Finite-Difference Electromagnetics Acceleration on Graphics Processor Units

John R Humphrey, Eric J Kelmelis, Fernando E Ortiz
{humphrey,kelmelis,ortiz}@emphotonics.com
EM Photonics, Inc. Newark, DE 19711, USA

Dennis W Prather
dprather@ee.udel.edu
Department of Electrical and Computer Engineering, University of Delaware, Newark DE 19716, USA

Keywords

Graphics Processor Unit (GPU), Computational Electromagnetics (CEM), Finite-difference Time-Domain Method (FDTD)

Abstract

Our group has employed the use of modern graphics processor units (GPUs) for the acceleration of finite-difference based computational electromagnetics (CEM) codes. In particular, we accelerated the well-known Finite-Difference Time-Domain (FDTD) method, which is commonly used for the analysis of electromagnetic phenomena. This algorithm uses difference-based approximations for Maxwell's Equations to simulate the propagation of electromagnetic fields through space and materials. The method is very general and is applicable to a wide array of problems, but runtimes are long enough that acceleration is highly desired.

Introduction

A recent trend in scientific computing is harnessing the immense power of commodity graphics processing hardware to accelerate numerical algorithms. A current graphics processing unit (GPU) costs roughly the same amount as a high-end CPU, but is capable of significantly higher floating-point performance for many numerical applications. The reason for this disparity is that the GPU needs only to perform specialized calculations, such as those required to render graphics to screen, while the CPU must offer a complete set of functions. The boon for scientific computing is that GPUs are high-powered computation engines, while also providing hardware support for many common linear algebra and trigonometric functions. As this is precisely what many numerical algorithms require, there has been considerable effort lately focused on achieving large performance improvements by adapting codes to these platforms.

To date, many algorithms have been implemented on GPUs. Each expands the toolset of techniques required to map general purpose computing to GPUs. Both high-level algorithms and low-level fundamental mathematical techniques have been demonstrated.

The list of low-level techniques includes the FFT [1], and linear algebra operations on dense [2], banded [3], and sparse [4] matrices. High-level examples are full-fledged algorithms and include computational fluid dynamics [5], ray tracers [6], and tone mapping of images [7].

This paper presents one of one of the high-level class of algorithms, with a complete GPU-based implementation of a popular computational electromagnetics (CEM) method. The Finite-Difference Time-Domain (FDTD) algorithm is commonly used for the analysis of electromagnetic phenomena [8], sound, and heat transfer. For electromagnetics, this method applies the definition of the derivative to the spatial terms of the differential form of Maxwell's Equations at discrete points in space, called "nodes." The temporal derivatives are handled by updating the nodes at discrete time intervals, referred to as "timesteps." The working set of data is composed of electric (E) and magnetic (H) fields, and each timestep includes an update of the E fields followed by an update of the H fields. Each timestep also includes the introduction of the electromagnetic source field, which may be a plane wave, point source, or any of a number of different types. Lastly, an absorbing boundary condition is used in order to give the illusion of infinite space surrounding the region being simulated.

The FDTD method has been formulated in one-, two-, and three-dimensional variations. While 3D is the most flexible in terms of the range of problems to which it can be applied, it significantly increases the time required per simulation. Our group has already approached the 3D problem in other work, utilizing fully-custom hardware on a reconfigurable hardware platform to realize an enormous speed increase [9]. Full 3D implementations of the FDTD method are very resource-intensive in terms of both floating-point operations and memory, and clusters of computers or supercomputers are often required to alleviate the long runtimes. Fortunately, many problems can be cast into forms that can be analyzed using 2D methods, which significantly reduce the computational resources required to produce accurate results. Because of its advantages and applicability to a wide range of problems, we pursued a 2D implementation of the FDTD method on the GPU.

Related Work

The work contained in this paper is an extension of the only other known published implementation of FDTD on the GPU, carried out at the University of Calgary [10]. In their paper, they presented a solver using 8-bit fixed precision arithmetic, and with perfect-electric conductor (PEC) boundary conditions. While they did show a considerable speedup over a standard PC, such a solver has little practical value. Our implementation manages a greater speedup with considerably more features and flexibility. The PEC boundaries are the most troubling feature of the Calgary work, as they produce perfect *reflections* rather than *absorptions* at the edge of the solution space, which is the desired mode of behavior so that the boundaries do not affect the solution. Additionally, the 8-bit fixed-point arithmetic they describe will introduce numerical errors that are unacceptable in the FDTD algorithm.

In the interest of developing a useable tool, we chose to make a fully capable solver based on their original idea. The features we claim unique to our GPU-FDTD implementation are as follows:

- 32-bit floating-point arithmetic
- Perfectly Matched Layer (PML), with PEC boundary conditions as an option
- Isotropic or anisotropic materials
- Support for plane waves and point sources
- Support for a connecting boundary
- Support for nonsquare matrices
- Use of a modern high-level shader language, instead of shader assembly.

Our Implementation

Our implementation of the FDTD method on the GPU was carried out with the following design parameters. The target GPU was a NVIDIA GeForce 5 or 6 series, and our solver was shown to work on both a GeForce 5700 and GeForce 6800 Ultra with 256 MB onboard RAM. The host platform was Windows XP, with a 3.0 GHz Pentium 4 processor. The software framework consisted of OpenGL 1.5.3 with GLUT and GLEW, and the Cg shader language. The NVIDIA driver version was 71.89 on all test machines, and thus Framebuffer Objects were not yet available.

Algorithm implementations for GPU platforms differ from implementations targeted at a microprocessor environment, due to the differences in the architectures of these platforms. For instance, moving data to and from the GPU is an asymmetric process, as downloading data is much more efficient than uploading it. Fortunately, the FDTD algorithm is well-suited for this, as the mesh is downloaded once at the beginning and only uploaded back to the main processor a few times during a run with many thousands of iterations. This paper will focus on the areas that make GPU implementations unique, as they are the most pertinent. We will discuss the following topics: data structures and how they are stored in texture memory; render-to-texture (RTT); vertex and fragment programs; and displaying the resulting data on screen.

Texture memory

The fundamental data type in GPU programs is a matrix stored in texture memory. Normally, this memory is used to store power-of-two (POT) sized arrays, with each element holding a clamped (in the range [0,1]) fixed-point red-green-blue-alpha (RGBA) vector. Recent extensions allow much finer control over this memory, with non-power-of-two (NPOT) textures available that hold many types of data. Extensions also enable what is possibly the most important feature to general-purpose computing on GPUs: the use of 16- and 32-bit floating-point values.

The FDTD method requires storage for field components at each point in the mesh. We chose to implement a non-adaptive, uniformly sampled mesh for simplicity – this results

in a regular 2D array of values, with each entry representing a single point in space. For our 2D transverse electric (TE) formulation, each node requires the storage of four field components: Ezx, Ezy, Hx, and Hy, where Ez=Ezx+Ezy. The separation of the Ez field components allow for implementation of the PML boundary conditions.

The requirement of our mesh to store four field values at each point is a perfect mapping to the GPU. For this, we allocate a 32-bit floating-point 2D texture of size *Height x Width*, and map the RGBA components of the texture to the Ezx, Ezy, Hx, and Hy fields, respectively. This array is the only read/write memory required to perform the FDTD method, and all fields are updated each iteration. A typical texture is read-only, so our data must be handled in a special manner.

Currently, the standard method for creating a writable texture is through the use of "pbuffers", exposed by the extension WGL_ARB_pbuffer. This extension supplies the ability to render to an off-screen target, which, when combined with the WGL_ARB_render_texture extension, enables the creation of read/write textures. One caveat is that a texture cannot reliably be used for reading and writing in one rendering pass.

As the FDTD algorithm continually updates the same data set, feedback is required. There are two methods for handling this in GPUs: copy-to-texture (CTT) and render-to-texture (RTT). CTT mode renders to an off-screen target and then copies the results back to texture memory. This uses a considerable amount of texture memory bandwidth, thus producing sub-optimal results. It should be noted that this is currently the only method to produce texture feedback on a Linux platform, due to a limitation of the NVIDIA drivers. The preferred method is RTT, which allows rendering directly to the texture memory and requires no texture copy to produce feedback.

In order to use a pbuffer with RTT, we employed the RenderTexture class by Mark Harris [11]. We allocate a single RenderTexture to hold all fields, and perform computations in a "ping-pong" fashion. The FDTD method allows for all E fields to be updated simultaneously, as there are no dependences among these fields. The same applies for H fields, which can also all be updated independently of one another. However, E and H fields depend on each other, and thus all E fields must be updated before moving on to update the H fields, and vice versa. To do this on the GPU, we do each timestep in two passes. The first pass updates the E fields and the H fields are passed through the shader without an update. On the second pass, the H fields are updated and the E fields are passed through. The shader programs are detailed in the following sections.

The FDTD algorithm also requires many constant values that describe the propagation of fields between cells. The exact nature and formulation of these constants is dependant on the functionality required, and we will therefore discuss these in a manner that can be used for many different formulations. For our purposes, we used two 32-bit floating-point RGBA textures to describe the field propagation constants, one for use when updating E fields, and one for H field updates. The red and green channels were used to

describe the coefficients for one update equation, and the blue and alpha channels were used for the other equation. Since we have chosen to store the coefficients at each node rather than using a lookup table method, we have a virtually limitless degree to which we can customize the mesh. As these only need to be read, they were stored as conventional read-only textures, albeit using the nonstandard floating-point number format. Both are downloaded once at the beginning of the analysis and are thereafter used as constants.

These two textures were sufficient to implement our chosen boundary condition, "perfectly matched layers" (PML). PMLs function by providing impedance-matched layers that are made of lossy material. The effect is that the energy from the outgoing fields is absorbed without reflections. This method is straightforward to implement, as it requires manipulating the propagation coefficients on the edges of the mesh. Thus, the boundary conditions were transparently handled by the main shaders, without additional or conditional processing, a key advantage of our formulation. Supporting PMLs is the reason that the Ez fields were split into their Ezx and Ezy components.

Texture memory was also instrumental in implementing our source fields, of which two FDTD source types were implemented: point-sources and plane waves. For point sources, a single extra 2D read-only texture was used to add in the source field at the point. The texture was used as a 2D mask, with zeros everywhere except the source point, which was assigned a one. While this may seem a waste of memory, it actually allows a good deal of flexibility and generality. With this technique, many copies of the source can be included with varying amplitudes, and there is also no need to hard-code the location of the source in the shader program.

To implement plane waves, we built directly on the foundation of the point source. Plane waves require the application of both electric and magnetic fields, so a second texture was allocated to hold the magnetic field source coefficients. Additionally, our plane wave source method employs lookups in a 1D table to fetch source values before multiplying by the coefficient which controls how the source is introduced to that particular node. The values in this table are precomputed by the host PC and are downloaded each timestep into the texture memory on the card. The values in the 1D table represent a plane wave propagating through free space, and these are then projected into the 2D mesh via the values in the source coefficient texture.

Fragment Programs

When performing computations on the GPU, most of the work is performed by shader programs. These are executed per-pixel, which the programmer generally forces to correspond to a matrix on a 1:1 basis. In our application, this condition was ensured by drawing a single polygon the exact size of our computation space. In total, three fragment programs were created for this project. The first two are the most important, as they are used for the field update calculations.

FDTD uses central differences instead of evaluating Maxwell's Equations in their direct differential form. However, due to the locations of the fields in the Yee cell, the E fields

behave as more of a backwards difference and the H fields behave as a forward difference [12]. For instance, updating the field Hx(i,j) requires Ez(i,j) and Ez(i,j+1). This and other small algorithmic differences in the two updates made it sensible to have two separate shader programs.

The two shaders follow a very similar flow, beginning with a texture fetch from the appropriate coefficient texture at the current (x,y) coordinates. A second fetch is performed immediately to obtain the field values of the current node. The fields being updated (E or H) are multiplied by their propagation constants using Cg syntax sufficient to ensure SIMD operation.

The shader then performs several more texture fetches, the first two of which are for the fields from the neighbor nodes, used to calculate the central differences. The offset coordinates for these are generated by a vertex program and are passed via the TEXCOORD0 binding semantic, discussed later. Each node requires the fields from a horizontal and vertical neighbor, meaning that there should be many cache hits for these fetches. At this point, the formulations for point sources and plane waves diverge, and for performance reasons are stored in different shader programs. For point sources, a simple lookup is performed into the source coefficient texture at the current coordinates and the resulting value is multiplied by a single floating-point number representing the value of the source at that particular timestep. This value is updated by the CPU once per timestep and is downloaded with negligible performance implications. For plane waves, two lookups are required. The first is the same as point sources, as it is a lookup into the source coefficient texture. This texture contains a value that is used for a lookup into a small 1D table that is downloaded by the CPU once per timestep and a second value that is used as a multiplicative coefficient for the first.

With all of the texture fetches complete, the shaders can perform the actual FDTD field update equations. Part of this has already been completed earlier with the multiplication of the fields to be updated by their propagation coefficients. The remainder of the operation is to add in the contribution from the partial differences and source terms. These operations were all performed with vectorized syntax when possible. The procedure is as follows: required fields from the neighbor nodes are added together and then multiplied by coefficients, and then this and the value of the source field are added to attain the value of the updated field. Finally, the updated fields are packed into an RGBA vector and returned.

A third fragment program was used in order to exercise one of the graphics card's unique abilities, rendering results to screen. As intermediate results are always located on-card, it is a simple matter to occasionally update a box on screen with the current state of the fields as the algorithm progresses. One of the unique advantages of FDTD over other CEM methods is that time-domain calculations allow the viewing of transient fields – other methods can only access results that show the state of the system as if the source had been active for an infinite amount of time. Therefore, we wrote a small shader to convert the intermediate field values to colors in the displayable range, which is [0,1). The shader adds the two E field components of our algorithm (as their splitting is done

for mathematical purposes, and not physical), scales either linearly or logarithmically depending on user input, and shows positive values in the red channel and negative values in the blue, as shown in Figure 1. On other platforms, displaying graphical results to screen can be difficult due to the amount of data involved, but on GPUs it is easy. Because of the nature of graphics processing hardware, displaying the results to screen resulted in less than a 5% performance penalty.

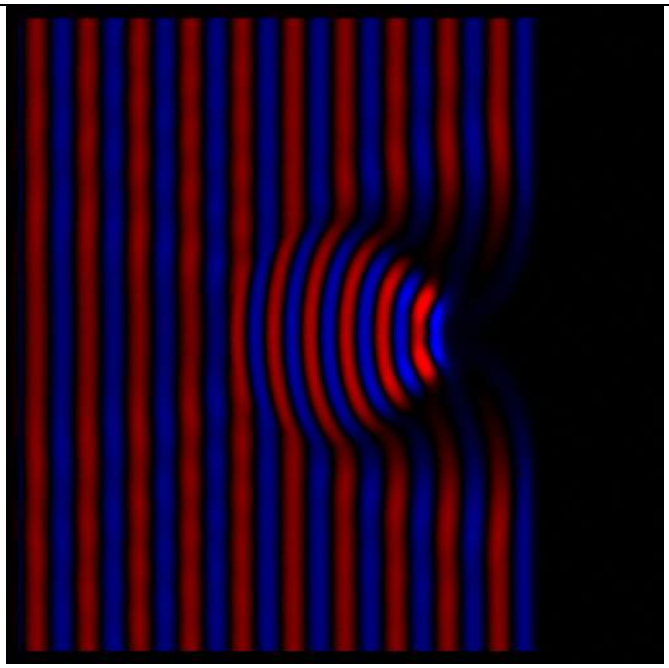

Figure 1: Transient results displayed with a fragment program. This shows the relative strength of the electric fields from a plane wave striking a dielectric cylinder in free space.

The Vertex Program

As the fragment shaders are often the bottleneck in general-purpose computing on GPUs, it makes sense to move as much work as possible to the vertex units. Vertex programs are executed on a per-vertex basis, rather than a per-pixel basis. Results of these programs are passed through the rasterizer where they are linearly interpolated to per-pixel values. This operation can be extremely useful when data is required to vary linearly over a data set. For the purposes of FDTD, this is exactly the behavior required to calculate the locations for nearest-neighbor lookups. Consequently, we offloaded this work from the fragment shaders to the vertex units as originally described in [13], using the TEXCOORD0 binding semantic to pass the results to the fragment shaders. The improvements from this operation were minimal, indicating that our method is memory-bound. At a later date, we hope harness more of the power of the GPU by shifting the bottleneck back towards the shader units. When this occurs, the vertex program described above will be incredibly useful in increasing the overall system performance.

Results

Our preliminary results for this work are very encouraging. On a NVIDIA GeForce 6800 Ultra card, our benchmarking results show a speedup of approximately 5x over a 3.0 GHz Pentium 4 processor. The percentage error is on the order of 10^{-5} compared to a software-only implementation. This amount of error occurs naturally when working with floating-point numbers on different platforms due to compiler differences, rounding modes, and order of operations. As stated earlier, our program is extremely memory bandwidth intensive, and early optimization work points towards a memory bottleneck. We are looking at several possibilities for reducing the bandwidth requirement, though most have tradeoffs that must be considered. An often-used GPU optimization is to use 16-bit floating-point storage instead of 32-bit, which adds an unacceptable amount of error when analyzing large problems. Our best option is to use 8-bit storage for all values except fields, and then use 8-bit values to perform lookups into arrays storing 32-bit coefficients. This is a common FDTD technique, but we will have to study its effects in the GPU environment. We expect this to greatly relieve our memory bottleneck, enabling a speedup greater than our current 5x.

Future Work

While our implementation is fairly capable and fast, there are several avenues of future research we are planning to explore. For enhancing performance even further, we will use several techniques. The new Framebuffer Object extension to appear in coming NVIDIA drivers allows for faster rendering context switches. Additionally, we will switch from storing 32-bit floating-point coefficients to using a lookup table method. While our current method is more flexible, most simulations only require a few dozen materials. The lookup table will allow us to store only an 8-bit index value at each node, rather than four 32-bit numbers. This will significantly ease our memory bottleneck, and should improve performance.

We will also look to add several feature enhancements. The first is off-axis plane waves, which are especially useful when analyzing lenses and gratings. These can be implemented very effectively in GPUs as they require sin/cos operations and linear interpolation, which are all supported natively on GPUs. Second, we will support periodic and other specialized boundary conditions. This change will require some re-work of the underlying algorithm and will most likely use different shaders built around the same concepts we have already demonstrated. Third, we would like to calculate steady-state results without requiring readbacks to the CPU. This can be implemented by allocating extra storage on-board for the required mesh snapshots, and using an extra shader to combine them into the final results. Last, we will support transverse magnetic (TM) mode FDTD, which will be straightforward as it only differs from our current transverse electric (TE) mode by a small amount and is no more computationally complex.

Difficulties in Implementing 3D FDTD

As stated earlier, a 2D FDTD implementation is not capable of analyzing all problems of interest, and a 3D implementation is required. Our group has particular expertise in the acceleration of the 3D method with special-purpose hardware, and we foresee difficulties in performing 3D FDTD on the GPU.

The foremost difficulty is the lack of onboard memory for storing the mesh. Graphics cards with 512 MB onboard memory are only just becoming available as of this writing, and cards with more than that are not expected in the foreseeable future. The 3D FDTD method is notorious for requiring vast amounts of memory, and is often implemented on computer clusters to gain access to extremely large pools of memory. Analyzing larger problems would require constantly swapping data from main memory to the graphics card, resulting in a significant loss of performance. Further analysis must be performed to find an acceptable approach to overcome this limitation if 3D FDTD is ever to be practical on a GPU.

Summary

The work in this paper shows that modern GPUs are capable of large speedups over standard processors for finite-difference applications. Our embodiment was the CEM FDTD method for the analysis of electromagnetic interactions. However, finite-difference formulations have been applied to the analysis of many physical phenomena such as heat and sound. The CEM formulation is significantly more complex than most other methods because it deals with the analysis of transverse waves, where the other methods deal with longitudinal waves. Thus, the calculations of these can be built by *simplifying* the implementation described in this paper. Accordingly, the speedups for those methods should be at least as good as ours, if not better.

In some places, our work was limited by the capabilities of the graphics hardware languages and drivers. We used the FP30 fragment profile for the majority of our computations. The FP40 profile was available, but did not offer any additional desired features. In future releases, the ability to arbitrarily index constant arrays would be extremely useful. These would enable the creation of flexible lookup tables, whereas we now need to use 1D arrays stored in texture memory. This can be inefficient, as it must contend for texture cache and bandwidth. Besides this, the language is reasonably easy to work with and provides a good deal of syntactical power for numerical computing.

References

1. Sumanaweera, T. and D. Liu, *Medical Image Reconstruction with the FFT*, in *GPU Gems 2*, M. Pharr, Editor. 2005, Addison-Wesley: Boston.

2. Fatahalian, K., J. Sugerman, and P. Hanrahan. *Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication*. in *Eurographics/SIGGRAPH Workshop on Graphics Hardware*. 2004.
3. Kruger, J. and R. Westermann, *Linear Algebra Operators for GPU Implementation of Numerical Algorithms.* ACM Transactions on Graphics (TOG), 2003. **22**(3): p. 908--916.
4. Bolz, J., et al., *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid.* ACM Transactions on Graphics (TOG), 2003. **22**(3): p. 917-924.
5. Harris, M.J., *Fast Fluid Dynamics Simulation on the GPU*, in *GPU Gems*, R. Fernando, Editor. 2004, Addison-Wesley: Boston. p. 637-665.
6. Purcell, T.J., *Ray Tracing on a Stream Processor*. March 2004, Stanford University.
7. Goodnight, N., et al. *A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware*. in *Eurographics/SIGGRAPH Workshop on Graphics Hardware*. 2003.
8. Taflove, A., *Advances in Computational Electrodynamics - The Finite Difference Time Domain Method*. 1998, Norwood, MA: Artech House.
9. Durbano, J.P., et al. *FPGA-based Acceleration of the Three-Dimensional Finite-Difference Time-Domain Method for Electromagnetic Calculations*. in *Global Signal Processing Expo & Conference (GSPx)*. 2004.
10. Krakiwsky, S.E., L.E. Turner, and M.M. Okoniewski. *Graphics Processor Unit (GPU) Acceleration of Finite-Difference Time-Domain (FDTD) Algorithm*. in *International Symposium on Circuits and Systems*. May 2004.
11. Harris, M.J., *RenderTexture*. 2004, http://www.markmark.net/misc/rendertexture.html.
12. Yee, K.S., *Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media.* IEEE Transactions Antennas and Propagation, 1966. **AP-14**: p. 302-307.
13. Woolley, C. *Efficient Data Parallel Computing on GPUs*. in *SIGGRAPH 2005 GPGPU Course*. August 2005 (to appear).