Implementation of the Finite Different Time Domain Method on Graphics Processing Units using CUDA

S. David Lively

June 2014

# Abstract

Traditionally, optical circuit design is tested and validated using software which implement some sort of numerical modeling technique such as Beam Propagation, Finite Element Analysis or the finite-difference, time-domain (FDTD[[1]](#footnote-0)) method.

While effective and accurate, FDTD requires significant computational power. Existing installations may distribute the computational requirements across large clusters of high-powered servers. While effective, this approach entails signficant expense in terms of hardware investment, staffing and software support which may be prohibitive for some research facilities and private sector engineering firms. Such institutions may have difficulty with hardware cost, retaining the required hardware and software expertise to administer and maintain such a cluster, or supporting the enormous power requirements with limited financial or personnel resources.

Application of modern programmable GPGPU[[2]](#footnote-1) to problems in scientific visualization and computation has facilitated dramatically accelerated development cycles for a variety of industry segments including large dataset visualization, microprocessor design, aerospace and electromagnetic wave propagation in the context of optical circuit design.

The FDTD algorithm as envisioned by its creators maps well to the massively-multithreaded data-parallel nature of GPUs. This thesis explores a GPU FDTD implementation and details performance gains, limitations of the GPU approach, optimization techniques and potential future enhancements that may provide even greater benefits from this underutilized and poorly-understood area.

# Declaration

The work in this thesis is based on research carried out in the Department of Electrical Engineering, Southern Methodist University, Dallas, Texas, USA. No part of this thesis has been submitted elsewhere for any other degree or qualification and comprises original work unless otherwise explicitly stated within the text.

Copyright © 2014 by S. David Lively

# Table of Contents

[Abstract](#h.yxr4q450h6r4)

[Declaration](#h.z2aq2tlff8iu)

[Table of Contents](#h.ocn2sss3c15x)

[Introduction](#h.vxvfm2g67t7l)

[Background](#h.c69d0n5hvl0x)

[Overview of FDTD](#h.4wdoz6onfg5r)

[Why It’s Important (need a better name for this)](#h.ht5xrt6nbx6d)

[Existing Solutions](#h.myuyxkr0zdpx)

[Meep (Massachusetts Institute of Technology)](#h.g0fylbfsro3g)

[RSoft FullWave](#h.nbdqo9dmwe3c)

[Problems with Existing Solutions](#h.6hjaryrm9bhx)

[Device Architecture](#h.pl8gzqmqhr1z)

[CPU](#h.5n775691jlv4)

[GPU](#h.7h886cyymjz6)

[Summary](#h.gopaxnobw6p4)

[Design Goals](#h.xo10qdm1djfq)

[Implementation](#h.ql03opfhtmu6)

[Results](#h.58vz8ered0ku)

[Test Model Configuration](#h.xjn10qbomlqu)

[Validation Fresnel](#h.5hxnwqvxt2ye)

[Performance](#h.45n5ld8ynbzb)

[Future Work](#h.41kzd4vlq6o2)

[Conclusions](#h.ta42bg8umuhq)

# Introduction

## Background

FDTD requires trivial amounts of work for a given point within a problem domain (“simulation space”). However, any interesting problem will require a large domain, requiring significant processor power and time.

GPGPU programming is a growing field finding adoption in many areas, from the scientific and engineering communities to the financial sector. GPUs provide large arrays of processing cores with somewhat limited processing power.

A fast, GPU-based FDTD tool enables engineers and researchers to iterate designs faster than ever, approaching interactive speeds in some cases. This allows designers to reduce time-to-market for any new product, explore potential research areas that may otherwise have required large computational support such as HPC clusters, and close the iterative design loop.

## 

## Overview of FDTD

*This section needs serious work, but in the interest of getting something down...*

A canonical FDTD implementation works by maintaining parallel arrays describing the electromagnetic state of the problem domain, and updating these fields in an alternating fashion to simulate the propagation of waves. This section describes that process.

The original FDTD algorithm as described by Yee begins by creating an array of so-called “Yee cells.” A cell is a description of the electromagnetic state of a given point in the simulation space, containing electric and magnetic field intensities, as well as material properties of the location in question.

*insert picture of Yee cell here*

As can be seen in this image, each cell maintains electric field values at a given location, as well as the coupled magnetic field values at adjacent “half locations.” In practice, the underlying data structures describe an interlaced field of electric and magnetic fields connected in a mesh, with a magnetic field value placed between each electric field value.

FDTD describes a system where, for a given time step, one set of fields - electric or magnetic - are updated based on the value of their coupled fields at adjacent locations within the simulation space. Once all fields of a given type - electric or magnetic - are updated, the coupled fields are updated given the values calculated in the previous step.

This iterative process may be summarized as follows:

1. Update each E field component in the simulation space using the adjacent H field values
2. Update each H field component using the adjacent E field values calculated in the previous step.
3. Update the current E field to reflect activity of any E or H **sources** within the simulation space. (For example, modulating the amplitude of a point source to simulate a sinusoidal amplitude delta to simulate a monochromatic light source).
4. Repeat all steps until an exit condition is reached.

Individual field components are updated using their previous value and coefficients describing material properties as well as spatial and temporal distance between adjacent grid cells and frames. These are calculated during the initialization phase using the following formulae:

*insert fancy Greek-laden Mathematica screen cap showing calculation of dx and dt based on feature size and source wavelength.*

The concept of a “half step” in time is fundamental to FDTD. In order to effectively simulate the interaction of electric and magnetic fields in a way that accurately portrays the propagation of a wave in space, FDTD requires that E and H fields are updated simultaneously at a given point in simulation time.

However, since each field update depends on the adjacent orthogonal fields, this is not possible. Thus, for a given time t, FDTD updates each E field given the current electromagnetic state of the domain. H fields are then updated assuming that simulated time is now t + 0.5dt, where dt represents a whole time step between E or H frame updates.

Assuming a scale-invariant implementation (ie, all constants such as C, Epsilon0 and Mu0 are set to 1) the value for dx is calculated as 1/10th of the minimum lambda for any source within the domain in order to maintain stability. The temporal step value dt may be set to one half of the scalar value of dx in order to satisfy Courant stability requirements.

*insert nice citation about Courant stability stuff*

By repeatedly updating a matrix of E and H field values taking into account the material properties of the domain (indicating free space or refractive index of any waveguides or other geometry), and inducing changes in the E or H field corresponding to the effect of any sources, the FDTD algorithm convincingly simulates propagation of waves within the simulation domain.

## Why It’s Important *(need a better name for this)*

FDTD and related techniques such as finite element analysis play a crucial role in the design and evaluation of complex physical systems. For example, specification of an optical circuit may require the balancing of a large number of variables.

A recent example is a 3dB waveguide coupler which was designed at SMU using a commercial FDTD package from RSoft .

(coupler schematic image)

As can be seen in the preceding image, the circuit is a cross shape consisting of a substrate, intersecting waveguides and a “mirror” channel at the intersection point. In this case, the some of variables in play were the dimensions and orientation of the mirror channel and waveguides, material properties such as indices of refraction, etc . If the splitter is intended to be integrated into a new design, even the wavelength and amplitude of the incident light source may be flexible.

Waveguide dimensions may range from nanometers to centimeters. Orientation of the mirror channel must be calculated along two independent axis. Light source parameters may range from from near-infrared to ultraviolet or beyond.

A comprehensive analysis of each possible parameter combination for this problem is complex and impractical using analytic methods. Even using high-performance compute clusters, FDTD simulation of a single circuit configuration can be prohibitively expensive in terms of compute time. Simulation of dozens or hundreds of possible combinations may be impossible without the availability of a hyper-scale cluster, utilization of which may dramatically increase the cost of a design project.

In many cases, an engineer may guide this process by eliminating potential designs based on intuition and experience, and combining promising configurations to find an acceptable solution. However, this process can potentially ignore efficient, counter-intuitive solutions. Increases in simulation performance enable analysis of more configurations in less time, increasing the probability of finding near-optimal solutions for a given design problem.

Another technique that has received attention in the field of numerical modeling is the use of genetic, or evolutionary, algorithms to generate and analyze solutions to complex modeling problems. This technique may require hundreds or thousands of iterations to find an acceptable solution. Current modeling tools may require hours or days to simulate a single configuration, in which case such an iterative approach becomes impractical.

A significant reduction in the required time to analyze a single configuration makes this approach a practical tool for numerical analysis of systems with many independent variables, allowing designers to consider a solution domain orders of magnitude larger than was previously possible.

# 

# 

# Existing Solutions

A number of commercial and open-source FDTD simulation packages are currently available and enjoy various levels of adoption within the optics community. This section details two such packages, and provides an overview of the relative strengths and weaknesses of CPU vs GPU-based simulation.

## Meep (Massachusetts Institute of Technology)

Packages such as MIT’s Meep[[3]](#footnote-2) provide a poor user interface. Users are required to provide material definitions in Scheme,[[4]](#footnote-3) an obscure programming language which does not easily facilitate the simulation of complex, arbitrarily-shaped structures. Engineers seeking to utilize Meep must become familiar with Scheme and either adapt their designs to its limitations, or be capable of generating functional definitions of their domain through voxelization[[5]](#footnote-4) or parameterized surface definition.

Additionally, as a CPU-based implementation, Meep cannot take advantage of the underutilized GPU resources that are present in many simulation platforms.

## RSoft FullWave

Another popular solution is FullWave[[6]](#footnote-5) by RSoft, Inc.

While RSoft provides a cleaner user experience than Meep, its performance is similarly inhibited by its reliance on clusters of high-power CPUs. As such, it is not a viable option for engineers working with limited resources *(hardware, financial, etc.)*.

## Problems with Existing Solutions

Popular existing solutions such as those detailed above are strictly CPU-based. Modern compute devices such as GPUs and cell processors are inexpensive and incredibly efficient when applied to parallelizable numerical analysis techniques such as FDTD and finite element analysis.

A modern consumer-grade graphics card may offer upwards of 500 processing cores that are designed to perform similar operations in parallel at high speed. While the design of these devices is ill-suited to general purpose computing in which cores are required to perform dissimilar operations in parallel, they perform well when applied to techniques such as FDTD, which is the primary focus of this paper.

These devices are becoming commonplace in desktop workstations and other hardware that facilities may not be using for numerical modeling. Such systems represent valuable processing cycles that are not currently being leveraged to improve the efficiency and effectiveness of numerical modeling.

# Device Architecture

While GPUs offer vastly increased domain-specific processing capability when compared to CPUs, the two platforms each have their strengths and weaknesses.

## CPU

Modern CPUs typically offer four execution cores. Each core offers an independent L1 and L2 cache and register set. This design allows each core to function as a discrete processor.

This approach presents a number of benefits. A workstation may run multiple unrelated applications simultaneously with little performance loss.

As with GPUs, memory bandwidth and access latency is a critical performance bottleneck. While the dedicated caches for each CPU core alleviate this problem, resource contention between cores requiring use of the L3 cache (and thus high-latency, low-bandwidth host memory) can prove very detrimental to performance.

Todo: insert pretty picture of multicore processor with cache hierarchy

## GPU

GPU architecture varies greatly from that of the CPU. This divergence is a result of the history of GPUs and their primary application: real-time image generation in games and visualization applications.

Unlike the CPU, where each core is effectively a discrete processor, GPU cores are divided into collections sometimes called a “warp”, each of which operates in single-instruction, multiple-data SIMD[[7]](#footnote-6) fashion. This means that, within a warp, each thread must execute in lock-step with its neighbors. This has a number of benefits:

todo: outline SIMD vs MIMD benefits, including memory coalescing, locality and cache performance

A significant disadvantage of this design is that applications must work to ensure that threads within a warp avoid divergence in both memory accesses and execution.

Since the hardware implementation of a GPU core provides different data resources (registers, etc.), but only a single execution unit, per warp, all threads must execute the same instruction at the same time. When any thread in a warp follows a different execution path than its neighbors (as a result of branching control structures within the kernel code), all threads within a warp must execute all possible paths. Each thread then discards any results that are not relevant to itself. This can result in severe performance degradation.

TODO: insert pseudocode describing an if...else block in the context of SIMD execution

An additional concern is data dependence and cache coherence. Given the large number of threads in operation and any given time, and the finite bandwidth between an execution core and global memory (TODO: explain global vs local vs shared memory) cache optimization is critical. Contention for resources such as global memory, due to limited bandwidth per core, and for global memory in general, can effectively cause each core to behave as if it is executing serially rather than in parallel. This can easily cause GPU-based code to run orders of magnitude slower than the traditional CPU version.

Despite these limitations, GPUs offer several advantages over CPUs which, if properly utilized, can provide dramatic performance increases.

- Thread count, thousands vs 4 or 8

- SIMD performance within a warp

- Task-level parallelism - (might be too similar to pipelining to be considered an advantage, but a compare/contrast may be worth including)

## Summary

Ultimately, it may be concluded that GPUs are best applied to processes which repeatedly perform identical (or sufficiently similar) operations on large data sets with little or no interdependence. As such, FDTD is a prime candidate for a GPU-centric implementation.

### 

# Design Goals

Talk about the app structure - host code, visualizer, reporting - more than a flow chart but less than an engineering spec

My simulator is structured as a simple command-line application which performs the following functions:

* Process scripts describing configuration data. For example:
  + source location(s) and characteristics such as wavelength and amplitude
  + exit conditions (number of time steps to be calculated)
  + output file format and location
  + visualizer configuration
* Load and process models defined in the Wavefront OBJ[[8]](#footnote-7) file format, typically exported from a modeling package such as Autodesk 3DSMAX. These models must indicate a triangle-based geometry describing any geometry present within the simulation domain, as well material properties of those items. (In my implementation, this is limited to the refractive index of a given material.)
* Allocate and initialize data structures describing E, H and material state of the simulation domain.
* Iteratively update all sources, E and H fields until the exit conditions are met.
* Optionally update the real-time visualizer, if active
* Generate reports in the forms of images and raw data dumps for further analysis and processing.
* Allocate and manage GPU-based resources (field arrays and compute kernels)

A primary design goal was the ability to selectively enable and disable features such as visualization in order to optimize performance. As such, distinct componnets such as the simulation core and visualization subsystem were created as completely separate software libraries, with a third application coordinating their interaction. This provides several benefits, chief of which is the ability to attach a visualizer to a running simulation at any time in order to determine what progress has been made. The visualizer may then be detached in order to conserve resources that may be used to accelerate the FDTD loop.

# Implementation

Detailed implementation description. Insert lots of source code here. Include sequence diagram, flow chart (if appropriate), process communication, pseudocode. This could be 10-15 pages. Should break into subsections based on application module, etc.

Although I created several implementations using different languages (C#, C++ and Python) and graphics systems (OpenGL and DirectX), the current version is a C++ application using NVIDIA CUDA 6.5 for calculation and OpenGL for visualization via a custom rendering engine.

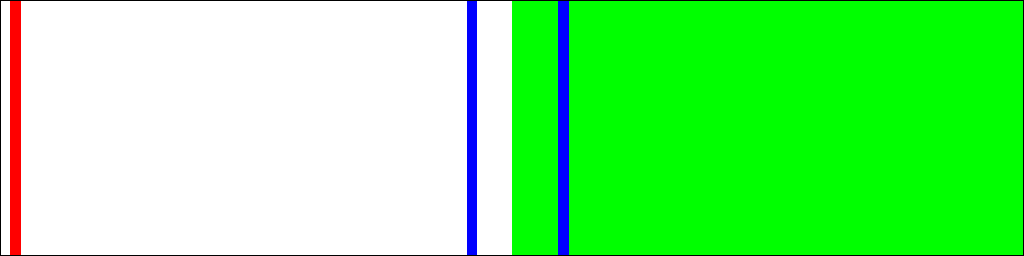
# Results

blah blah blah

## Test Model Configuration

For testing and validation purposes, a simulation comprising a plane wave source, dielectric slab with relative epsilon of 9, and two monitors was used. In the figure below, the plane wave source is represented by the red vertical line, while the incident and transmission monitors are represented by the vertical blue lines.

The green area represents the dielectric slab.



*Figure 1: Test model*

## Testing Methodology

Testing was performed with a TMz simulation. The simulation was first run with a relative epsilon of 1 to record incident magnitude with no reflection.

The simulation was then re-run with a relative epsilon of 9 in the dielectric slab area to record combined incidence and reflection, as well as transmittance within the dielectric.

In a post-processing step, the reflection magnitude was found by subtracting the incident wave magnitude from the first simulation from the combined incidence and reflection magnitudes from the second simulation.

## Fresnel Validation

Validation was performed by comparing the theoretical Fresnel coefficients for the test model with the time-averaged power (RMS) recorded during simulation.

## Performance

0. Describe simulation configuration - geometry, source and monitor placement, plane wave, wavelength, refractive indices.

1. Fresnel Equations - cite equations for P-polarized transmittance and reflectivity

2. Fresnel analytical results - show calculated transmittance and reflectivity for this simulation

2.5 describe approach used to calculate I, by running the simulation twice (with and without the dielectric)

2.75 lots of fancy pictures

3. Show a few fancy pictures showing the domain just before the launch (to visualizer dielectric, monitors and source placement)

4. Show a few other pics showing the evolution of the wave, from just after launch, just before contact with the interface, and just after contact with the transmittance-side monitor

5. Show raw output (the messy wave)

6. Describe RMS-over-time approach

7. Show cleaner graph (the nice one I sent out earlier, not the crazy noisy one)

8. Show calculated T and R coefficients from the data used to generate the graph

9. Show error relative to the calculated coefficients from step #1, which should be basically zero (within the realm of numerical precision and PML loss which,

honestly, should also be insignificant.

10. Summarize.

# Future Work

genetic algorithms, heterogeneous platforms, visualization, interactivity???

# Conclusions

Duh.

1. Yee, Kane S. "Numerical solution of initial boundary value problems involving Maxwell’s equations in isotropic media." *IEEE Trans. Antennas Propag* 14.3 (1966): 302-307. [↑](#footnote-ref-0)
2. Luebke, David et al. "GPGPU: general-purpose computation on graphics hardware." *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* 11 Nov. 2006: 208. [↑](#footnote-ref-1)
3. "Meep - AbInitio." 2005. 18 Jun. 2014 <<http://ab-initio.mit.edu/wiki/index.php/Meep>> [↑](#footnote-ref-2)
4. "Meep Tutorial - AbInitio." 2005. 18 Jun. 2014 <<http://ab-initio.mit.edu/wiki/index.php/Meep_Tutorial>> [↑](#footnote-ref-3)
5. Wang, Sidney W, and Arie E Kaufman. "Volume sampled voxelization of geometric primitives." *Visualization, 1993. Visualization'93, Proceedings., IEEE Conference on* 25 Oct. 1993: 78-84. [↑](#footnote-ref-4)
6. "FullWAVE™ - Synopsys Optical Solutions." 2013. 18 Jun. 2014 <<http://optics.synopsys.com/rsoft/rsoft-passive-device-fullwave.html>> [↑](#footnote-ref-5)
7. Maresca, Massimo, and Hungwen Li. "Connection autonomy in SIMD computers: a VLSI implementation." *Journal of Parallel and Distributed Computing* 7.2 (1989): 302-320. [↑](#footnote-ref-6)
8. Wavefront OBJ is an industry standard, text-based file format with provisions for describing triangle- and quad-based geometry as well as extended properties such as refractive index. [↑](#footnote-ref-7)