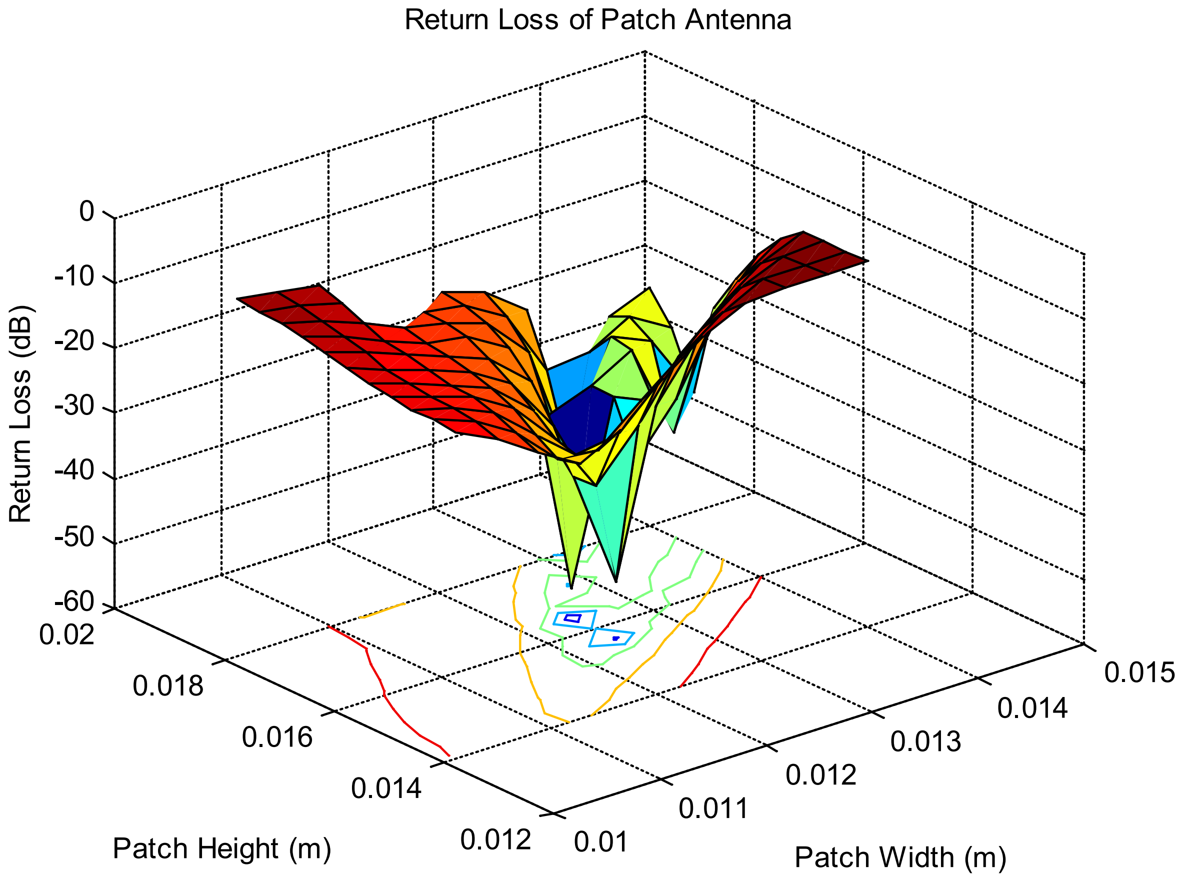
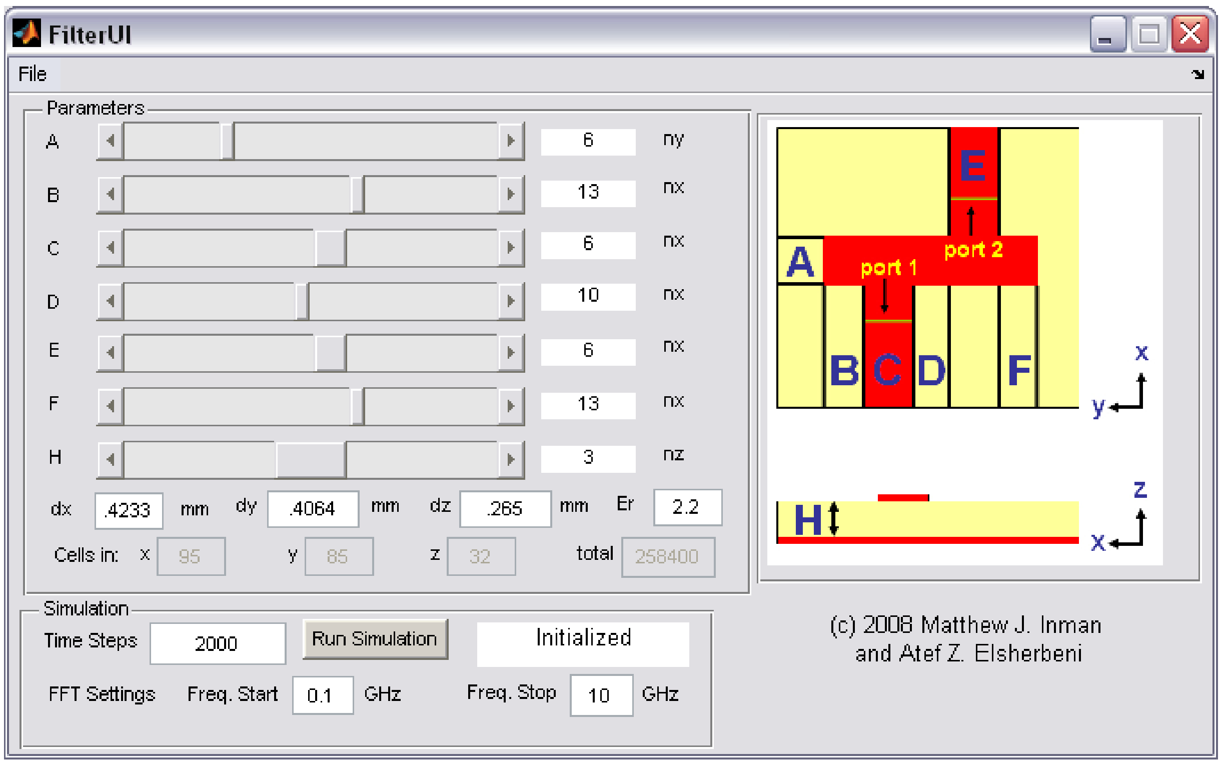


Figure 39. PSO return loss data at target frequency as the simulator explores the domain

Figure 40 shows a second program created for teaching the design of a microstrip filter. In this example the user may adjust any of a wide variety of parameters in the design of the filter not only controlling the shape of the filter but also the material parameters as well. The type of filter shown is commonly taught in design classes as its operation is very well known. Adding the ability the modify the design and have the results available in a very short time (8 seconds for the GPU vs 3 minutes for the CPU) allows the instructor to easily demonstrate how the various parts of the filter effect its operation. Results from this GUI can be seen in Figure 41.



A 6 ny
B 13 nx
C 6 nx
D 10 nx
E 6 nx
F 13 nx
H 3 nz

dx .4233 mm dy .4064 mm dz .265 mm Er 2.2

Cells in: x 95 y 85 z 32 total 258400

Time Steps 2000 Initialized

FFT Settings Freq. Start 0.1 GHz Freq. Stop 10 GHz

(c) 2008 Matthew J. Inman
and Atef Z. Elsherbeni

Figure 40. Sample Matlab GUI Interface For Microstrip Filters

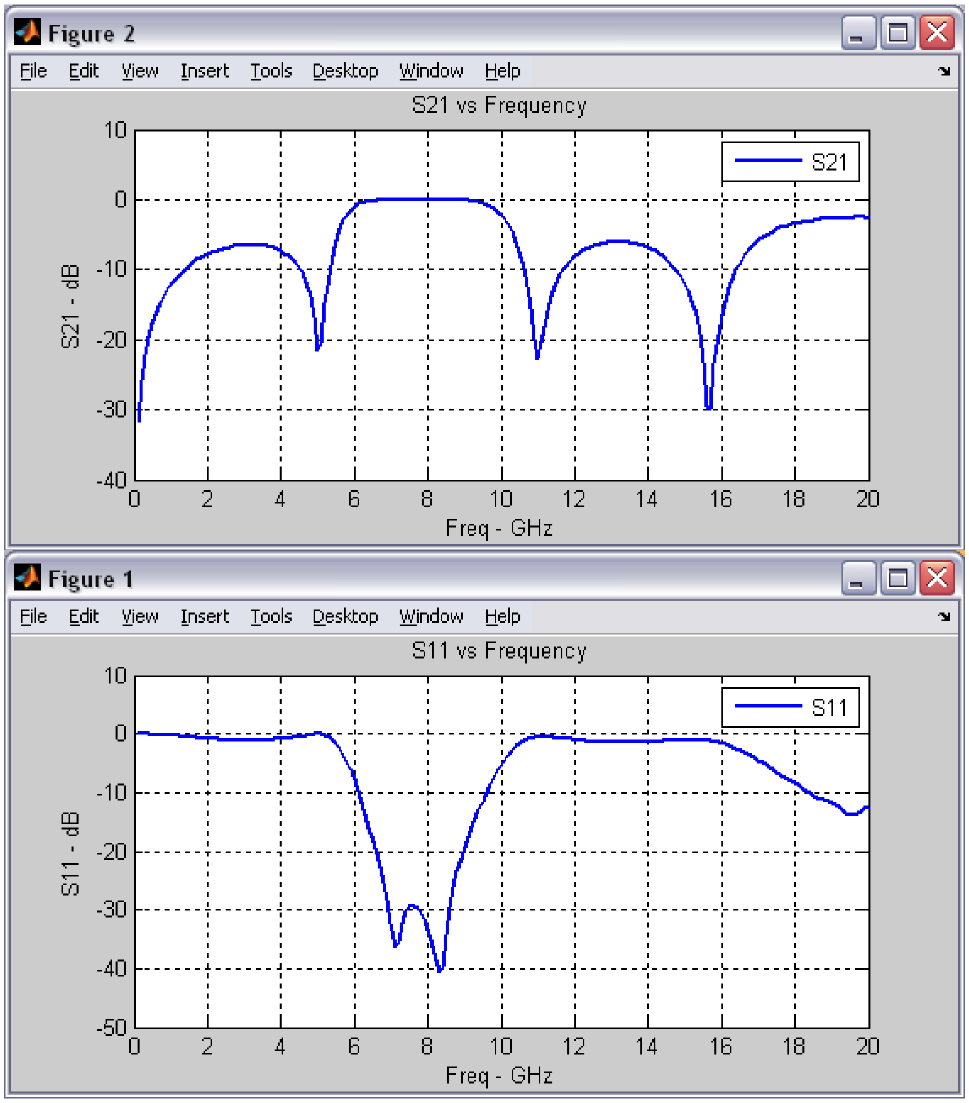
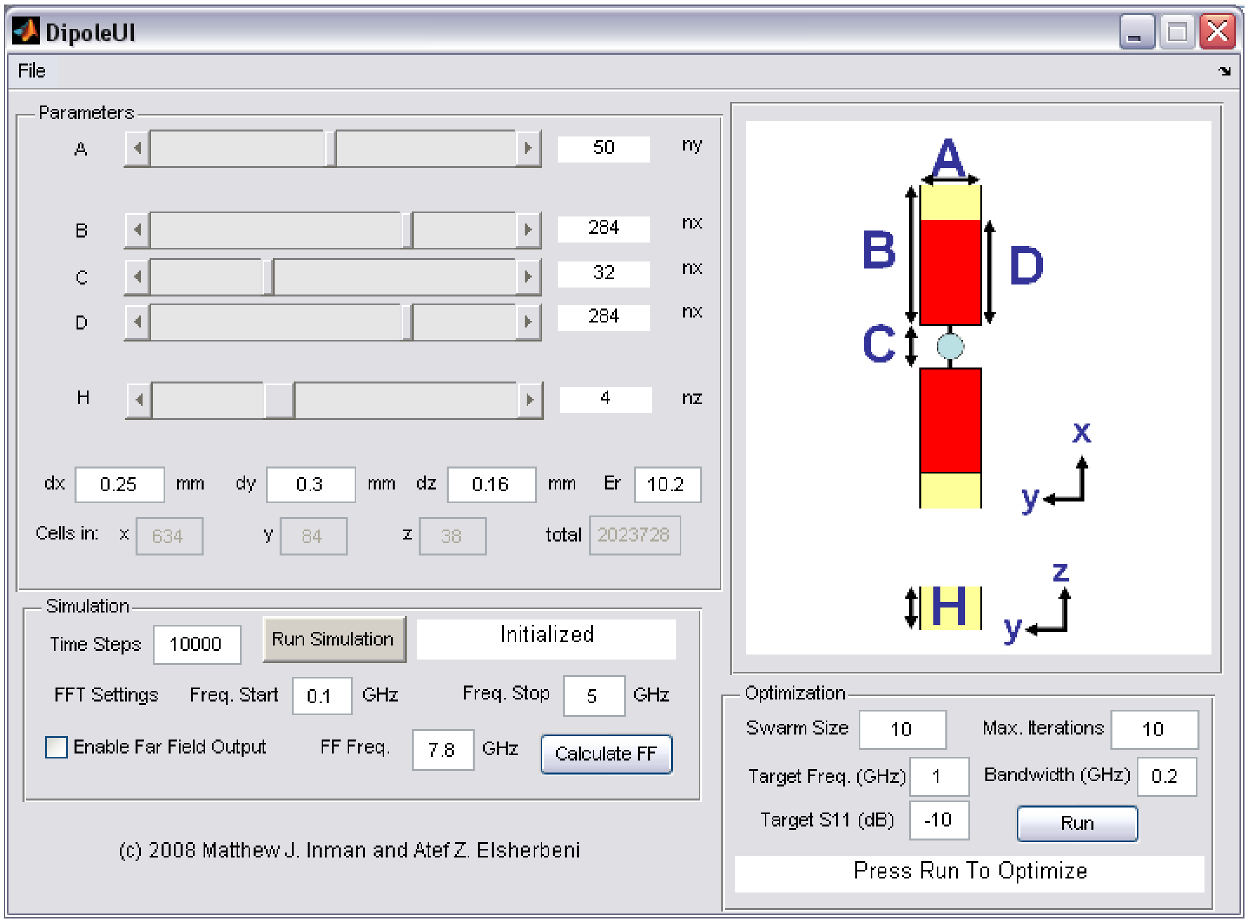


Figure 2
File Edit View Insert Tools Desktop Window Help
S21 vs Frequency
S21
S21 - dB
Freq - GHz
Figure 1
File Edit View Insert Tools Desktop Window Help
S11 vs Frequency
S11
S11 - dB
Freq - GHz

Figure 41. Matlab GUI Interface For Microstrip Filters Results

A third example program is shown in Figure 42. This module allows for the simulation of a printed dipole antenna similar to that one shown in chapter 4. This type of antenna is very common in antenna courses as well due to its simple nature. The interface allows the user to adjust various any of the size and material parameters of the antenna and includes an addition component of optimization.



Parameters

A 50 ny

B 284 nx

C 32 nx

D 284 nx

H 4 nz

dx 0.25 mm dy 0.3 mm dz 0.16 mm Er 10.2

Cells in: x 634 y 84 z 38 total 2023728

Simulation

Time Steps 10000 Initialized

FFT Settings Freq. Start 0.1 GHz Freq. Stop 5 GHz

☐ Enable Far Field Output FF Freq. 7.8 GHz

(c) 2008 Matthew J. Inman and Atef Z. Elsherbeni

Optimization

Swarm Size 10 Max. Iterations 10

Target Freq. (GHz) 1 Bandwidth (GHz) 0.2

Target S11 (dB) -10

Press Run To Optimize

Figure 42. Sample Matlab GUI Interface For Printed Dipole Antennas

Similarly there are many more opportunities where GPU based solvers can be used in classroom environments. Many various topics in electromagnetics often require students and instructors to use complex simulations to solve problems, from antenna analysis, to object scattering, and even circuit design at high frequencies. Often the instructor will demonstrate the use of the simulator and assign the actual problems to be solved as homework due to the long execution times needed.

Allowing a wide variety of simulations to be performed in class would allow the instructor to better communicate and demonstrate how these devices being examined operate. Being able to interactively show the effects of how these devices operate will allow the students a greater understanding of the class material.

CHAPTER VI

CONCLUSIONS

The use of GPU based computational electromagnetic simulations have shown in a wide variety of applications to significantly speedup computational time. In both cases of matrix solving and iterative methods, computational time can be drastically reduced. Depending upon the application matrix solving CEM applications can be speed up anywhere from two times for complex double-precision solves up to over ten times for real single-precision solves. In the case of iterative methods such as FDTD the speed gains are even more significant often resulting in speedups over twenty times faster than their CPU counterparts.

The reduced computational time available by utilizing GPU based simulations allows for their integration in a wide variety of teaching environment beyond their traditional roles in homework and numerical methods classes. While there does exist a few commercial products utilizing the computational power of the GPU for simulation purposes, these are often both expensive and narrowly focused for large commercial simulators. Utilizing these packages in classroom environments becomes unwieldy for simple uses such as demonstrating how small changes in common antennas and devices affect their performance.

Presented here are a few small packages that can be easily adapted and used in teaching situations. With easily usable interfaces these modules as present can be simply run in the classroom to demonstration easily and quickly the operation of electromagnetic devices such as antennas and filters without the need for expensive and unwieldy simulators. By creating several GPU simulators for most common simulation types, a wide variety of modules can be crafted to fit seamlessly in various electromagnetic courses to aid in the teaching and understanding of the students.

BIBLIOGRAPHY

Atef Elsherbeni, Veysel Demir, and Fan Yang, FDTD Electromagnetic Simulations Using Matlab, SciTech Publishing Inc., 2009.

J. A. Roden, S. D. Gedney, “Convolutional PML (CPML): An Efficient FDTD Implementation of the CFS-PML for Arbitrary Media,” Microwave Optical Tech. Let., Vol. 27, pp. 334-339, 2000.

Ian Buck, "Brook Spec v0.2", Stanford University. 2003

K. Fatahalian, et al, “Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication”, Stanford University, 2004.

D. Luebke, et al, “General-Purpose Computation on Graphics Hardware”, Workshop, SIGGRAPH, 2004.

S. D. Gedney, “Perfectly Matched Layer Absorbing Boundary Conditions,” in Computational Electrodynamics: The Finite Difference Time Domain, third edition, A.

Taflove, and Susan C. Hagness, Editors, Artech House, Norwood, MA, pp. 295-313, 2005.

K. S. Yee, “Numerical solution of initial boundary value problems involving Maxwell’s equations in isotropic media,” IEEE Trans. Antenna Propagat., Vol. AP-14, pp. 302-307, May 1966.

S. E. Krakiwsky, M. Okoniewski, and L. E. Turner, “Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPUs)”, Proc. Intl. Microwave Symp., Fort Worth, TX, 2004.

V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra, SC08

N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, Proceedings of the 2005 ACM/IEEE conference on Supercomputing.

E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. Mckenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, And D. Sorensen, 1990. LAPACK: a portable linear algebra library for high-performance computers, Supercomputing ’90.

David M. Sheen, Sami M. Ali, Mohamed D. Abouzahra, and Jin Au Kong
IEEE Transactions on Microwave Theory and Techniques, Vol 37, No 7, July 1990
Pages 849-857

M. Baboulin, J. Dongarra, and S. Tomov. Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures, LAPACK Working Note 200.

CUDA User Forums, http://forums.nvidia.com

APPENDIX A

```
/*******************************************************************
*  MatInv.cu
*  Gaussian Matrix Inversion
*  Matthew Inman
********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil.h>
#include <math.h>

#define BLOCKSIZE 16

/***********************************************************************/
/* CUDA Kernels                                                        */
/***********************************************************************/

__global__ void GPUsetIdentity (float2* matrix,
                                int width)
{
    int tx = (blockIdx.x * blockDim.x + threadIdx.x);
    int ty = (blockIdx.y * blockDim.y + threadIdx.y);

// Set Imaginary Part to 0
    matrix[ty  *width + tx].y = 0;

// Set Real Part
    if (tx==ty)
               matrix[ty * width + tx].x = 1;
    else
               matrix[ty * width + tx].x = 0;
}


//--------------------------------------------------------------------------------


__global__ void pivotBlock_kernel (float2 *dInData, float2 *dInDataInv,float2 *dInData2, int
loop, int size)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;

       if (ty == loop) {
           float a = dInData[ty * size + tx].x;
           float b = dInData[ty * size + tx].y;
           float c = dInData2[loop * size + loop].x;
           float d = dInData2[loop * size + loop].y;
           float a1 = dInDataInv[ty * size + tx].x;
           float b1 = dInDataInv[ty * size + tx].y;


           dInData[ty * size + tx].x = (a*c + b*d)/(c*c+d*d);
           dInData[ty * size + tx].y = (b*c - a*d)/(c*c+d*d);

           dInDataInv[ty * size + tx].x = (a1*c + b1*d)/(c*c+d*d);
           dInDataInv[ty * size + tx].y = (b1*c - a1*d)/(c*c+d*d);

       }
}


//--------------------------------------------------------------------------------
```

```
__global__ void divBlock_kernel (float2 *dInData, float2 *dInDataInv,float2 *dInData2, int loop,
int size)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;

       if (ty != loop) {
               float a = dInData[loop * size + tx].x;
           float b = dInData[loop * size + tx].y;

           float c = dInData2[ty * size + loop].x;
           float d = dInData2[ty * size + loop].y;

           float a1 = dInDataInv[loop * size + tx].x;
           float b1 = dInDataInv[loop * size + tx].y;

           float e = dInData2[ty * size + tx].x;
           float f = dInData2[ty * size + tx].y;

           float g = dInDataInv[ty * size + tx].x;
           float h = dInDataInv[ty * size + tx].y;



             dInData[ty * size + tx].x = e - (a*c - b*d);
             dInData[ty * size + tx].y = f - (b*c + a*d);

             dInDataInv[ty * size + tx].x = g - (a1*c - b1*d);
             dInDataInv[ty * size + tx].y = h - (b1*c + a1*d);

       }
}

//--------------------------------------------------------------------------------


__global__ void copy_kernel (float2 *dInData,float2 *dInData2, int size)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;

       dInData[ty * size + tx].x = dInData2[ty * size + tx].x;
       dInData[ty * size + tx].y = dInData2[ty * size + tx].y;
}



/************************************************************************/
/* Init CUDA                                                            */
/************************************************************************/
#if __DEVICE_EMULATION__

bool InitCUDA(void){return true;}

#else
bool InitCUDA(void)
{
       int count = 0;
       int i = 0;

       cudaGetDeviceCount(&count);
       if(count == 0) {
               fprintf(stderr, "There is no device.\n");
               return false;
       }

       for(i = 0; i < count; i++) {
```

```
                cudaDeviceProp prop;
                if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
                        if(prop.major >= 1) {
                                break;
                        }
                }
        }
        if(i == count) {
                fprintf(stderr, "There is no device supporting CUDA.\n");
                return false;
        }
        cudaSetDevice(i);

        printf("CUDA initialized.\n");
        return true;
}

#endif


/************************************************************************/
/* Main Program                                                         */
/************************************************************************/
int main(int argc, char* argv[])
{

        if(!InitCUDA()) {
                return 0;
        }


    int i, j,k;

    // Matrix Size (NxN)
    int size = 32;



    // Initialize CPU/GPU Memory
    float *dataInput = (float*) malloc (sizeof (float) * size * size * 2);
    float *resultGPU = (float*) malloc (sizeof (float) * size * size * 2);
    float2 *dDataIn;
    float2 *dDataIn2;
    float2 *dDataInv;

    int size2InBytes = size * size * sizeof (float2);

    float f,f2;
    FILE* pFile;

    //Allocating memory for the datamatrix and identity matrix (Einheitsmatrix)
    if (cudaMalloc ((void **) &dDataIn, size2InBytes) != cudaSuccess) {
               printf("cudaMalloc1 Failed for %d bytes", size2InBytes);
               return 0;
    }
    if (cudaMalloc ((void **) &dDataIn2, size2InBytes) != cudaSuccess) {
               printf("cudaMalloc2 Failed for %d bytes", size2InBytes);
               return 0;
    }
    if (cudaMalloc ((void **) &dDataInv, size2InBytes) != cudaSuccess) {
               printf("cudaMalloc3 Failed for %d bytes", size2InBytes);
               return 0;
    }



// ---  Set Thread Sizes for GPU
```

```
        dim3 dimBlock(BLOCKSIZE, BLOCKSIZE);
        dim3 dimGrid((size) / dimBlock.x, size / dimBlock.y);


// --- Read Data File


pFile = fopen ("input.txt","r");
    k=0;
        for (i = 0; i < size; i++)
    {
       for (j = 0; j < size*2; j=j+2)
       {
                        fscanf (pFile, " (%E,%E)\n", &f, &f2);
                        dataInput[k] =  (float) f;
                        dataInput[k+1] =  (float) f2;
                        k=k+2;
       }
    }


fclose(pFile);

// --- Initialize Timers
        unsigned int timer = 0;
        CUT_SAFE_CALL( cutCreateTimer( &timer));
        CUT_SAFE_CALL( cutStartTimer( timer));

// ------------- Run The Kernel

//Prepare the calculation of the identitymatrix

    cudaMemset ((void *) dDataInv, 0, size2InBytes);

//Transfer the matrix from host to device

    cudaMemcpy ( dDataIn, dataInput, size2InBytes, cudaMemcpyHostToDevice);
    cudaMemcpy ( dDataIn2, dataInput, size2InBytes, cudaMemcpyHostToDevice);


//Calculate the Identitymatrix

    GPUsetIdentity <<< dimGrid, dimBlock >>> (dDataInv, size);
    cudaThreadSynchronize ();


// Loop over size (N)
   for (i=0; i<size; i++) {
        pivotBlock_kernel <<< dimGrid, dimBlock >>> (dDataIn,dDataInv, dDataIn2, i, size);
        cudaThreadSynchronize ();

        copy_kernel <<< dimGrid, dimBlock >>> (dDataIn2,dDataIn, size);
        cudaThreadSynchronize ();

        divBlock_kernel <<< dimGrid, dimBlock >>> (dDataIn,dDataInv, dDataIn2, i, size);
        cudaThreadSynchronize ();

        copy_kernel <<< dimGrid, dimBlock >>> (dDataIn2,dDataIn, size);
        cudaThreadSynchronize ();


        if (i%10==0)
                printf("Iteration %d \n", i);

    }
```

```
// Copy Result Back to CPU
  cudaMemcpy ((void *) resultGPU, (void *) dDataInv, size2InBytes, cudaMemcpyDeviceToHost);




// ----------------------------
        CUDA_SAFE_CALL( cudaThreadSynchronize() );
        CUT_SAFE_CALL( cutStopTimer( timer));
        printf("Processing time: %f (ms)\n", cutGetTimerValue( timer));
        CUT_SAFE_CALL( cutDeleteTimer( timer));


// -- Free GPU Memory
        cudaFree (dDataIn);
        cudaFree (dDataIn2);
        cudaFree (dDataInv);




// -- Write Results Out To File
pFile = fopen ("output.txt","wt");
k=0;
    for (i = 0; i < size; i++)
    {
        for (j = 0; j < size*2; j=j+2)
        {
                       fprintf (pFile, "%f %f\n", resultGPU[k], resultGPU[k+1]);
                       k=k+2;
        }
    }
fclose(pFile);


// -- Free CPU Memory
        free(resultGPU);
        free(dataInput);


// Shutdown
        CUT_EXIT(argc, argv);

        return 0;
}
```

APPENDIX B

```
//

// CuBLAS Implementation for Double Precision Complex
// Matthew J. Inman
// Adapted From Single Precision Real Documentation by:  Vasily Volkov
//

#include "gpu_lapack_internal.h"


//
//  Symmetric rank k update
//  See http://www.netlib.org/blas/ssyrk.f
//
extern "C" void gpu_ssyrkLN( int n, int k, double alpha2, const p2_t A, double beta2, p2_t C )
{

        cuDoubleComplex alpha, beta;

        alpha.x = alpha2;
        alpha.y = 0;

        beta.x  = beta2;
        beta.y  = 0;

        if( n <= 0 || k <= 0 )
                return;

        cublasZsyrk( 'L', 'N', n, k, alpha, A.A, A.lda, beta, C.A, C.lda );
    Q( cublasGetError( ) );
}

//
//  Matrix-matrix multiplications
//  See http://www.netlib.org/blas/sgemm.f
//
extern "C" void gpu_sgemmNN( int m, int n, int k, double alpha2, const p2_t A, const p2_t B,
double beta2, p2_t C )
{
        cuDoubleComplex alpha, beta;

        alpha.x = alpha2;
        alpha.y = 0;

        beta.x  = beta2;
        beta.y  = 0;
        if( m <= 0 || n <= 0 || k <= 0 )
                return;
    cublasZgemm( 'N', 'N', m, n, k, alpha, A.A, A.lda, B.A, B.lda, beta, C.A, C.lda );
    Q( cublasGetError( ) );

}

extern "C" void gpu_sgemmNT( int m, int n, int k, double alpha2, const p2_t A, const p2_t B,
double beta2, p2_t C )
{
        cuDoubleComplex alpha, beta;

        alpha.x = alpha2;
        alpha.y = 0;

        beta.x  = beta2;
        beta.y  = 0;


        if( m <= 0 || n <= 0 || k <= 0 )
                return;
    cublasZgemm( 'N', 'T', m, n, k, alpha, A.A, A.lda, B.A, B.lda, beta, C.A, C.lda );
    Q( cublasGetError( ) );

}
```

APPENDIX C

```
//
// CuBLAS Implementation for Double Precision Complex
// Transpose routines
// Matthew J. Inman
// Adapted From Single Precision Real Documentation by:  Vasily Volkov
//

#include "gpu_lapack_internal.h"

#define BLOCK_SIZE 4 // hand-tuned parameter

static __global__ void transpose_device( cuDoubleComplex *dst, int ldd, cuDoubleComplex *src, int
lds )
{
       src += blockIdx.x*16 + threadIdx.x + ( blockIdx.y*16 + threadIdx.y ) * lds;
       dst += blockIdx.y*16 + threadIdx.x + ( blockIdx.x*16 + threadIdx.y ) * ldd;

       __shared__ cuDoubleComplex a[16][17];

    //
    //  load 32x32 block
    //
       for( int i = 0; i < 16; i += BLOCK_SIZE )
       {
              a[i+threadIdx.y][threadIdx.x] = src[i*lds];
       }
       __syncthreads();

    //
    //  store transposed block
    //
       for( int i = 0; i < 16; i += BLOCK_SIZE )
       {
              dst[i*ldd].x = a[threadIdx.x][i+threadIdx.y].x;
        dst[i*ldd].y = a[threadIdx.x][i+threadIdx.y].y;
    }
}

static __global__ void transpose_inplace_device( cuDoubleComplex *matrix, int lda, int half, int
parity )
{
    bool bottom = blockIdx.x + parity > blockIdx.y;
       int ibx = bottom ? (blockIdx.x + parity - 1) : (blockIdx.y + half - parity);
       int iby = bottom ? blockIdx.y       : (blockIdx.x + half);

       ibx *= 16;
       iby *= 16;
       int inx = threadIdx.x;
       int iny = threadIdx.y;

       __shared__ cuDoubleComplex a[16][17], b[16][17];

    //
    //  load 32x32 block
    //
       cuDoubleComplex *A = matrix + ibx + inx + ( iby + iny ) * lda;
       for( int i = 0; i < 16; i += BLOCK_SIZE ) {
               a[i+threadIdx.y][threadIdx.x] = A[i*lda];
       }

       if( ibx == iby )
       {
        //
        //  this is a diagonal block
        //
              __syncthreads();
        //
        //  store transposed block
        //
        for( int i = 0; i < 16; i += BLOCK_SIZE )
        {
                A[i*lda] = a[threadIdx.x][i+threadIdx.y];
            }
        }
        else
        {
        //
        //  load the opposite 32x32 block
        //
               cuDoubleComplex *B = matrix + iby + inx + ( ibx + iny ) * lda;
        for( int i = 0; i < 16; i += BLOCK_SIZE ) {
                b[i+threadIdx.y][threadIdx.x] = B[i*lda];
            }

        __syncthreads();

        //
        //  store transposed blocks in reverse order
        //
        for( int i = 0; i < 16; i += BLOCK_SIZE )
        {
                A[i*lda] = b[threadIdx.x][i+threadIdx.y];
        }
        for( int i = 0; i < 16; i += BLOCK_SIZE )
        {
                B[i*lda] = a[threadIdx.x][i+threadIdx.y];
            }
        }
}

extern "C" void gpu_transpose( int m, int n, p2_t dst, p2_t src )
{
    if( m <= 0 || n <= 0 )
        return;

        dim3 threads( 16, BLOCK_SIZE, 1 );
        dim3 grid( (m+15)/16, (n+15)/16, 1 );
        transpose_device<<< grid, threads >>>( dst.A, dst.lda, src.A, src.lda );
    Q( cudaGetLastError( ) );
}

extern "C" void gpu_transpose_inplace( int n, p2_t matrix )
{
    if( n <= 0 )
        return;

        int in = (n+15) / 16;
        dim3 threads( 16, BLOCK_SIZE );
        dim3 grid( in|1, in/2+(in&1) );
        transpose_inplace_device<<< grid, threads >>>( matrix.A, matrix.lda, grid.y, in&1 );
    Q( cudaGetLastError( ) );
                                                    }
```

APPENDIX D

```
/*******************************************************************
*  FDTD.CU
*  Core CUDA FDTD Program
*  Matthew J. Inman and Atef Z. Elsherbeni
*******************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>

/*********************************************************************/
/* Init CUDA                                                         */
/*********************************************************************/
#if __DEVICE_EMULATION__

bool InitCUDA(void){return true;}

#else
bool InitCUDA(void)
{
	int count = 0;
	int i = 0;

	cudaGetDeviceCount(&count);
	if(count == 0) {
		fprintf(stderr, "There is no device.\n");
		return false;
	}

	for(i = 0; i < count; i++) {
		cudaDeviceProp prop;
		if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
			if(prop.major >= 1) {
				break;
			}
		}
	}
	if(i == count) {
		fprintf(stderr, "There is no device supporting CUDA.\n");
		return false;
	}
	cudaSetDevice(i);

	printf("CUDA initialized on %d.\n",i);
	return true;
}

#endif
/*********************************************************************/
/* Example                                                           */
/*********************************************************************/

__global__
void update_E  (float* Ex, float* Ey, float* Ez,
                              float* Hx, float* Hy, float* Hz,
                              float* CExe, float* CEye, float* CEze,
                              float* CExhz, float* CEyhx, float* CEzhy,
                              float* CExhy, float* CEyhz, float* CEzhx,
                              float* CExs, float* CEys, float* CEzs,
                              float V,
                              unsigned int Blocks_Y, float invBlocks_Y,
                              int NX, int NY, int NZ,
                              float* Vout, int iteration, int ii, int jj, int kk)
{
    unsigned int blockIdx_z = __float2uint_rd(blockIdx.y * invBlocks_Y);
    unsigned int blockIdx_y = blockIdx.y - __umul24(blockIdx_z, Blocks_Y);
    unsigned int tx = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    unsigned int ty = __umul24(blockIdx_y, blockDim.y) + threadIdx.y;
    unsigned int tz = __umul24(blockIdx_z, blockDim.z) + threadIdx.z;

        if ((tx >= NX) || (ty >= NY) || (tz >= NZ))
                return;

    // Locations of Indicies
    long int it = tz * NX * NY + ty * NX + tx;
    long int itxp1 = tz * NX * NY + ty * NX + tx+1;
    long int itxm1 = tz * NX * NY + ty * NX + tx-1;
    long int ityp1 = tz * NX * NY + (ty+1) * NX + tx;
    long int itym1 = tz * NX * NY + (ty-1) * NX + tx;
    long int itzp1 = (tz+1) * NX * NY + ty * NX + tx;
    long int itzm1 = (tz-1) * NX * NY + ty * NX + tx;


        if ((tx < NX-1) && (ty > 0) && (ty < NY-1) && (tz > 0) && (tz < NZ-1)) {
                Ex[it] =
                          (CExe[it] * Ex[it]
                          + CExhz[it]*(Hz[it] - Hz[itym1])
                          - CExhy[it]*(Hy[it] - Hy[itzm1])) * (1-CExs[it])
                          + V*CExs[it];
    }

    if ((tx > 0) && (tx < NX-1) && (ty < NY-1) && (tz > 0) && (tz < NZ-1)) {
                Ey[it] =
                          (CEye[it] * Ey[it]
                          + CEyhx[it]*(Hx[it] - Hx[itzm1])
                          - CEyhz[it]*(Hz[it] - Hz[itxm1])) * (1-CEys[it])
                          + V*CEys[it];
        }

        if ((tx > 0) && (tx < NX-1) && (ty > 0) && (ty < NY-1) && (tz < NZ-1)) {
                Ez[it] =
                          (CEze[it] * Ez[it]
                          + CEzhy[it]*(Hy[it] - Hy[itxm1])
                          - CEzhx[it]*(Hx[it] - Hx[itym1])) * (1-CEzs[it])
                          + V*CEzs[it];
    }

    if ((tx == ii) && (ty == jj) && (tz == kk))
    {
                Vout[iteration] = Ez[it];
    }

}

__global__
void update_H  (float* Ex, float* Ey, float* Ez,
                               float* Hx, float* Hy, float* Hz,
                               float* CHxh, float* CHyh, float* CHzh,
                               float* CHxey, float* CHyez, float* CHzex,
                               float* CHxez, float* CHyex, float* CHzey,
                               unsigned int Blocks_Y, float invBlocks_Y,
                               int NX, int NY, int NZ)
{
    unsigned int blockIdx_z = __float2uint_rd(blockIdx.y * invBlocks_Y);
    unsigned int blockIdx_y = blockIdx.y - __umul24(blockIdx_z, Blocks_Y);
    unsigned int tx = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    unsigned int ty = __umul24(blockIdx_y, blockDim.y) + threadIdx.y;
    unsigned int tz = __umul24(blockIdx_z, blockDim.z) + threadIdx.z;

        if ((tx >= NX) || (ty >= NY) || (tz >= NZ))
                return;

    // Locations of Indicies
    long int it = tz * NX * NY + ty * NX + tx;
    long int itxp1 = tz * NX * NY + ty * NX + tx+1;
    long int itxm1 = tz * NX * NY + ty * NX + tx-1;
    long int ityp1 = tz * NX * NY + (ty+1) * NX + tx;
    long int itym1 = tz * NX * NY + (ty-1) * NX + tx;
    long int itzp1 = (tz+1) * NX * NY + ty * NX + tx;
    long int itzm1 = (tz-1) * NX * NY + ty * NX + tx;


//      Ex[it]=V*Evx[it];

        if ((tx < NX-1) && (tx > 0) && (ty < NY-1) && (tz < NZ-1)) {
                Hx[it]  = CHxh[it] * Hx[it]
                              + CHxey[it] * (Ey[itzp1] - Ey[it])
                              - CHxez[it] * (Ez[ityp1] - Ez[it]);
    }

    if ((ty > 0) && (tx < NX-1) && (ty < NY-1) && (tz < NZ-1)) {
                Hy[it]  = CHyh[it] * Hy[it]
                              + CHyez[it] * (Ez[itxp1] - Ez[it])
                              - CHyex[it] * (Ex[itzp1] - Ex[it]);
        }

        if ((tx < NX-1) && (tz > 0) && (ty < NY-1) && (tz < NZ-1)) {
                Hz[it]  = CHzh[it] * Hz[it]
                              + CHzex[it] * (Ex[ityp1] - Ex[it])
                              - CHzey[it] * (Ey[itxp1] - Ey[it]);
    }


}

/**********************************************************************/
/* Main Program                                                       */
/**********************************************************************/
int main(int argc, char* argv[])
{


/**********************************************************************/
/* Define Vars                                                        */
/**********************************************************************/

        float x1,x2,y1,y2,z1,z2,dx,dy,dz,mu0,eps0, c, dtfactor,  nc,tau,t0,dt,Ce,Ch, pi, dsmax,
time;
        int nx, ny, nz, ncells, nsteps;
        int i,j,k;
        long int it;
        unsigned int flags;
        cudaDeviceProp deviceProp;



        float *V = NULL;
        float *Vout = NULL; float *d_Vout = NULL;
        // Fields
        float *Ex = NULL;      float *Hx = NULL; float *d_Ex = NULL;        float *d_Hx = NULL;
        float *Ey = NULL;      float *Hy = NULL; float *d_Ey = NULL;        float *d_Hy = NULL;
        float *Ez = NULL;      float *Hz = NULL; float *d_Ez = NULL;        float *d_Hz = NULL;

        float *CExe = NULL;   float *d_CExe = NULL;
        float *CExhz = NULL;  float *d_CExhz = NULL;
        float *CExhy = NULL;  float *d_CExhy = NULL;
        float *CExs = NULL;   float *d_CExs = NULL;

        float *CEye = NULL;   float *d_CEye = NULL;
        float *CEyhz = NULL;  float *d_CEyhz = NULL;
        float *CEyhx = NULL;  float *d_CEyhx = NULL;
        float *CEys = NULL;   float *d_CEys = NULL;

        float *CEze = NULL;   float *d_CEze = NULL;
        float *CEzhx = NULL;  float *d_CEzhx = NULL;
        float *CEzhy = NULL;  float *d_CEzhy = NULL;
        float *CEzs = NULL;   float *d_CEzs = NULL;

        float *CHxh = NULL;   float *d_CHxh = NULL;
        float *CHxey = NULL;  float *d_CHxey = NULL;
        float *CHxez = NULL;  float *d_CHxez = NULL;
        float *CHxm = NULL;   float *d_CHxm = NULL;

        float *CHyh = NULL;   float *d_CHyh = NULL;
        float *CHyex = NULL;  float *d_CHyex = NULL;
        float *CHyez = NULL;  float *d_CHyez = NULL;
        float *CHym = NULL;   float *d_CHym = NULL;

        float *CHzh = NULL;   float *d_CHzh = NULL;
        float *CHzex = NULL;  float *d_CHzex = NULL;
        float *CHzey = NULL;  float *d_CHzey = NULL;
        float *CHzm = NULL;   float *d_CHzm = NULL;

        cudaError_t ret;

/*********************************************************************/
/* Initialize CUDA                                                   */
/*********************************************************************/

        if(!InitCUDA()) {
                return 0;
        }

        cutilSafeCall(cudaGetDeviceProperties(&deviceProp, 0));

        #if CUDART_VERSION >= 2020
          if(!deviceProp.canMapHostMemory)
          {
                fprintf(stderr, "Device %d cannot map host memory!\n", 0);
                printf("PASSED");
                cutilExit(argc, argv);
          }
          cutilSafeCall(cudaSetDeviceFlags(cudaDeviceMapHost));
        #else
          fprintf(stderr, "This CUDART version does not support <cudaDeviceProp.canMapHostMemory>
field\n");
          printf("PASSED");
          cutilExit(argc, argv);
        #endif

/*********************************************************************/
/* Initialize Constants                                              */
/*********************************************************************/

        x1 = -10e-3; x2 = 10e-3; y1 = -10e-3; y2 = 10e-3; z1 = -10e-3; z2 = 10e-3;
        dx = 0.4064e-3;   dy = 0.4233e-3;   dz = 0.265e-3;
        pi = 3.14159265;

        mu0 = 4*pi*1e-7; eps0 = 8.8419e-012; c = 1/sqrt(mu0*eps0);



//      nx = round((x2-x1)/dx)+1; ny = round((y2-y1)/dy)+1; nz = round((z2-z1)/dz)+1;

for (nx=100;nx<201; nx=nx+100) {
for (ny=100;ny<201; ny=ny+100) {
for (nz=100;nz<201; nz=nz+100) {
        ncells = nx*ny*nz;

        if (argc > 1) {
               nsteps = atoi(argv[1]);
        } else {
               nsteps = 100;
        }

        //printf("Number of Cells: cells=%d x=%d y=%d z=%d n=%d\n", ncells, nx,ny ,nz, nsteps);

        nc = 25;
        dtfactor = 0.95;


        //dsmax = max([dx,dy,dz]);
        if ( (dx >= dy) && ( dx >= dz ) ) {
               dsmax = dx;
        }
        if ( (dz >= dy) && ( dz >= dx ) ) {
               dsmax = dz;
        }
        if ( (dy >= dx) && ( dy >= dz ) ) {
               dsmax = dy;
        }

        tau = nc*dsmax/(2*c);
        t0 = 3 * tau;
        dt = 1/(c*sqrt((1/(dx*dx))+(1/(dy*dy))+(1/(dz*dz))));
        dt = dtfactor*dt;
        Ce = dt/(2*eps0); Ch = dt/(2*mu0);

/************************************************************************/
/* Allocate Host and Device Arrays                                      */
/************************************************************************/
        flags = cudaHostAllocMapped;
        ret = cudaMallocHost( (void**)&V,   nsteps*sizeof(float) );
        ret = cudaMallocHost( (void**)&Vout,   nsteps*sizeof(float) );


        ret = cudaHostAlloc( (void**)&Ex, nx*ny*nz*sizeof(float), flags  );
        ret = cudaHostAlloc( (void**)&Hx, nx*ny*nz*sizeof(float), flags  );


        ret = cudaMallocHost( (void**)&Ey, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&Ez, nx*ny*nz*sizeof(float) );

        ret = cudaMallocHost( (void**)&Hy, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&Hz, nx*ny*nz*sizeof(float) );

        ret = cudaMallocHost( (void**)&CExe, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CExhz, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CExhy, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CExs, nx*ny*nz*sizeof(float) );

        ret = cudaMallocHost( (void**)&CEye, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CEyhx, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CEyhz, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CEys, nx*ny*nz*sizeof(float) );

        ret = cudaMallocHost( (void**)&CEze, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CEzhx, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CEzhy, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CEzs, nx*ny*nz*sizeof(float) );

        ret = cudaMallocHost( (void**)&CHxh, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CHxey, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CHxez, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CHxm, nx*ny*nz*sizeof(float) );

        ret = cudaMallocHost( (void**)&CHyh, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CHyex, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CHyez, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CHym, nx*ny*nz*sizeof(float) );

        ret = cudaMallocHost( (void**)&CHzh, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CHzex, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CHzey, nx*ny*nz*sizeof(float) );
        ret = cudaMallocHost( (void**)&CHzm, nx*ny*nz*sizeof(float) );




        // Device Arrays
        ret = cudaMalloc( (void**)&d_Vout,   nsteps*sizeof(float) );


        //ret = cudaMalloc( (void**)&d_Ex, nx*ny*nz*sizeof(float) );
        //ret = cudaMalloc( (void**)&d_Hx, nx*ny*nz*sizeof(float) );
        cudaHostGetDevicePointer((void **)&d_Ex, (void *)Ex, 0);
        cudaHostGetDevicePointer((void **)&d_Hx, (void *)Hx, 0);


        ret = cudaMalloc( (void**)&d_Ey, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_Ez, nx*ny*nz*sizeof(float) );

        ret = cudaMalloc( (void**)&d_Hy, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_Hz, nx*ny*nz*sizeof(float) );

        ret = cudaMalloc( (void**)&d_CExe, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CExhz, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CExhy, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CExs, nx*ny*nz*sizeof(float) );

        ret = cudaMalloc( (void**)&d_CEye, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CEyhx, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CEyhz, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CEys, nx*ny*nz*sizeof(float) );

        ret = cudaMalloc( (void**)&d_CEze, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CEzhx, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CEzhy, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CEzs, nx*ny*nz*sizeof(float) );

        ret = cudaMalloc( (void**)&d_CHxh, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CHxey, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CHxez, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CHxm, nx*ny*nz*sizeof(float) );

        ret = cudaMalloc( (void**)&d_CHyh, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CHyex, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CHyez, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CHym, nx*ny*nz*sizeof(float) );

        ret = cudaMalloc( (void**)&d_CHzh, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CHzex, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CHzey, nx*ny*nz*sizeof(float) );
        ret = cudaMalloc( (void**)&d_CHzm, nx*ny*nz*sizeof(float) );

/********************************************************************/
/* Initialize Arrays                                                */
/********************************************************************/

        time=-dt;
        for (i=0;i<nsteps;i++) {
                time=time+dt;
                V[i] = exp(-(((time - t0)*(time - t0)))/(tau*tau));
        }

        for (k=0;k<nz;k++) {
                for (j=0;j<ny;j++) {
                        for (i=0;i<nx;i++) {
                                it = i+(nx*j)+(k*nx*ny);
                                Ex[it] = 0;
                                Ey[it] = 0;
                                Ez[it] = 0;

                                Hx[it] = 0;
                                Hy[it] = 0;
                                Hz[it] = 0;

                                CExe[it]= 1;
                                CExhz[it]=(2*Ce/dy);
                                CExhy[it]=(2*Ce/dz);
                                CExs[it]=0;


                                CEye[it]= 1;
                                CEyhx[it]=(2*Ce/dz);
                                CEyhz[it]=(2*Ce/dx);
                                CEys[it]=0;

                                CEze[it]= 1;
                                CEzhy[it]=(2*Ce/dx);
                                CEzhx[it]=(2*Ce/dy);
                                CEzs[it]=0;


                                CHxh[it]= 1;
                                CHxey[it]=(2*Ch/dz);
                                CHxez[it]=(2*Ch/dy);
                                CHxm[it]=(2*Ch);


                                CHyh[it]= 1;
                                CHyez[it]=(2*Ch/dx);
                                CHyex[it]=(2*Ch/dz);
                                CHym[it]=(2*Ch);


                                CHzh[it]= 1;
                                CHzex[it]=(2*Ch/dy);
                                CHzey[it]=(2*Ch/dx);
                                CHzm[it]=(2*Ch);
                        }
                }
        }
        //Set Voltage Source Point
        CEzs[(nx/2)+(nx*(ny/2))+(((nz/2)+1)*nx*ny)] = 1;
        CEzs[(nx/2)+(nx*(ny/2))+(((nz/2))*nx*ny)] = 1;

/*********************************************************************/
/* Copy Arrays To GPU                                                */
/*********************************************************************/
        cudaMemcpy(d_Ex, Ex , nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_Ey, Ey, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_Ez, Ez, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);

        cudaMemcpy(d_Hx, Hx, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_Hy, Hy, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_Hz, Hy, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CExe, CExe, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CExhz, CExhz, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CExhy, CExhy, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CExs, CExs, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);

        cudaMemcpy(d_CEye, CEye, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CEyhz, CEyhz, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CEyhx, CEyhx, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CEys, CEys, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);

        cudaMemcpy(d_CEze, CEze, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CEzhx, CEzhx, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CEzhy, CEzhy, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CEzs, CEzs, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);


        cudaMemcpy(d_CHxh, CHxh, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CHxey, CHxey, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CHxez, CHxez, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CHxm, CHxm, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);

        cudaMemcpy(d_CHyh, CHyh, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CHyex, CHyex, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CHyez, CHyez, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CHym, CHym, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);

        cudaMemcpy(d_CHzh, CHzh, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CHzex, CHzex, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CHzey, CHzey, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_CHzm, CHzm, nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);


/*************************************************************************/
/* Setup for Calling Kernel                                              */
/*************************************************************************/

   // Thead Block Dimensions
    int tBlock_x = 5;
    int tBlock_y = 4;
    int tBlock_z = 4;

    // Used to build "3D Grid"
    int blocksInX;  dim3 dimGrid;
    int blocksInY;  dim3 dimBlock;
    int blocksInZ;

    FILE *debug_file;
    // Each element in the volume (each voxel) gets 1 thread
    blocksInX = (nx+tBlock_x-1)/tBlock_x;
    blocksInY = (ny+tBlock_y-1)/tBlock_y;
    blocksInZ = (nz+tBlock_z-1)/tBlock_z;

    dimGrid  = dim3(blocksInX, blocksInY*blocksInZ);
    dimBlock = dim3(tBlock_x, tBlock_y, tBlock_z);

    unsigned int timer = 0;
    unsigned int timer2 = 0;
    cutCreateTimer( &timer);
    cutCreateTimer( &timer2);
    cutStartTimer( timer);


    for (i=0;i<nsteps;i++) {

       update_E<<<dimGrid, dimBlock>>>(d_Ex, d_Ey, d_Ez,
                                                                         d_Hx, d_Hy, d_Hz,
                                                                            d_CExe, d_CEye,
d_CEze,
                                                                            d_CExhz, d_CEyhx,
d_CEzhy,
                                                                            d_CExhy, d_CEyhz,
d_CEzhx,
                                                                            d_CExs, d_CEys,
d_CEzs,
                                                                            V[i], blocksInY,
1.0f/(float)blocksInY, nx, ny, nz,
                                                                            d_Vout, i, 49, 49, 49
);
          CUT_CHECK_ERROR("Kernel E execution failed");

       cudaThreadSynchronize();

        update_H<<<dimGrid, dimBlock>>>(d_Ex, d_Ey, d_Ez,
                                                                            d_Hx, d_Hy, d_Hz,
                                                                            d_CHxh, d_CHyh,
d_CHzh,
                                                                            d_CHxey, d_CHyez,
d_CHzex,
                                                                            d_CHxez, d_CHyex,
d_CHzey,
                                                                            blocksInY,
1.0f/(float)blocksInY, nx, ny, nz);
               CUT_CHECK_ERROR("Kernel H execution failed");
               cudaThreadSynchronize();
     }

    CUT_SAFE_CALL( cutStopTimer( timer));

    cutStartTimer( timer2);
       for (i=0;i<nsteps;i++) {
               //cudaMemcpy(Ex, d_Ex , nx*ny*nz*sizeof(float), cudaMemcpyDeviceToHost);
               for (k=0;k<nz;k++) {
                       for (j=0;j<ny;j++) {
                               it = (nx-1)+(nx*j)+(k*nx*ny);
                               Ex[it]=Ex[it]+1;
                       }
                }
               //cudaMemcpy(d_Ex, Ex , nx*ny*nz*sizeof(float), cudaMemcpyHostToDevice);
       }
       CUT_SAFE_CALL( cutStopTimer( timer2));


    printf("%d %d %d %d %d %f %f\n", ncells, nx,ny ,nz, nsteps, cutGetTimerValue(
timer)/(float)nsteps,cutGetTimerValue( timer2)/(float)nsteps);


    CUT_SAFE_CALL( cutDeleteTimer( timer));
    CUT_SAFE_CALL( cutDeleteTimer( timer2));
/*********************************************************************/
/* Copy Data Back                                                    */
/*********************************************************************/
    cudaMemcpy( Vout, d_Vout, nsteps * sizeof(float), cudaMemcpyDeviceToHost );
// Debug

/*********************************************************************/
/* Export Data                                                       */
/*********************************************************************/
      printf("Writing debug file...\n");
    debug_file = fopen("debug_out2.txt", "wt");
/*

    for(i=0; i < nx* ny * nz; i++)
    {
         fprintf(debug_file, "[%i]\t%f\t%f\t%f\n",i , Ex1[i], Ey1[i], Ez1[i]);
    }


    k=4;
              for (j=0;j<ny;j++) {
                     for (i=0;i<nx;i++) {
                            fprintf(debug_file, "%e ", Ez1[i+(nx*j)+(k*nx*ny)]);
                     }
                     fprintf(debug_file,"\n");
              }


       */

       for (i=0; i<nsteps; i++) {
              fprintf(debug_file, "%e %e \n", V[i], Vout[i]);
    }

    fclose(debug_file);
/*********************************************************************/
/* Clean up                                                          */
/*********************************************************************/

       cudaFreeHost(V);
       cudaFreeHost(Vout);


       cudaFreeHost(Ex);
       cudaFreeHost(Ey);
       cudaFreeHost(Ez);

       cudaFreeHost(Hx);
       cudaFreeHost(Hy);
       cudaFreeHost(Hz);

       cudaFreeHost(CExe);
       cudaFreeHost(CExhz);
       cudaFreeHost(CExhy);
       cudaFreeHost(CExs);

       cudaFreeHost(CEye);
       cudaFreeHost(CEyhx);
       cudaFreeHost(CEyhz);
       cudaFreeHost(CEys);

       cudaFreeHost(CEze);
       cudaFreeHost(CEzhx);
       cudaFreeHost(CEzhy);
       cudaFreeHost(CEzs);

       cudaFreeHost(CHxh);
       cudaFreeHost(CHxey);
       cudaFreeHost(CHxez);
       cudaFreeHost(CHxm);

       cudaFreeHost(CHyh);
       cudaFreeHost(CHyex);
       cudaFreeHost(CHyez);
       cudaFreeHost(CHym);

       cudaFreeHost(CHzh);
       cudaFreeHost(CHzex);
       cudaFreeHost(CHzey);
       cudaFreeHost(CHzm);
       // Device Arrays
       cudaFree(d_Vout);


       cudaFree(d_Ex);
       cudaFree(d_Ey);
       cudaFree(d_Ez);

       cudaFree(d_Hx);
       cudaFree(d_Hy);
       cudaFree(d_Hz);

       cudaFree(d_CExe);
       cudaFree(d_CExhz);
       cudaFree(d_CExhy);
       cudaFree(d_CExs);

       cudaFree(d_CEye);
       cudaFree(d_CEyhx);
       cudaFree(d_CEyhz);
       cudaFree(d_CEys);

       cudaFree(d_CEze);
       cudaFree(d_CEzhx);
       cudaFree(d_CEzhy);
       cudaFree(d_CEzs);

       cudaFree(d_CHxh);
       cudaFree(d_CHxey);
       cudaFree(d_CHxez);
       cudaFree(d_CHxm);

       cudaFree(d_CHyh);
       cudaFree(d_CHyex);
       cudaFree(d_CHyez);
       cudaFree(d_CHym);

       cudaFree(d_CHzh);
       cudaFree(d_CHzex);
       cudaFree(d_CHzey);
       cudaFree(d_CHzm);
       }
       }
       }

       CUT_EXIT(argc, argv);


   return 0;
                                                }
```

APPENDIX E

```
// Full 3D FDTD w CPML Program
// Executes FDTD in GPU
// Matthew J. Inman


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


kernel void process_field_H( float3 H[][], float3 E[][], float3 Kh[], float3 b[], float3 c[],
float XE, float XH,
                      float dx, float dy, float dz, float ysize, iter float2 it<>,
                      float3 psiH1[][], float3 psiH2[][],
                      out float3 o_psiH1<>, out float3 o_psiH2<>, out float3 o_H<> ) {

    float2 t0 = float2(0.0f, ysize);
    float2 t1 = float2(0.0f, 1.0f);
    float2 t2 = float2(1.0f, 0.0f);


    float pxy, pxz, pyx, pyz, pzx, pzy;

       pxy = (b[it.y].y * psiH1[it].x) + (c[it.y].y * (E[it+t1].z-E[it].z));
       pxz = (b[it.y].z * psiH2[it].x) + (c[it.y].z * (E[it+t0].y-E[it].y));

       pyx = (b[it.x].x * psiH1[it].y) + (c[it.x].x * (E[it+t2].z-E[it].z));
       pyz = (b[it.y].z * psiH2[it].y) + (c[it.y].z * (E[it+t0].x-E[it].x));

       pzx = (b[it.x].x * psiH1[it].z) + (c[it.x].x * (E[it+t2].y-E[it].y));
       pzy = (b[it.y].y * psiH2[it].z) + (c[it.y].y * (E[it+t1].x-E[it].x));

       o_psiH1.x=pxy;
       o_psiH2.x=pxz;

       o_psiH1.y=pyx;
       o_psiH2.y=pyz;

       o_psiH1.z=pzx;
       o_psiH2.z=pzy;

       o_H.x  = (XH * H[it].x) + (Kh[it.y].z*(XE/dz) * (E[it+t0].y-E[it].y)) -
(Kh[it.y].y*(XE/dy) * (E[it+t1].z-E[it].z))  - ((XE) * pxy) + ((XE) * pxz);
       o_H.y  = (XH * H[it].y) + (Kh[it.x].x*(XE/dx) * (E[it+t2].z-E[it].z)) -
(Kh[it.y].z*(XE/dz) * (E[it+t0].x-E[it].x))  + ((XE) * pyx) - ((XE) * pyz);
       o_H.z  = (XH * H[it].z) + (Kh[it.y].y*(XE/dy) * (E[it+t1].x-E[it].x)) -
(Kh[it.x].x*(XE/dx) * (E[it+t2].y-E[it].y))  - ((XE) * pzx) + ((XE) * pzy);
}


kernel void process_field_E( float3 E[][], float3 H[][], float3 Cee[][], float3 Ceh[][], float3
Ces[][], float3 Ke[],
                              float3 b[], float3 c[], float dx, float dy, float dz, float ysize,
float gauss, iter float2 it<>,
                              float3 psiE1[][], float3 psiE2[][],
                              out float3 o_psiE1<>, out float3 o_psiE2<>, out float3 o_E<> ) {


    float2 t0 = float2(0.0f, -1.0f);
    float2 t1 = float2(0.0f,-1*ysize);
    float2 t3 = float2(-1.0f,0.0f);


    float pxy, pxz, pyx, pyz, pzx, pzy;

       pxy = (b[it.y].y * psiE1[it].x) + (c[it.y].y * (H[it].z-H[it+t0].z));
```

```
        pxz = (b[it.y].z * psiE2[it].x) + (c[it.y].z * (H[it].y-H[it+t1].y));

        pyx = (b[it.x].x * psiE1[it].y) + (c[it.x].x * (H[it].z-H[it+t3].z));
        pyz = (b[it.y].z * psiE2[it].y) + (c[it.y].z * (H[it].x-H[it+t1].x));

        pzx = (b[it.x].x * psiE1[it].z) + (c[it.x].x * (H[it].y-H[it+t3].y));
        pzy = (b[it.y].y * psiE2[it].z) + (c[it.y].y * (H[it].x-H[it+t0].x));

        o_psiE1.x=pxy;
        o_psiE2.x=pxz;

        o_psiE1.y=pyx;
        o_psiE2.y=pyz;

        o_psiE1.z=pzx;
        o_psiE2.z=pzy;

        o_E.x  = (Cee[it].x * E[it].x) + (Ke[it.y].y*(Ceh[it].x/dy) * (H[it].z-H[it+t0].z)) -
(Ke[it.y].z*(Ceh[it].x/dz) * (H[it].y-H[it+t1].y)) -
                (Ces[it].x*gauss) + (Ceh[it].x * pxy) - (Ceh[it].x * pxz);

        o_E.y  = (Cee[it].y * E[it].y) + (Ke[it.y].z*(Ceh[it].y/dz) * (H[it].x-H[it+t1].x)) -
(Ke[it.x].x*(Ceh[it].y/dx) * (H[it].z-H[it+t3].z)) -
                (Ces[it].y*gauss) - (Ceh[it].y * pyx) + (Ceh[it].y * pyz);

        o_E.z  = (Cee[it].z * E[it].z) + (Ke[it.x].x*(Ceh[it].z/dx) * (H[it].y-H[it+t3].y)) -
(Ke[it.y].y*(Ceh[it].z/dy) * (H[it].x-H[it+t0].x)) -
                (Ces[it].z*gauss) + (Ceh[it].z * pzx) - (Ceh[it].z * pzy);

}


kernel void process_test( float3 H[][], iter float2 it<>, out float3 o_H<>) {

//      o_H.x=(H[it].x+(2*H[it].x)-(4.5*H[it].y)+(6.7*H[it].x)+(3.1*H[it].y)+(5.1*H[it].y)-
(10.2/H[it].x)+(123*H[it].z)+(H[it].x*22.123)+(H[it].x*H[it].z)+(3.1*H[it].y)+(5.1*H[it].y)-
(10.2/H[it].x)+(123*H[it].z)+(H[it].x*22.123));
//      o_H.y=(H[it].y+(2*H[it].x)-(4.2*H[it].y)+(6.2*H[it].x)+(3.12*H[it].y)+(5.31*H[it].y)-
(10.52/H[it].x)+(152*H[it].z)+(H[it].x*22.1323)+(H[it].x*H[it].z)+(3.1*H[it].y)+(5.1*H[it].y)-
(10.2/H[it].x)+(123*H[it].z)+(H[it].x*22.123));
//      o_H.z=(H[it].z+(2*H[it].x)-(4.8*H[it].y)+(6.1*H[it].x)+(3.16*H[it].y)+(5.61*H[it].y)-
(10.32/H[it].x)+(121*H[it].z)+(H[it].x*22.1233)+(H[it].x*H[it].z)+(3.1*H[it].y)+(5.1*H[it].y)-
(10.2/H[it].x)+(123*H[it].z)+(H[it].x*22.123));

        o_H.x=H[it].x+it.x;
        o_H.y=H[it].y+it.y;
        o_H.z=H[it].z+it.y;


//      o_H.x=H[it].x;
//      o_H.y=H[it].y;
//      o_H.z=H[it].z;
}

// Obs Points
kernel void Copy( float3 input<>, out float3 output<> ) {
        output.x = input.x;
        output.y = input.y;
        output.z = input.z;
        }




int main(int argc, char* argv[]) {

    int i, j, k, N, l;
    int xsize, ysize, zsize,outsize, outsize2, iPML;
    int px1, px2, py1, py2, pz1, pz2;

    int mXxHpml, mYxHpml, mZxHpml;
    int mXxLpml, mYxLpml, mZxLpml;

    int mXyHpml, mYyHpml, mZyHpml;
    int mXyLpml, mYyLpml, mZyLpml;

    int mXzHpml, mYzHpml, mZzHpml;
    int mXzLpml, mYzLpml, mZzLpml;

    char op[2];
    float op2, op3, op4, op5, op6, op7, op8, op9, op10;

    int s0;


    float eps0, mu0, c, pi, dx, dy, dz, dfactor, dt, M, tau, t0,fmax, ii;
    float dx2, dy2, dz2, kappa, k1, k2 ,o1,o2,i1, i2;
    float ce, XE, XH,m,sigmax, sigmay, sigmaz, amax,sig1,sig2,a1,a2, vObs, R;
    float* aObs=NULL;

    float* t=NULL;
    float* gauss=NULL;

    float* aH=NULL;
    float* aE=NULL;

    float* aCee=NULL;
    float* aCeh=NULL;
    float* aCes=NULL;

    float* aKe=NULL;
    float* aKh=NULL;

    float* aBe=NULL;
    float* aBh=NULL;

    float* aCe=NULL;
    float* aCh=NULL;

    float* apsiHxy=NULL;
    float* apsiHxz=NULL;
    float* apsiHyx=NULL;
    float* apsiHyz=NULL;
    float* apsiHzx=NULL;
    float* apsiHzy=NULL;

    float* apsiExy=NULL;
    float* apsiExz=NULL;
    float* apsiEyx=NULL;
    float* apsiEyz=NULL;
    float* apsiEzx=NULL;
    float* apsiEzy=NULL;

    FILE * pFile;
    FILE * pFile2;
    FILE * pFile3;
    FILE * pFile4;
    FILE * pFile5;
    FILE * pFile6;


    pFile = fopen ("myfile1.txt","wt");
    pFile6 = fopen ("source.txt","wt");
```

```
    // Constants
    pi=3.14159265;

    // Broken Down because brook hates small numbers
    eps0= 8.854;
    eps0=eps0*1e-6;
    eps0=eps0*1e-6;

    mu0= 4*pi;
    mu0= mu0*1e-4;
    mu0= mu0*1e-3;

    c=2.99792479e8;

    fprintf (pFile,"pi=%f eps0=%e mu0=%e c=%f \n",pi,eps0,mu0,c);
   // Patch Definition

    s0=10;
    vObs=0;


    // Read Input file 1
   pFile2 = fopen ("domain_parameters.txt","r");

   while (fscanf(pFile2, "%s %f", op, &op2) != EOF) {
        if (strcmp(op,"ts")==0) {
          N = (int) op2;
        }

        if (strcmp(op,"dx")==0) {
          dx = op2;
        }

        if (strcmp(op,"dy")==0) {
          dy = op2;
        }

        if (strcmp(op,"dz")==0) {
          dz = op2;
        }

        if (strcmp(op,"nx")==0) {
          xsize = (int) op2+20;
        }

        if (strcmp(op,"ny")==0) {
          ysize = (int) op2+20;
        }

        if (strcmp(op,"nz")==0) {
          zsize = (int) op2+20;
        }

   }

  fclose(pFile2);
/*-------------------------
Old Stuff
    N = atoi(argv[1]);
    dx=.4233e-3;
    dy=.4064e-3;
    dz=.265e-3;

    xsize = s0+s0+x1+x2+x3+x4+x5+1; // 95
    ysize = s0+s0+y1+y2+y3+y4+y5+y6+y7+y8+y9+1; // 89
    zsize = s0+s0+z1+z2+z3+1; // 32
---------------------------*/
```

```
   // Check for unsafe sizes

    if ((ysize*zsize)%2==1) { ysize=ysize+1; }

    outsize=ysize*zsize;


    dx2=(c/dx)*(c/dx);
    dy2=(c/dy)*(c/dy);
    dz2=(c/dz)*(c/dz);

    fprintf (pFile,"dx=%e dy=%e dx2=%e dy2=%e \n",dx,dy,dx2,dy2);




    dfactor=.9;
    dt=(1/(sqrt(dx2 + dy2 + dz2)))*dfactor;

    M=25;


    fmax=c/(M*dy);
    tau=(M*dy)/(2*c);
        t0=3*tau;
    fprintf (pFile,"dt=%e tau=%e t0=%e fmax=%e \n",dt,tau,t0,fmax);




    // PML Params
        iPML= 10;

        mXxHpml=iPML;
        mYxHpml=ysize;
        mZxHpml=zsize;
        mXxLpml=iPML;
        mYxLpml=ysize;
        mZxLpml=zsize;

        mXyHpml=xsize;
        mYyHpml=iPML;
        mZyHpml=zsize;
        mXyLpml=xsize;
        mYyLpml=iPML;
        mZyLpml=zsize;

        mXzHpml=xsize;
        mYzHpml=ysize;
        mZzHpml=iPML;
        mXzLpml=xsize;
        mYzLpml=ysize;
        mZzLpml=iPML;

        kappa=8;

        m=4;

        sigmax = (.8*m+1) / (150*pi*dx);
        sigmay = (.8*m+1) / (150*pi*dy);
        sigmaz = (.8*m+1) / (150*pi*dz);

        amax = (fmax/2.1)*2*pi*eps0/10;


    printf("GPU FDTD Code\n x=%d y=%d z=%d ns=%d d1=%d d2=%d\n sigmax=%e amax=%e \n", xsize,
ysize, zsize, N, xsize, ysize*zsize, sigmax, amax);
```

```
    // Initialize de arrays muhahaha
    gauss=    (float*)malloc(N*sizeof(float));
    aObs=     (float*)malloc(10*sizeof(float));
    t=        (float*)malloc(N*sizeof(float));
    aH=       (float*)malloc(3*xsize*ysize*zsize*sizeof(float));
    aE=       (float*)malloc(3*xsize*ysize*zsize*sizeof(float));
    aCee=     (float*)malloc(3*xsize*ysize*zsize*sizeof(float));
    aCeh=     (float*)malloc(3*xsize*ysize*zsize*sizeof(float));
    aCes=     (float*)malloc(3*xsize*ysize*zsize*sizeof(float));


        // 1D CPML Arrays
        aBe=    (float*)malloc(3*ysize*zsize*sizeof(float));
        aCe=    (float*)malloc(3*ysize*zsize*sizeof(float));

        aBh=    (float*)malloc(3*ysize*zsize*sizeof(float));
        aCh=    (float*)malloc(3*ysize*zsize*sizeof(float));

        // 3D CPML Arrays
        apsiHxy=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiHxz=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiHyx=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiHyz=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiHzx=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiHzy=    (float*)malloc(xsize*ysize*zsize*sizeof(float));

        apsiExy=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiExz=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiEyx=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiEyz=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiEzx=    (float*)malloc(xsize*ysize*zsize*sizeof(float));
        apsiEzy=    (float*)malloc(xsize*ysize*zsize*sizeof(float));


        aKe=    (float*)malloc(3*ysize*zsize*sizeof(float));
        aKh=    (float*)malloc(3*ysize*zsize*sizeof(float));

        // Source Terms

    t[0]=dt;
    gauss[0]=0;
    for (i=1; i<N; i++) {
              t[i]=t[i-1]+dt;
              gauss[i]= exp(-( ((t[i]-t0)*(t[i]-t0))/(tau*tau)))/3;
              fprintf (pFile6,"%e\n",gauss[i]);
        }
        fclose(pFile6);
        // Coefficients
    ce=dt/(2*eps0);
    XE=(dt/mu0);
    XH=1;



//Initialize Variables
for (l=0;l<3;l++) {
        for (k=0; k<zsize; k++) {
                for (j=0; j<ysize; j++) {
                        for (i=0; i<xsize; i++) {

                        aH[((xsize*ysize*k)+(xsize*j)+i)*3+l]=0;
                        aE[((xsize*ysize*k)+(xsize*j)+i)*3+l]=0;

                        aCee[((xsize*ysize*k)+(xsize*j)+i)*3+l]=1;
```

```
                       aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+l]=2*ce;
                       aCes[((xsize*ysize*k)+(xsize*j)+i)*3+l]=0;
/// 3D Pml

                       apsiHxy[((xsize*ysize*k)+(xsize*j)+i)]=0;
                       apsiHxz[((xsize*ysize*k)+(xsize*j)+i)]=0;

                       apsiHyx[((xsize*ysize*k)+(xsize*j)+i)]=0;
                       apsiHyz[((xsize*ysize*k)+(xsize*j)+i)]=0;

                       apsiHzx[((xsize*ysize*k)+(xsize*j)+i)]=0;
                       apsiHzy[((xsize*ysize*k)+(xsize*j)+i)]=0;

                       apsiExy[((xsize*ysize*k)+(xsize*j)+i)]=0;
                       apsiExz[((xsize*ysize*k)+(xsize*j)+i)]=0;

                       apsiEyx[((xsize*ysize*k)+(xsize*j)+i)]=0;
                       apsiEyz[((xsize*ysize*k)+(xsize*j)+i)]=0;

                       apsiEzx[((xsize*ysize*k)+(xsize*j)+i)]=0;
                       apsiEzy[((xsize*ysize*k)+(xsize*j)+i)]=0;

// 1D Pml
                       aBe[3*i]=0;
                       aCe[3*i]=0;
                       aBe[((ysize*k)+(j))*3+1]=0;
                       aCe[((ysize*k)+(j))*3+1]=0;
                       aBe[((ysize*k)+(j))*3+2]=0;
                       aCe[((ysize*k)+(j))*3+2]=0;

                       aBh[3*i]=0;
                       aCh[3*i]=0;
                       aBh[((ysize*k)+(j))*3+1]=0;
                       aCh[((ysize*k)+(j))*3+1]=0;
                       aBh[((ysize*k)+(j))*3+2]=0;
                       aCh[((ysize*k)+(j))*3+2]=0;

                       aKe[3*i]=1;
                       aKe[((ysize*k)+(j))*3+1]=1;
                       aKe[((ysize*k)+(j))*3+2]=1;
                       aKh[3*i]=1;
                       aKh[((ysize*k)+(j))*3+1]=1;
                       aKh[((ysize*k)+(j))*3+2]=1;
                       }
               }
       }
}






//--------------------------------------------------------------------------------------------
//
// Initialize Objects
// For Patch
//

// Read Input Files
// Read Input file 2
   pFile2 = fopen ("object_parameters.txt","r");

   while (fscanf(pFile2, "%s %f %f %f %f %f %f %f %f %f", op, &op2, &op3, &op4, &op5, &op6, &op7,
&op8, &op9, &op10) != EOF) {
     if (strcmp(op,"//")==0) {
} else if (strcmp(op,"pt")==0) {

   // pt is port definition
   // 1-3 start x, y, z
   // 4-6 stop x, y, z
   px1=(int) op2;
   py1=(int) op3;
   pz1=(int) op4;
   px2=(int) op5;
   py2=(int) op6;
   pz2=(int) op7;
   printf("Port - %d %d %d %d %d %d\n", px1, py1, pz1, px2, py2, pz2);

} else if (strcmp(op,"bx")==0) {

    printf("Box - %f %f %f %f %f %f %f %f\n", op2, op3, op4, op5, op6, op7, op8, op9);
   // bx is a box
   // 8 Params
   // 1-3 start x, y, z
   // 4-6 stop x, y, z
   // 7 epsilon
   // 8 sigma
    //    Cexh(i,j,k)=(2*ce)/(epsr(mt)+(ce*sigmae(mt)));
    //    Cexe(i,j,k)=(epsr(mt)-(ce*sigmae(mt)))/(epsr(mt)+(ce*sigmae(mt)));


   for (k=(s0+(int) op4); k<(s0+(int) op7); k++) {
           for (j=(s0+(int) op3); j<(s0+(int) op6); j++) {
                   for (i=(s0+(int) op2); i<(s0+(int) op5); i++) {
           aCeh[((xsize*ysize*k)+(xsize*j)+i)*3]        =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3]    =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3]    =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+1]      =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]    =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]      =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]    =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(2*ce)/(op8+(ce*op9));

   aCee[((xsize*ysize*k)+(xsize*j)+i)*3]        =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3]    =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3]    =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(op8-(ce*op9))/(op8+(ce*op9));

   aCee[((xsize*ysize*k)+(xsize*j)+i)*3+1]      =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]    =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]      =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]    =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(op8-(ce*op9))/(op8+(ce*op9));


                   }
           }
   }

 } else if (strcmp(op,"sh")==0) {

   printf("Sheet - %f %f %f %f %f %f %f %f\n", op2, op3, op4, op5, op6, op7, op8, op9);
   // sh is a *-plane sheet
   // If 2 x's, y's, or z's dont match this does nothing
   // 8 Params
   // 1-3 start x, y, z
   // 4-6 stop x, y, z
   // 7 epsilon
   // 8 sigma
    //    Cexh(i,j,k)=(2*ce)/(epsr(mt)+(ce*sigmae(mt)));
    //    Cexe(i,j,k)=(epsr(mt)-(ce*sigmae(mt)))/(epsr(mt)+(ce*sigmae(mt)));

   if ((int) op2 == (int) op5) {
           // X-plane sheet
           i=s0+(int) op2;
           for (j=(s0+(int) op3); j<(s0+(int) op6); j++) {
                   for (k=(s0+(int) op4); k<(s0+(int) op7); k++) {
           aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+1]      =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(2*ce)/(op8+(ce*op9));

           aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]      =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(2*ce)/(op8+(ce*op9));

   aCee[((xsize*ysize*k)+(xsize*j)+i)*3+1]      =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]      =(op8-(ce*op9))/(op8+(ce*op9));
   aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(op8-(ce*op9))/(op8+(ce*op9));

                   }
           }
   }

   if ((int) op3 == (int) op6) {
           // Y-plane sheet
           j=s0+(int) op3;
           for (k=(s0+(int) op4); k<(s0+(int) op7); k++) {
                   for (i=(s0+(int) op2); i<(s0+(int) op5); i++) {
           aCeh[((xsize*ysize*k)+(xsize*j)+i)*3]      =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(2*ce)/(op8+(ce*op9));

           aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(2*ce)/(op8+(ce*op9));

           aCee[((xsize*ysize*k)+(xsize*j)+i)*3]      =(op8-(ce*op9))/(op8+(ce*op9));
           aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(op8-(ce*op9))/(op8+(ce*op9));

           aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(op8-(ce*op9))/(op8+(ce*op9));
           aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(op8-(ce*op9))/(op8+(ce*op9));

                   }
           }
   }

   if ((int) op4 == (int) op7) {
           // Z-plane sheet
           k=s0+(int) op4;
           for (j=(s0+(int) op3); j<(s0+(int) op6); j++) {
                   for (i=(s0+(int) op2); i<(s0+(int) op5); i++) {
           aCeh[((xsize*ysize*k)+(xsize*j)+i)*3]      =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(2*ce)/(op8+(ce*op9));

           aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(2*ce)/(op8+(ce*op9));
           aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(2*ce)/(op8+(ce*op9));

           aCee[((xsize*ysize*k)+(xsize*j)+i)*3]      =(op8-(ce*op9))/(op8+(ce*op9));
           aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(op8-(ce*op9))/(op8+(ce*op9));

           aCee[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(op8-(ce*op9))/(op8+(ce*op9));
           aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(op8-(ce*op9))/(op8+(ce*op9));

                   }
                }

        }

     } else if (strcmp(op,"sr")==0) {

        printf("Source - %f %f %f %f %f %f %f %f\n", op2, op3, op4, op5, op6, op7, op8, op9,
op10);
        // sr is a *-plane source
        // If 2 x's, y's, or z's dont match this does nothing
        // Z directed right now
        // 8 Params
        // 1-3 start x, y, z
        // 4-6 stop x, y, z
        // 7 Resistance
        // 8 Voltage
         // 9 Epsilon

        if ((int) op2 == (int) op5) {
                // X-plane sheet

                R = op8 * (op6 - op3 + 1);
                R = R / (op7 - op4);

                i=s0+(int) op2;
                for (j=(s0+(int) op3); j<(s0+(int) op6+1); j++) {
                        for (k=(s0+(int) op4); k<(s0+(int) op7); k++) {

        aCes[((xsize*ysize*k)+(xsize*j)+i)*3+2]=op9*(2*dt/(R*dx*dy))/((2*op10*eps0)+((dt*dz)/(R*d
x*dy)));

        aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]=((2*op10*eps0) - ((dz*dt)/(dx*dy*R)))
/((2*op10*eps0) + ((dz*dt)/(dx*dy*R)));

        aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]=(2*dt) / ( (2*op10*eps0) + ((dt*dz) /
(dx*dy*R)));
                        }
                }
        }

        if ((int) op3 == (int) op6) {
                // Y-plane sheet
                R = op8 * (op6 - op3 + 1);
                R = R / (op7 - op4);

                j=s0+(int) op3;
                for (i=(s0+(int) op2); i<(s0+(int) op5+1); i++) {
                        for (k=(s0+(int) op4); k<(s0+(int) op7); k++) {

        aCes[((xsize*ysize*k)+(xsize*j)+i)*3+2]=op9*(2*dt/(R*dx*dy))/((2*op10*eps0)+((dt*dz)/(R*d
x*dy)));
                                aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]=((2*op10*eps0) -
((dz*dt)/(dx*dy*R))) /((2*op10*eps0) + ((dz*dt)/(dx*dy*R)));
                                aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]=(2*dt) / ( (2*op10*eps0) +
((dt*dz) / (dx*dy*R)));
                        }
                }
        }
        if ((int) op4 == (int) op7) {
                // Z-plane sheet
                // Not Implemented
        }

     }


   }
   fclose(pFile2);
```

```
// PEC Walls  Test (Use 100x45x45)

       for (k=0; k<zsize-1; k++) {
              for (j=0; j<ysize-1; j++) {
                     i=0;
                     aCeh[((xsize*ysize*k)+(xsize*j)+i)*3]    =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(2*ce)/(1+(ce*1e30));

                     aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(2*ce)/(1+(ce*1e30));

                     aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(2*ce)/(1+(ce*1e30));

              aCee[((xsize*ysize*k)+(xsize*j)+i)*3]    =(1-(ce*1e30))/(1+(ce*1e30));
              aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
              aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
              aCee[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(1-(ce*1e30))/(1+(ce*1e30));

              aCee[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(1-(ce*1e30))/(1+(ce*1e30));
              aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
              aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
              aCee[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(1-(ce*1e30))/(1+(ce*1e30));

              aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(1-(ce*1e30))/(1+(ce*1e30));
              aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
              aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
              aCee[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(1-(ce*1e30))/(1+(ce*1e30));


              i=xsize-2;
                     aCeh[((xsize*ysize*k)+(xsize*j)+i)*3]    =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(2*ce)/(1+(ce*1e30));

                     aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(2*ce)/(1+(ce*1e30));

                     aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(2*ce)/(1+(ce*1e30));
                     aCeh[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(2*ce)/(1+(ce*1e30));
```

```
aCee[((xsize*ysize*k)+(xsize*j)+i)*3]    =(1-(ce*1e30))/(1+(ce*1e30));
aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
aCee[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(1-(ce*1e30))/(1+(ce*1e30));

aCee[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(1-(ce*1e30))/(1+(ce*1e30));
aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
aCee[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(1-(ce*1e30))/(1+(ce*1e30));

aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(1-(ce*1e30))/(1+(ce*1e30));
aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
aCee[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(1-(ce*1e30))/(1+(ce*1e30));
          }
}

for (i=0; i<xsize-1; i++) {
          for (k=0; k<zsize-1; k++) {
                    j=0;
          aCeh[((xsize*ysize*k)+(xsize*j)+i)*3]    =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(2*ce)/(1+(ce*1e30));

          aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(2*ce)/(1+(ce*1e30));

          aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(2*ce)/(1+(ce*1e30));

          aCee[((xsize*ysize*k)+(xsize*j)+i)*3]    =(1-(ce*1e30))/(1+(ce*1e30));
          aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
          aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
          aCee[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(1-(ce*1e30))/(1+(ce*1e30));

          aCee[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(1-(ce*1e30))/(1+(ce*1e30));
          aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
          aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
          aCee[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(1-(ce*1e30))/(1+(ce*1e30));

          aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(1-(ce*1e30))/(1+(ce*1e30));
          aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
          aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
          aCee[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(1-(ce*1e30))/(1+(ce*1e30));

          j=ysize-2;

          aCeh[((xsize*ysize*k)+(xsize*j)+i)*3]    =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(2*ce)/(1+(ce*1e30));

          aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(2*ce)/(1+(ce*1e30));

          aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(2*ce)/(1+(ce*1e30));
          aCeh[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(2*ce)/(1+(ce*1e30));
```

```
        aCee[((xsize*ysize*k)+(xsize*j)+i)*3]    =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(1-(ce*1e30))/(1+(ce*1e30));

        aCee[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(1-(ce*1e30))/(1+(ce*1e30));

        aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(1-(ce*1e30))/(1+(ce*1e30));
        }
}

for (i=0; i<xsize-1; i++) {
        for (j=0; j<ysize-1; j++) {
                k=0;
        aCeh[((xsize*ysize*k)+(xsize*j)+i)*3]    =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(2*ce)/(1+(ce*1e30));

        aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(2*ce)/(1+(ce*1e30));

        aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(2*ce)/(1+(ce*1e30));

        aCee[((xsize*ysize*k)+(xsize*j)+i)*3]    =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(1-(ce*1e30))/(1+(ce*1e30));

        aCee[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(1-(ce*1e30))/(1+(ce*1e30));

        aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
        aCee[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(1-(ce*1e30))/(1+(ce*1e30));

        k=zsize-2;

        aCeh[((xsize*ysize*k)+(xsize*j)+i)*3]    =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(2*ce)/(1+(ce*1e30));

        aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(2*ce)/(1+(ce*1e30));

        aCeh[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(2*ce)/(1+(ce*1e30));
        aCeh[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(2*ce)/(1+(ce*1e30));
```

```
            aCeh[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(2*ce)/(1+(ce*1e30));


            aCee[((xsize*ysize*k)+(xsize*j)+i)*3]    =(1-(ce*1e30))/(1+(ce*1e30));
            aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
            aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3]  =(1-(ce*1e30))/(1+(ce*1e30));
            aCee[((xsize*ysize*(k+1))+(xsize*(j+1))+i)*3]=(1-(ce*1e30))/(1+(ce*1e30));

            aCee[((xsize*ysize*k)+(xsize*j)+i)*3+1]    =(1-(ce*1e30))/(1+(ce*1e30));
            aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
            aCee[((xsize*ysize*(k+1))+(xsize*j)+i)*3+1]  =(1-(ce*1e30))/(1+(ce*1e30));
            aCee[((xsize*ysize*(k+1))+(xsize*j)+i+1)*3+1]=(1-(ce*1e30))/(1+(ce*1e30));

            aCee[((xsize*ysize*k)+(xsize*j)+i)*3+2]    =(1-(ce*1e30))/(1+(ce*1e30));
            aCee[((xsize*ysize*k)+(xsize*j)+i+1)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
            aCee[((xsize*ysize*k)+(xsize*(j+1))+i)*3+2]  =(1-(ce*1e30))/(1+(ce*1e30));
            aCee[((xsize*ysize*k)+(xsize*(j+1))+i+1)*3+2]=(1-(ce*1e30))/(1+(ce*1e30));

            }
        }





// -----------------------------------------------------------------------------------------
-
// PML Init

// Initialize 1D Arrays
if (1==1) {
// X Arrays
for (i=2; i<12; i++) {
        ii=(float) i;
        ii=12-ii;

        sig1=pow(((ii-0.5)/iPML),m)*sigmax;
        sig2=(mu0/eps0)*pow(((ii)/iPML),m)*sigmax;

        a1=pow(((iPML-(ii-1+.5))/iPML),m)*amax;
        a2=(mu0/eps0)*pow(((iPML-(ii-1))/iPML),m)*amax;

        k1=1+(kappa-1)*pow(((ii-0.5)/iPML),m);
        k2=1+(kappa-1)*pow(((ii)/iPML),m);

        aKe[(i+1)*3]=1/k1;
        aKh[i*3]=1/k2;

        aKe[(xsize-i-1)*3]=1/k1;
        aKh[(xsize-i-1)*3]=1/k2;

        aBe[(i+1)*3]=exp((-dt/eps0)*((sig1/k1)+a1));
        aCe[(i+1)*3]=((sig1/dx)/((sig1*k1)+(a1*k1*k1)))*(exp((-dt/eps0)*((sig1/k1)+a1))-1);


        aBe[(xsize-i-1)*3]=exp((-dt/eps0)*((sig1/k1)+a1));
        aCe[(xsize-i-1)*3]=((sig1/dx)/((sig1*k1)+(a1*k1*k1)))*(exp((-dt/eps0)*((sig1/k1)+a1))-1);

        aBh[i*3]=exp((-dt/mu0)*((sig2/k2)+a2));
        aCh[i*3]=((sig2/dx)/((sig2*k2)+(a2*k2*k2)))*(exp((-dt/mu0)*((sig2/k2)+a2))-1);

        aBh[(xsize-i-1)*3]=exp((-dt/mu0)*((sig2/k2)+a2));
        aCh[(xsize-i-1)*3]=((sig2/dx)/((sig2*k2)+(a2*k2*k2)))*(exp((-dt/mu0)*((sig2/k2)+a2))-1);


}
```

```
// Y Arrays
if (1==1) {
for (k=0; k<zsize; k++) {
	for (j=2; j<12; j++) {
		ii=(float) j;
		ii=12-ii;

		sig1=pow(((ii-0.5)/iPML),m)*sigmay;
		sig2=(mu0/eps0)*pow(((ii)/iPML),m)*sigmay;

		a1=pow(((iPML-(ii-1+.5))/iPML),m)*amax;
		a2=(mu0/eps0)*pow(((iPML-(ii-1))/iPML),m)*amax;

		k1=1+(kappa-1)*pow(((ii-0.5)/iPML),m);
		k2=1+(kappa-1)*pow(((ii)/iPML),m);


		aKe[((k*ysize)+j+1)*3+1]=1/k1;
		aKh[((k*ysize)+j)*3+1]=1/k2;

		aKe[((k*ysize)+(ysize-j)-1)*3+1]=1/k1;
		aKh[((k*ysize)+(ysize-j)-1)*3+1]=1/k2;

		aBe[((k*ysize)+j+1)*3+1]=exp((-dt/eps0)*((sig1/k1)+a1));
		aCe[((k*ysize)+j+1)*3+1]=((sig1/dy)/((sig1*k1)+(a1*k1*k1)))*(exp((-
dt/eps0)*((sig1/k1)+a1))-1);

		aBe[((k*ysize)+(ysize-j)-1)*3+1]=exp((-dt/eps0)*((sig1/k1)+a1));
		aCe[((k*ysize)+(ysize-j)-1)*3+1]=((sig1/dy)/((sig1*k1)+(a1*k1*k1)))*(exp((-
dt/eps0)*((sig1/k1)+a1))-1);

		aBh[((k*ysize)+j)*3+1]=exp((-dt/mu0)*((sig2/k2)+a2));
		aCh[((k*ysize)+j)*3+1]=((sig2/dy)/((sig2*k2)+(a2*k2*k2)))*(exp((-
dt/mu0)*((sig2/k2)+a2))-1);

		aBh[((k*ysize)+(ysize-j)-1)*3+1]=exp((-dt/mu0)*((sig2/k2)+a2));
		aCh[((k*ysize)+(ysize-j)-1)*3+1]=((sig2/dy)/((sig2*k2)+(a2*k2*k2)))*(exp((-
dt/mu0)*((sig2/k2)+a2))-1);
	}
}
}

// Z Arrays
if (1==1) {
for (k=2; k<12; k++) {
	for (j=0; j<ysize; j++) {

		ii=(float) k;
		ii=12-ii;

		sig1=pow(((ii-0.5)/iPML),m)*sigmaz;
		sig2=(mu0/eps0)*pow(((ii)/iPML),m)*sigmaz;

		a1=pow(((iPML-(ii-1+.5))/iPML),m)*amax;
		a2=(mu0/eps0)*pow(((iPML-(ii-1))/iPML),m)*amax;

		k1=1+(kappa-1)*pow(((ii-0.5)/iPML),m);
		k2=1+(kappa-1)*pow(((ii)/iPML),m);

		aKe[(((k+1)*ysize)+j)*3+2]=1/k1;
		aKh[(((k)*ysize)+j)*3+2]=1/k2;

		aKe[(((zsize-k-1)*ysize)+j)*3+2]=1/k1;
		aKh[(((zsize-k-1)*ysize)+j)*3+2]=1/k2;
```

```
               aBe[(((k+1)*ysize)+j)*3+2]=exp((-dt/eps0)*((sig1/k1)+a1));
               aCe[(((k+1)*ysize)+j)*3+2]=((sig1/dz)/((sig1*k1)+(a1*k1*k1)))*(exp((-
dt/eps0)*((sig1/k1)+a1))-1);

               aBe[(((zsize-k-1)*ysize)+j)*3+2]=exp((-dt/eps0)*((sig1/k1)+a1));
               aCe[(((zsize-k-1)*ysize)+j)*3+2]=((sig1/dz)/((sig1*k1)+(a1*k1*k1)))*(exp((-
dt/eps0)*((sig1/k1)+a1))-1);

               aBh[(((k)*ysize)+j)*3+2]=exp((-dt/mu0)*((sig2/k2)+a2));
               aCh[(((k)*ysize)+j)*3+2]=((sig2/dz)/((sig2*k2)+(a2*k2*k2)))*(exp((-
dt/mu0)*((sig2/k2)+a2))-1);

               aBh[(((zsize-k-1)*ysize)+j)*3+2]=exp((-dt/mu0)*((sig2/k2)+a2));
               aCh[(((zsize-k-1)*ysize)+j)*3+2]=((sig2/dz)/((sig2*k2)+(a2*k2*k2)))*(exp((-
dt/mu0)*((sig2/k2)+a2))-1);
       }
}
}

}
// End PML Init
// --------------------------------------------------------------------------------------------
-


               pFile3 = fopen ("port.txt","wt");
//////
/// GPU Start
//////
outsize2= ((zsize-2)*ysize);
i1=(float) (xsize-1);
i2=(float) (xsize-2);
o1=(float) ysize;
o2=(float)  (outsize-ysize);
       {

       //      iter float2 it<outsize2, i2> = iter( float2((float) 1, o1), float2( i1 , o2) );
               iter float2 it<outsize, xsize> = iter( float2(0, 0), float2( (float) xsize ,
(float)  outsize) );

               float3 Obs<1,1>;
               float3 E<outsize, xsize>;
               float3 H<outsize, xsize>;
               float3 o_E<outsize, xsize>;
               float3 o_H<outsize, xsize>;
               float3 Cee<outsize, xsize>;
               float3 Ceh<outsize, xsize>;
               float3 Ces<outsize, xsize>;

               //PML Streams

               float3 psiH1<outsize, xsize>, psiH2<outsize, xsize>;

               float3 o_psiH1<outsize, xsize>, o_psiH2<outsize, xsize>;

               float3 psiE1<outsize, xsize>, psiE2<outsize, xsize>;

               float3 o_psiE1<outsize, xsize>, o_psiE2<outsize, xsize>;

               float3 Be<outsize>, Ce<outsize>;

               float3 Bh<outsize>, Ch<outsize>;

               float3 Ke<outsize>, Kh<outsize>;

               //Input Arrays
               streamRead(E, aE);
```

```
                streamRead(H, aH);

                streamRead(o_E, aE);
                streamRead(o_H, aH);

                streamRead(Cee, aCee);
                streamRead(Ceh, aCeh);
                streamRead(Ces, aCes);

                streamRead(psiH1, aE);
                streamRead(psiH2, aE);

                streamRead(o_psiH1, aE);
                streamRead(o_psiH2, aE);


                streamRead(psiE1, aE);
                streamRead(psiE2, aE);

                streamRead(o_psiE1, aE);
                streamRead(o_psiE2, aE);



                streamRead(Bh, aBh);
                streamRead(Ch, aCh);

                streamRead(Be, aBe);
                streamRead(Ce, aCe);

                streamRead(Ke, aKe);
                streamRead(Kh, aKh);


                //Do the requested number of iterations
                for(i=0; i<N; i++){
                        //We can't use the input and output buffers without hosing things
                        //so we'll need to "ping-pong" between them
                 if ((i+1)%100==0) {
                        printf("%d Time Steps Complete\n", i);
                }

                 if(i%2==0) {

                        process_field_H(H, E, Kh, Bh, Ch, XE, XH, dx, dy, dz,(float) ysize, it,
                                        psiH1, psiH2, o_psiH1, o_psiH2, o_H);

                        process_field_E(E, o_H, Cee, Ceh, Ces, Ke, Be, Ce, dx, dy, dz,(float)
ysize, gauss[i], it,
                                        psiE1, psiE2, o_psiE1, o_psiE2, o_E);


                } else {

                        process_field_H(o_H, o_E, Kh, Bh, Ch, XE, XH, dx, dy, dz,(float) ysize,
it,
                                        o_psiH1, o_psiH2, psiH1, psiH2, H);

                        process_field_E(o_E, H, Cee, Ceh, Ces, Ke, Be, Ce, dx, dy, dz,(float)
ysize, gauss[i], it,
                                        o_psiE1, o_psiE2, psiE1, psiE2, E);


                }


////
// --- Export Port Values
////
               if (1==1) {
                              vObs=0;
                              l=s0+px1;
                              for (j=(s0+py1); j<(s0+py2+1); j++) {
                                     for (k=(s0+pz1); k<(s0+pz2); k++) {

       Copy(E.domain(int2(l,j+(ysize*k)),int2(l,j+(ysize*k))),Obs);
                                            streamWrite(Obs,aObs);
                                            vObs=vObs-dz*aObs[2];
                                     }
                              }
                              fprintf (pFile3,"%e, ",(1/((float)py2-(float)py1+1))*vObs);

                              //Top
                              vObs=0;
                              k=(s0+pz2);
                              for (j=(s0+py1); j<(s0+py2+1); j++) {

       Copy(H.domain(int2(l,j+(ysize*k)),int2(l,j+(ysize*k))),Obs);
                                     streamWrite(Obs,aObs);
                                     vObs=vObs+dy*aObs[1];

                              }

                              //Bottom
                              k=(s0+pz2-1);
                              for (j=(s0+py1); j<(s0+py2+1); j++) {

       Copy(H.domain(int2(l,j+(ysize*k)),int2(l,j+(ysize*k))),Obs);
                                     streamWrite(Obs,aObs);
                                     vObs=vObs-dy*aObs[1];

                              }

                              //Right
                              k=(s0+pz2);
                              j=s0+py2;


       Copy(H.domain(int2(l,j+(ysize*k)),int2(l,j+(ysize*k))),Obs);
                                     streamWrite(Obs,aObs);
                                     vObs=vObs-dz*aObs[2];

                              //Left
                              j=s0+py1-1;


       Copy(H.domain(int2(l,j+(ysize*k)),int2(l,j+(ysize*k))),Obs);
                                     streamWrite(Obs,aObs);
                                     vObs=vObs+dz*aObs[2];

                              fprintf (pFile3," %e \n ",vObs);

                       }




////
// ---  End Export Port Values
////

               }
```

```
////
// ---  Write Streams Back Out and Close Port Observation File
////



                        streamWrite(E, aE);
                        streamWrite(H, aH);

                fclose (pFile3);





////
// --- Export Final Fields
////
        if (1 == 1) {
                pFile = fopen ("Ezdata.txt","wt");
                pFile2 = fopen ("Hxdata.txt","wt");
                pFile3 = fopen ("Hydata.txt","wt");
                pFile4 = fopen ("Hzdata.txt","wt");
                pFile5 = fopen ("Exdata.txt","wt");
                pFile6 = fopen ("Eydata.txt","wt");
                k=s0+pz2;

///             for (k=0; k<zsize; k++) {
                        for (j=0; j<ysize; j++) {
                                for (i=0; i<xsize; i++) {
                                fprintf (pFile,"%e ",aE[((xsize*ysize*k)+(xsize*j)+i)*3+2]);
                                 fprintf (pFile2,"%e ",aH[((xsize*ysize*k)+(xsize*j)+i)*3]);
                                fprintf (pFile3,"%e ",aH[((xsize*ysize*k)+(xsize*j)+i)*3+1]);
                                fprintf (pFile4,"%e ",aH[((xsize*ysize*k)+(xsize*j)+i)*3+2]);
                                 fprintf (pFile5,"%e ",aE[((xsize*ysize*k)+(xsize*j)+i)*3]);
                                fprintf (pFile6,"%e ",aE[((xsize*ysize*k)+(xsize*j)+i)*3+1]);
                        }


                        fprintf (pFile,"\n");
                        fprintf (pFile2,"\n");
                        fprintf (pFile3,"\n");
                        fprintf (pFile4,"\n");
                        fprintf (pFile5,"\n");
                        fprintf (pFile6,"\n");

                }

                fclose (pFile);
                fclose (pFile5);
                fclose (pFile6);
                fclose (pFile2);
                fclose (pFile4);
                fclose (pFile3);
        }

////
// --- End Export Fields
////


}
        printf("Run Complete\n");
        return 0;
}
```

APPENDIX F

```
__global__
static void update_E  (float* Ex, float* Ey, float* Ez,
                              float* Hx, float* Hy, float* Hz,
                              float* CExe, float* CEye, float* CEze,
                              float* CExhz, float* CEyhx, float* CEzhy,
                              float* CExhy, float* CEyhz, float* CEzhx,
                              float* CExs, float* CEys, float* CEzs,
                              float V,
                              unsigned int Blocks_Y, float invBlocks_Y,
                              int NX, int NY, int NZ,
                              float* Vout, int iteration, int ii, int jj, int kk)
{
    unsigned int blockIdx_z = __float2uint_rd(blockIdx.y * invBlocks_Y);
    unsigned int blockIdx_y = blockIdx.y - __umul24(blockIdx_z, Blocks_Y);
    unsigned int tx = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    unsigned int ty = __umul24(blockIdx_y, blockDim.y) + threadIdx.y;
    unsigned int tz = __umul24(blockIdx_z, blockDim.z) + threadIdx.z;

        if ((tx >= NX) || (ty >= NY) || (tz >= NZ))
                return;

    // Locations of Indicies
    long int it = tz * NX * NY + ty * NX + tx;
    long int itxp1 = tz * NX * NY + ty * NX + tx+1;
    long int itxm1 = tz * NX * NY + ty * NX + tx-1;
    long int ityp1 = tz * NX * NY + (ty+1) * NX + tx;
    long int itym1 = tz * NX * NY + (ty-1) * NX + tx;
    long int itzp1 = (tz+1) * NX * NY + ty * NX + tx;
    long int itzm1 = (tz-1) * NX * NY + ty * NX + tx;

        if ((tx > 10) && (tx < 40) &&(ty > 10) && (ty < 38) &&(tz > 10) && (tz < 66)) {

        if ((tx < NX-1) && (ty > 0) && (ty < NY-1) && (tz > 0) && (tz < NZ-1)) {
                Ex[it] =
                          (CExe[it] * Ex[it]
                          + CExhz[it]*(Hz[it] - Hz[itym1])
                          - CExhy[it]*(Hy[it] - Hy[itzm1])) * (1-CExs[it])
                          + V*CExs[it];
    }

    if ((tx > 0) && (tx < NX-1) && (ty < NY-1) && (tz > 0) && (tz < NZ-1)) {
                Ey[it] =
                          (CEye[it] * Ey[it]
                          + CEyhx[it]*(Hx[it] - Hx[itzm1])
                          - CEyhz[it]*(Hz[it] - Hz[itxm1])) * (1-CEys[it])
                          + V*CEys[it];
        }

        if ((tx > 0) && (tx < NX-1) && (ty > 0) && (ty < NY-1) && (tz < NZ-1)) {
                Ez[it] =
                          (CEze[it] * Ez[it]
                          + CEzhy[it]*(Hy[it] - Hy[itxm1])
                          - CEzhx[it]*(Hx[it] - Hx[itym1])) * (1-CEzs[it])
                          + V*CEzs[it];
    }
    }

  /*  if ((tx == ii) && (ty == jj) && (tz == kk))
    {
                Vout[iteration] = Ez[it];
    }
   */
}

__global__
static void update_H  (float* Ex, float* Ey, float* Ez,
                              float* Hx, float* Hy, float* Hz,
                              float* CHxh, float* CHyh, float* CHzh,
                              float* CHxey, float* CHyez, float* CHzex,
                              float* CHxez, float* CHyex, float* CHzey,
                              unsigned int Blocks_Y, float invBlocks_Y,
                              int NX, int NY, int NZ)
{
    unsigned int blockIdx_z = __float2uint_rd(blockIdx.y * invBlocks_Y);
    unsigned int blockIdx_y = blockIdx.y - __umul24(blockIdx_z, Blocks_Y);
    unsigned int tx = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    unsigned int ty = __umul24(blockIdx_y, blockDim.y) + threadIdx.y;
    unsigned int tz = __umul24(blockIdx_z, blockDim.z) + threadIdx.z;

        if ((tx >= NX) || (ty >= NY) || (tz >= NZ))
                return;

    // Locations of Indicies
    long int it = tz * NX * NY + ty * NX + tx;
    long int itxp1 = tz * NX * NY + ty * NX + tx+1;
    long int itxm1 = tz * NX * NY + ty * NX + tx-1;
    long int ityp1 = tz * NX * NY + (ty+1) * NX + tx;
    long int itym1 = tz * NX * NY + (ty-1) * NX + tx;
    long int itzp1 = (tz+1) * NX * NY + ty * NX + tx;
    long int itzm1 = (tz-1) * NX * NY + ty * NX + tx;


//      Ex[it]=V*Evx[it];
if ((tx > 10) && (tx < 40) &&(ty > 10) && (ty < 38) &&(tz > 10) && (tz < 66)) {

        if ((tx < NX-1) && (tx > 0) && (ty < NY-1) && (tz < NZ-1)) {
                Hx[it]  = CHxh[it] * Hx[it]
                              + CHxey[it] * (Ey[itzp1] - Ey[it])
                              - CHxez[it] * (Ez[ityp1] - Ez[it]);
    }

    if ((ty > 0) && (tx < NX-1) && (ty < NY-1) && (tz < NZ-1)) {
                Hy[it]  = CHyh[it] * Hy[it]
                              + CHyez[it] * (Ez[itxp1] - Ez[it])
                              - CHyex[it] * (Ex[itzp1] - Ex[it]);
        }

        if ((tx < NX-1) && (tz > 0) && (ty < NY-1) && (tz < NZ-1)) {
                Hz[it]  = CHzh[it] * Hz[it]
                              + CHzex[it] * (Ex[ityp1] - Ex[it])
                              - CHzey[it] * (Ey[itxp1] - Ey[it]);
    }

   }
}

extern "C"
void launch_EHSTEP(    float *d_Ex, float *d_Ey, float *d_Ez,
                                      float *d_Hx, float *d_Hy, float *d_Hz,
                                      float *d_CExe, float *d_CEye, float *d_CEze,
                                      float *d_CExhz, float *d_CEyhx, float *d_CEzhy,
                                      float *d_CExhy, float *d_CEyhz, float *d_CEzhx,
                                      float *d_CExs, float *d_CEys, float *d_CEzs,
                                      float V, int blocksInY, int nx, int ny, int nz)
{
        update_E<<<dimGrid, dimBlock>>>(d_Ex, d_Ey, d_Ez,
                                              d_Hx, d_Hy, d_Hz,
                                              d_CExe, d_CEye, d_CEze,
                                              d_CExhz, d_CEyhx, d_CEzhy,
                                              d_CExhy, d_CEyhz, d_CEzhx,
                                              d_CExs, d_CEys, d_CEzs,
                                              V,
                                      blocksInY, 1.0f/(float)blocksInY, nx, ny, nz);
          cudaThreadSynchronize();

           update_H<<<dimGrid, dimBlock>>>(d_Ex, d_Ey, d_Ez,
                                      d_Hx, d_Hy, d_Hz,
                                      d_CHxh, d_CHyh, d_CHzh,
                                      d_CHxey, d_CHyez, d_CHzex,
                                      d_CHxez, d_CHyex, d_CHzey,
                                      blocksInY, 1.0f/(float)blocksInY, nx, ny, nz);
        cudaThreadSynchronize();
}
```

VITA

Matthew Joseph Inman

College/University	**Major**	**Degree &Year**
The University of Mississippi	Electrical Engineering	B.S., 2000
The University of Mississippi	Engineering Science	M.S., 2004

Biography

M. J. Inman, "An Evaluation of Key Interval Measurements with Macromedia Authorware," The IEEE SoutheastCon 2000 Symposium, Lexington, Kentucky, April 2000.

M. J. Inman, A. Z. Elsherbeni, C. E. Smith, and Kai-Fong Lee, "Analysis of Printed Microstrip loop Antenna," Journal of The Mississippi Academy of Sciences, p. , February 2002.

M. J. Inman, A. Z. Elsherbeni, C. E. Smith, and Kai-Fong Lee, "FDTD Analysis of Rectangular Microstrip Loop Antennas," The IEEE SoutheastCon 2002 Symposium, Colombia, South Carolina, pp. - , April 2002.

M. J. Inman, A. Z. Elsherbeni, and C. E. Smith, "Finite Difference Time Domain Analysis of Moving Objects," The 18th Annual Review of Progress in Applied Computational Electromagnetics, ACES'02, Monterey, California, pp. 55-62, March 2002.

A. Z. Elsherbeni, Matthew J. Inman, and Robert Christopher-Lee Riley, "Antenna Design and Radiation Visualization," Journal of The Mississippi Academy of Sciences, p. 73 , February 2003.

A. Z. Elsherbeni, M. J. Inman, and C. Riley, "Antenna Design and Radiation Pattern Visualization," The 19th Annual Review of Progress in Applied Computational Electromagnetics, ACES'03, Monterey, California, March 2003.

A. Z. Elsherbeni, R. Christopher-Lee Riley, and Matthew J. Inman, "A Finite Difference Simulation Package for Quasi-Static Electromagnetic Analysis," Memphis Area Engineering Societies Conference (MAESC 2003), May 15, 2003.

M. J. Inman, A. Z. Elsherbeni, and C. E. Smith, Moving Object Analysis and Simulation Using

the Finite Difference Time Domain Technique," Memphis Area Engineering Societies Conference (MAESC 2003), May 15, 2003.

M. J. Inman, A. Z. Elsherbeni, and C. E. Smith, "Finite Difference Time Domain Simulation of Moving Objects," The 2003 IEEE Radar Conference, Huntsville, AL, May 2003.

M. J. Inman, A. Z. Elsherbeni, "Antenna Design and Radiation Pattern Visualization,", Applied Computational Electromagnetics Society (ACES), vol. 18, no. 4, pp. , November 2003

J. M. Earwood, M. J. Inman, A. Z. Elsherbeni, and C. E. Smith, "Bayesian Inference Techniques for The Design of Antenna Arrays," Journal of The Mississippi Academy of Sciences, p. 101, February 2004.

M. J. Inman, J. M. Earwood, A. Z. Elsherbeni and C. E. Smith, "Bayesian Optimization Techniques for Antenna Design," The 20th Annual Review of Progress in Applied Computational Electromagnetics, ACES'04, Syracuse, NY, April 2004.

J. M. Earwood, M. J. Inman, A. Z. Elsherbeni, and C. E. Smith, "Linear Array Design using Bayesian Parameter Estimation," The 2004 IEEE Radar Conference, Philadelphia, Pennsylvania, April 2004.

M. J. Inman, A. Elsherbeni,C. E. Smith, "Use Of Graphic Processing Units For General Scientific Computations," Journal of The Mississippi Academy of Sciences, February 2005.

M.J. Inman, C. Loo, and A. Elsherbeni, "Image processing and target identification using graphical processing units," Mississippi Academy of Sciences Annual Meeting, Vicksburg, MS, 22-24 February 2006.

M. J. Inman and A. Z. Elsherbeni, “3D FDTD Acceleration Using Graphical Processing Units”, “The 22nd Annual Review of Progress in Applied Computational Electromagnetics Society”, ACES'06, Miami, FL, USA, 12-16 March 2006..

M. J. Inman and A. Z. Elsherbeni, “Acceleration of field computations using graphical processing units,” The Twelfth Biennial IEEE Conference on Electromagnetic Field Computation CEFC 2006 Miami, FL, USA, April 30 - May 3, 2006.

M. J. Inman, A. Z. Elsherbeni, J. G. Maloney, and B. N. Baker, “Practical implementation of a CPML absorbing boundary for GPU accelerated FDTD technique,” The 23rd Annual Review of Progress in Applied Computational Electromagnetics Society, ACES'07, Verona, Italy, 19-23 March 2007.

M. J. Inman, A. Z. Elsherbeni, “Interactive GPU Based FDTD Simulations for Teaching Applications,” The 24rd Annual Review of Progress in Applied Computational

Electromagnetics Society, ACES'08, Niagara Falls, Canada, April 2008.

M. J. Inman, A. Z. Elsherbeni, "Optimization and Parameter Exploration Using GPU Based FDTD Solvers," 2008 IEEE MTT-S Int. Microwave Symp., Atlanta, June 2008.

M. J. Inman, A. Z. Elsherbeni, "MATLAB Graphical Interface for GPU Based FDTD Method," 2008 Asia-Pacific Symposium on EMC & 19th International Zurich Symposium on Electromagnetic Compatibility, Singapore, May 19-22, 2008.

A. Z. Elsherbeni, M. J. Inman, and V. Demir, " From Entertainment to Education: The Role of Video Card Accelerated Programs in Electromagnetics Education", 35th Annual Conference of The Association of Egyptian American Scholars, Cairo, Egypt, December 2008. (Invited)

M. J. Inman and A. Z. Elsherbeni, "Optimization of GPU Codes for FDTD Simulations," The 25th Annual Review of Progress in Applied Computational Electromagnetics Society , ACES'09, Monterey, California, March 2009.

M. J. Inman, A. Z. Elsherbeni, "GPU Acceleration of Linear Systems for Computational Electromagnetic Simulations," The 2009 IEEE International Symposium on Antennas and Propagation, June 2009.

M. J. Inman, A. Z. Elsherbeni, C. J. Reddy, "CUDA Based LU Decomposition Solvers for CEM Applications,", Applied Computational Electromagnetics Society (ACES), vol. 25, no. 4, pp. , April 2010

M. J. Inman and A. Z. Elsherbeni, "Optimizing Multiple GPU Simulations in CUDA," The 27th Annual Review of Progress in Applied Computational Electromagnetics Society , ACES'11, Williamsburg, Virginia, March 2011.

A. Z. Elsherbeni and Matthew J. Inman, Antenna Design and Visualization Using MATLAB, Scitech, January 2006.

Chapter on GPU Based FDTD Methods:
Atef Elsherbeni, Veysel Demir, and Fan Yang, FDTD Electromagnetic Simulations Using Matlab, SciTech Publishing Inc., 2009.