Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors

Samuel Adams[†] and Jason Payne
US Air Force Research Laboratory, Human Effectiveness Directorate (AFRL/HE), Brooks City-Base, TX
{samuel.adams, jason.payne}@brooks.af.mil

Rajendra Boppana
The University of Texas at San Antonio, Department of Computer Science, San Antonio, TX
boppana@cs.utsa.edu

Abstract

This paper presents a graphics processor based implementation of the Finite Difference Time Domain (FDTD), which uses a central finite differencing scheme for solving Maxwell's equations for electromagnetics. FDTD simulations can be very computationally expensive and require thousands of CPU hours to solve on traditional general purpose processors. Modern Graphics Processing Units (GPUs) found in desktop computers are programmable and are capable of much higher vector floating-point performance than general purpose CPUs.

This paper shows how GPUs can be used to greatly speedup FDTD simulations. The main objective is to leverage GPU processing power for FDTD update calculations and complete computationally expensive simulations in reasonable time. This allows researchers to simulate much longer pulse lengths and larger models than was possible in the past.

A new FDTD code was developed to leverage graphics processors using Linux, C, OpenGL, Cg, and commodity GeForce 7 series GPUs. The graphics hardware was accessed through standard OpenGL. The FDTD model space was then transferred to the GPU device memory through OpenGL textures and host readable via frame buffer objects exposed by the OpenGL 2.0 application programming interface (API). GPU fragment processors were utilized for the FDTD update computations via Cg fragment programs.

For models that were sufficiently large, greater than $(140)^3$ cells, the GPU performed FDTD update calculations at least 12 times faster than the execution of the same simulation on a contemporary multicore CPU from Intel or AMD. The use of GPUs shows great promise for high performance computing applications like FDTD that have high arithmetic intensity and limited or no data dependencies in computation streams. Until recently, to use GPUs as a co-processor, the normal CPU-based code needed to be rewritten extensively using special graphics programming language Cg and OpenGL APIs, which is difficult for non-graphics programmers. However, newer GPUs, such as NVIDIA's G80, provide unified shader models for programming GPU processing elements and APIs that allow compiler tools to allow direct programming of graphics hardware without extra intermediate graphics programming with OpenGL and Cg. Currently, a message passing interface-based parallel GPU FDTD code is being developed and benchmarked on a cluster of G80 GPUs.

1. Introduction

The FDTD method has long been a popular computational method for solving Maxwell's Equations for electromagnetics. FDTD can solve models with arbitrary geometries comprised of dielectric materials. Additionally, FDTD can easily handle sinusoidal, transient and pulsed sources, making it a useful tool for many problems in electromagnetics. However, simulations with large model spaces or long non-sinusoidal waveforms can require a tremendous amount of floating point calculations and run times of several months or more are possible even on large HPC systems. Even for small models of approximately 10^9 cells, a simulation of a millisecond duration waveform could easily take over a month even on 1,000 processors on a traditional high-performance computing (HPC) cluster. It is obvious that current FDTD codes running on existing HPC clusters are not viable for this class of problems, prompting a need to investigate methods of increasing computational speed for these simulations. In the past few years, GPUs have shown great promise for general purpose computation codes and methods, such as FDTD.

GPUs have high floating point performance, exceeding the performance of general purpose CPUs by many times, due to their specialized streaming architecture. In the past few years, programmability of

[†]General Dynamics Information Technology, Needham Heights, MA

GPUs and increased floating point precision has allowed GPUs to perform general purpose computations. We have developed new FDTD codes leveraging these commodity GPUs for performing scattered field update computations, and for sufficiently large model space, achieved speedup of many times.

The rest of the paper is organized as follows. First, an overview of using GPUs for general purpose computation is given. Next, a brief description of the FDTD method and a commonly used algorithm for FDTD simulations is given. Next, our GPU implementation of FDTD simulation code is described followed by performance comparison of CPU and GPU executions. Finally, conclusions and directions for further work are given.

2. GPUs for General Purpose Computation

For a few years now, GPUs have been used for general purpose computation[1]. In terms of raw floating-point computation capability, they are many times more powerful than general purpose CPUs and have much higher memory bandwidth. For example, a GeForce 8800 GTX can produce 345 peak GFLOPS and has an 86.4GB/sec memory interface, whereas an Intel Core 2 Extreme quad core processor at 2.66 GHz (1,066 MHz FSB) has a theoretical 21.3 peak GFLOPS and 10.7 GB/sec maximum memory bandwidth. GPUs are also improving their performance at a much faster rate; GPU performance doubles every 6 months compared to 18 months for CPUs.

GPUs are able to achieve this high performance at the cost of generality. GPUs are optimized for data parallelism, or in other words, to execute similar operations on large vectors or streams of data in parallel. Explicit data parallelism allows GPUs to largely exclude control logic, and data streaming with high-speed memory interfaces eliminates the need for large on-die caches. This frees up much of the GPU's die surface for computational units in exchange for large caches and complicated control logic; in the case of the GeForce 8800 GTX, there are 128 stream processors on the die. Data parallelism also requires computation streams to not intercommunicate or share memory, and branching logic is allowed only at high performance costs. Since data is streamed and not maintained in cache, it is also important that there is sufficient arithmetic intensity, or in other words, enough computation per word fetched to mask the memory latency. Due to these restrictions, many traditional algorithms cannot be directly ported to run on streaming architectures, but for algorithms that are data parallel and are arithmetically intense, great speed up can be achieved.

3. Finite Difference Time Domain

FDTD is used to solve Maxwell's equations for arbitrary model spaces. Indeed, FDTD allows us to solve models that would be difficult or impossible with analytical methods. FDTD is a direct time-domain solution to Maxwell's curl equations[2], which are given here below.

$$\frac{\partial \overline{E}}{\partial t} = \frac{1}{\varepsilon}\nabla \times \overline{H} - \frac{1}{\varepsilon}\left(\overline{J}_{\text{source}} + \sigma \overline{E}\right)$$

$$\frac{\partial \overline{H}}{\partial t} = -\frac{1}{\mu}\nabla \times \overline{E} - \frac{1}{\mu}\left(\overline{M}_{\text{source}} + \sigma^{*} \overline{H}\right)$$

In the FDTD scheme, Maxwell's curl equations are first scalarized into their x, y, and z field components. Then, centered finite difference expressions are used to approximate the spatial and time derivatives. Below is the resulting x-directed H field equation; the other 5 field components are similar.

$$\frac{Hx^{n+1/2}_{i,j,k} - Hx^{n+1/2}_{i,j,k}}{\Delta t} = \frac{1}{\mu}\left[\frac{Ey^{n}_{i,j,k+1/2} - Ey^{n}_{i,j,k-1/2}}{\Delta z} - \frac{Ez^{n}_{i,j+1/2,k} - Ez^{n}_{i,j-1/2,k}}{\Delta y} - \left(M_{\text{source}} + \sigma^{*} Hx^{n-1/2}_{i,j,k}\right)\right]$$

This method was first introduced in the original FDTD paper by Kane Yee[3]. In particular, he introduced the next important FDTD concept known as the Yee Space Grid, shown in Figure 1.

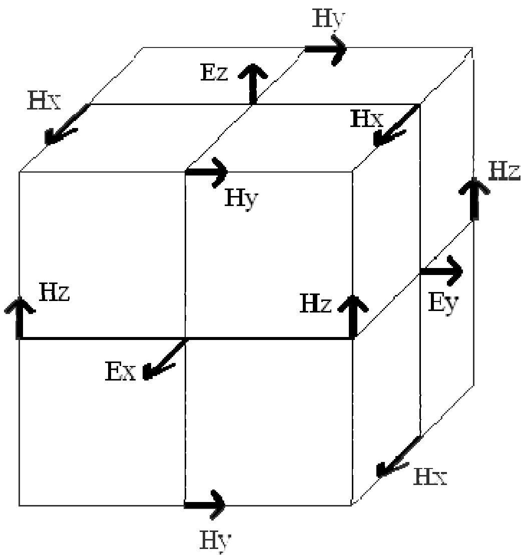

Figure 1. Yee's space-grid model

The key features to the Yee Space Grid relate to the staggering of the E and H fields. The E and H field are staggered to one another with respect to time by one half of the time step. E and H are centered in space such that each E field component is surrounded by 4 H field components and vice versa.

4. Basic FDTD Algorithm

The FDTD algorithm solves Maxwell's equations by first performing the E field update equations for each voxel at time step n, and then performing the H field update equations for each voxel at time step n+1/2[4]. The time resolution of the simulation is determined by the model's spatial resolution, and the number of time steps is determined by the waveform and duration of the source being modeled.

The problem space is stored in a three dimensional grid. Each cell in the grid is assigned a material type that has corresponding dielectric properties, and stores the x, y, and z components for both the E and H field initialized to 0. After initializing the model space, the basic FDTD algorithm used is shown in Figure 2.

```
for(n = 0; n < time_steps; n++){
  /* e field updates */
  for(k = 0; k < z_dim; k++){
    for(j = 0; j < y_dim; j++){
      for(i = 0; i <  x_dim; i++){
        update e_x[i][j][k]
        update e_y[i][j][k]
        update e_z[i][j][k]
      }
    }
  }
  /* h field updates */
  for(k = 0; k < z_dim; k++){
    for(j = 0; j < y_dim; j++){
      for(i = 0; i <  x_dim; i++){
        update h_x[i][j][k]
        update h_y[i][j][k]
        update h_z[i][j][k]
      }
    }
  }
}
```

Figure 2. High-level overview of FDTD algorithm

Most of the execution time is spent in the nested loops for E and H field updates. Individual voxel updates (performed by the computations in the inner two loops) are independent and can be executed in parallel. The computational complexity and memory requirements during execution can be estimated as follows.

$$\text{Computational Complexity} \propto \text{Number of time steps} \times \text{Number of cells} \times 6 \times \text{Operations/update}$$

The factor 6 in the above expression represents the number of field components that are updated in a 3D implementation. The field update calculations are as described in the previous section, and can be reduced to 21 floating point operations per component per cell for an anisotropic three-dimensional (3D) model space.

$$\text{Memory required} \propto \times \text{Number of cells} \times 6 \times \text{Constant}$$

The factor 6 in the above expression refers to number of field components, and the constant value includes the bytes used to represent a floating-point number, and 1 byte used to indicate the material type of the cell.

5. GPU Implementation

FDTD update calculations are both data parallel, and arithmetically intense, thus making them a good candidate for execution on a GPU. The same E and H update computations occurs for every cell in the model space satisfying the required data parallelism, and each update is at least 18 FLoating-point Operations per Second (FLOPS), enough to mask memory latency on most GPUs.

The initial GPU FDTD code was implemented in C and Cg for GeForce 7 series based GPUs, which have distinct vertex and fragment processors. E and H fields were stored on device memory as 32 bit floating point red-green-blue (RGB) two-dimensional (2D) textures. The RGB color channels were used to store the X, Y, and Z field components. A half precision luminance texture was stored as a pointer stream to store the material types in the model space. In both of these cases, the 3D volume was flattened into a 2D textures and is accessed via a dot product based 3D to 2D address translation[1]. This allows the entire 3D space to be updated in one render pass, and also avoids potential read after write data corruption. The material type pointer stream was used for dielectric material property lookups stored in textures[1]. E and H scattered field update calculations were converted to fragment programs written in Cg. The update shader programs took the E and H fields stored in textures as inputs and Frame Buffer Objects (FBOs) are used as render target outputs. The computation for each update shader program is initiated by rendering a quad. Between each time step, input memory and output memory on the device is swapped providing a feedback loop and avoiding the performance penalty of pushing data across the system bus[5]. The basic GPU FDTD implementation is as follows:

5. Results

For our performance testing, we benchmarked a variety of GPUs and CPUs. All tests were run with a Linux 2.6 kernel, compiled with GCC 4.1 with "-O3" and architecture specific flags, and Cg 1.5. The CPU and GPU specifications are as listed in Table 1.

We used execution time and overall rate of floating point operations per second as the performance metrics. For our tests, FLOPS are calculated by multiplying the 63 floating point operations per cell per time step by the number of cells in the model and by the number of total time steps, and then dividing by the total execution time of the program including initialization. Steady state performance is achieved when the initialization cost is amortized over many iterations (>1,000) and approaches zero per iteration (as is shown in Table 2).

Table 1. GPU and CPU Specifications

	GPUs			**CPUs**		
	GeForce 8800 GTX	**GeForce 7800 GS**	**GeForce Go 7400**	**Core 2 Duo T7600**	**Opteron 890**	**Opteron 270**
Clock Speed	575 MHz	440 MHz	400 MHz	2.33 GHz	2.8 GHz	2.0 GHz
Processing Elements	128 Stream Processors	22 (6 Vertex, 12 Fragment) Processors	8 (4 Vertex, 4 Fragment) Processors	2 Cores	2 Cores	2 Cores
Memory Bandwidth	86.4 GB/s	38.4 GB/s	7.2 GB/s	3.8 GB/s	6.4 GB/s	6.4 GB/s
Street Price	$550	$170	$65	$650	$1500	$230

Table 2. GPU and CPU FDTD Performance

	GPUs			**CPUs**		
	GeForce 8800 GTX *(0.575 GHz)*	**GeForce 7800 GS** *(0.44 GHz)*	**GeForce Go 7400** *(0.4 GHz)*	**Core 2 Duo T7600** *(2.33 GHz)*	**Opteron 890** *(2.8 GHz)*	**Opteron 270** *(2.0 GHz)*
Steady-state performance (MFLOPs)	33,959.76	5,365.90	980.84	170.95	173.74	79.12
Initialization penalty for 140^3 model(s)	7.41	12.31	1.77	negligible	negligible	negligible
Minimum model size required for maximum performance	100^3	150^3	140^3	10^3	10^3	10^3
Largest possible model size	220^3	150^3	140^3	420^3	1760^3	690^3
Speedup vs. Opteron 270 (times faster)	429.20	67.82	12.40	2.16	2.20	1.00

The initial GPU accelerated FDTD written in C and Cg showed speedup many times faster than the traditional FDTD algorithm executed on the CPU in almost all cases. Generally if the model space was large enough to keep the fragment processors busy, the GPU version was always several times faster, even for the slowest seven series GPU tested, the GeForce Go 7400. For models that are not big enough to saturate the GPUs fragment processors, the GPUs would run slower than on the CPU. Therefore, using the GPU results in a faster execution time, if the FDTD model is large enough; however, due to the efficient scheme of packing a 3D model space into a 2D texture, speedup greater than one was observed at model spaces of only 8^3 given 1,000 iterations or more on the GeForce 8800 GTX. Another interesting observation is that with a small number of iterations, one can clearly see the performance penalty in initializing the streams on a GPU, but as the number of iterations increase, this initialization cost is amortized. Since typical FDTD simulations require tens of thousands of iterations to produce results, our GPU implementation produces significant speedup over the CPU implementation in almost all cases except very small model sizes. Our experimental results, given in Figure 4, shows that the GeForce 8800 GTX is 429 times faster than the base CPU, the Opteron 270, once steady state is achieved, and is faster in almost all cases except for very small models and iterations.

6. Ongoing Research

In the past few months, great strides have been made to allow the GPU to be used as a general purpose streaming processor architecture. Current GPU FDTD efforts are aimed at leveraging these new technologies.

NVIDIA's new GPU architecture debuted in G80 GPUs was a radical departure from previous GPU architectures making it much easier to program for general purpose applications. Gone were the specific vertex and fragment processors in favor of general purposed stream processors that could be tasked for either purpose or even set to a general purpose computation mode. GPU memory was also drastically changed, allowing it to be read in a general way, and new memory was added to allow scatter.

The Compute Unified Device Architecture (CUDA) API allows programming of new G80-based GPUs, to be accomplished without the need of graphics APIs. CUDA was designed as an extension to the C programming language using C style idioms to program the GPU. This eliminates graphics programming in GPU accelerated codes in favor of extensions to standard C thus greatly simplifying the learning curve for non-graphics programmers. CUDA also allows much finer grained control of device memory and how computation is executed on the GPU's stream processors[6].

Currently a production quality parallel GPU-based FDTD code that leverages both the new G80-based GPU architecture and the CUDA API is being completed. These new technologies provide flexibility to implement robust boundary conditions like Perfectly Matching Layers and more complex sources. Greater memory flexibility provides more robust memory management permitting GPU computation overlapped with MPI communication. Extensive experimental evaluations of the new code will be conducted on a GPU cluster.

References

1. Pharr, Matt, Ed., *GPU Gems 2*, Upper Saddle River, Addison-Wesley, 2005.

2. Jackson, John David, *Classical Electrodynamics*, New York:, John Wiley & Sons, Inc., 1999.

3. Kane Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media." *IEEE Transactions on Antennas and Propagation*, vol. AP-14, No. 3, pp. 802–807, May 1966.

4. Kunz, Karl S. and Raymond J. Luebbers, *The Finite Difference Time Domain Method for Electromagnetics*, Boca Raton, CRC Press, 1993.

5. Göddeke, Dominik, "GPGPU--Basic Math Tutorial," November 2005, http://www.mathematik.uni-dortmund.de/~goeddeke/.

6. NVIDIA Corporation Technical Staff, *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, NVIDIA Corporation, 2007.