

GOLIGHTLY:
A GPU IMPLEMENTATION OF THE
FINITE-DIFFERENCE TIME-DOMAIN METHOD

Approved by:

Dr. Marc Christensen

Professor Ira Greenberg

Dr. Nathan Huntoon

GOLIGHTLY:
A GPU IMPLEMENTATION OF THE
FINITE-DIFFERENCE TIME-DOMAIN METHOD

A Thesis Presented to the Graduate Faculty of the

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Master of Science

with a

Major in Electrical Engineering

by

S. David Lively

(B.S.E.E, Southern Methodist University, 2008)

December 1, 2018

ACKNOWLEDGMENTS

I thank my committee for their patience, insight and unfailing encouragement. Without them, this thesis would remain vaporware. Never give up, never surrender!

Lively , S. David

B.S.E.E, Southern Methodist University, 2008

GoLightly:

A GPU Implementation of the
Finite-Difference Time-Domain Method

Advisor: Professor Marc Christensen

Master of Science degree conferred December 1, 2018

Thesis completed December 1, 2018

Traditionally, optical circuit design is tested and validated using software which implement numerical modeling techniques such as Beam Propagation, Finite Element Analysis and the Finite-Difference Time-Domain (FDTD) method.

While effective and accurate, FDTD simulations require significant computational power. Existing installations may distribute the computational requirements across large clusters of high-powered servers. This approach entails significant expense in terms of hardware, staffing and software support which may be prohibitive for some research facilities and private-sector engineering firms.

The application of modern programmable GPUs to problems in scientific visualization and computation has facilitated dramatically accelerated development cycles for a variety of industry segments including large dataset visualization[21], aerospace[18] and electromagnetic wave propagation in the context of optical circuit design. GPU-based supercomputers such as National Labs' Summit[13], co-designed by NVIDIA and IBM, provide dramatically increased compute capability while using less power and fewer computers.

The FDTD algorithm 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.

TABLE OF CONTENTS

LIST OF FIGURES	vii
LIST OF TABLES	viii
CHAPTER	
1. Introduction	1
2. Device Architecture	3
2.1. CPU	3
2.2. GPU	3
2.2.1. SIMD	3
2.3. Resource Limitations	4
3. FDTD	5
3.1. Wave equation	5
3.2. Yee Cell	5
3.3. Leap Frog: Stepping in Space and Time	8
3.4. Boundary Conditions	9
3.5. FDTD in SIMD	9
4. Meep	11
4.1. Modeling	11
4.2. Performance	11
4.3. Popularity	12
5. GoLightly	13
5.1. Goals	13
5.2. Architecture	14

5.3.	Model Processor	14
5.4.	Simulator	16
5.5.	Visualizer	23
5.6.	Modeling approach	28
5.7.	Testing and Validation Methodology	30
5.7.1.	Analytical Result	30
5.7.2.	Numerical Result	32
6.	Results.....	38
6.1.	Test Environment	38
6.2.	Performance Metrics	38
6.3.	Optimization and Enhancements	42
7.	Conclusions	44
7.1.	Usability	44
7.2.	Future Work	44
7.2.1.	GoLightly Improvements	44
7.2.2.	Genetic Algorithms	45
7.2.3.	Arbitrary Domain Shape and PML Sinks	45
7.2.4.	Load Balancing	46
7.3.	Final Words.....	46
APPENDIX		
A.	APPENDIX.....	47
A.1.	Meep scripts	47
REFERENCES		49

LIST OF FIGURES

Figure	Page
3.1 2D TM_Z Yee Cell	6
3.2 4x4 Yee Lattice	8
5.1 Visualizer Update Pipeline	24
5.2 2D Whispering Gallery Mode Sensor	27
5.3 Arbitrarily-shaped source (Red pixels)	29
5.4 Arbitrarily-shaped source after 20 frames	29
5.5 Arbitrarily-shaped source after 100 frames	29
5.6 TM_Z Test Model.....	30
5.7 Snell's Law	31
5.8 Plane wave with $\epsilon_R = 1$	33
5.9 Time-averaged (RMS) output in free space $\epsilon_R = 1$	33
5.10 Steady state with $\epsilon_R = 9$	34
5.11 Raw monitor output with $\epsilon_R = 9$	34
5.12 RMS output with $\epsilon_R = 9$	35
5.13 Normalized output with $\epsilon_R = 9$	36
6.1 GoLightly: seconds for 5000 frames with the given domain size	39
6.2 GoLightly: Completed E and H updates per second	40
6.3 GoLightly vs Meep	41
6.4 Speedup - Meep Time / GoLightly Time.....	42

LIST OF TABLES

Table	Page
3.1 TM_Z FDTD Equation Terms	7
5.1 Model processor inputs	14
5.2 Color component usage.....	28

*To Audrey, Wyatt, Walter and Gwendolyn,
and
Morgan and Kelley*

Chapter 1

Introduction

The Finite Difference Time Domain (FDTD)[22] method is an electromagnetic wave simulation algorithm employed by many commercial simulation and design packages, as well as open source software such as MIT's Meep[14].

Given the computationally-intensive nature of FDTD, organizations requiring simulation of large domains or complex circuits must provide significant resources. These may take the form of leased server time or utilization of an on-site high-performance cluster, amongst other options.

In this thesis, we explore an implementation of FDTD utilizing graphics processing units (GPUs) via NVIDIA's CUDA[12] language. Initially designed to perform image generation tasks such as those required by games, cinema and related fields, modern versions are well-suited for general computation work. GPUs are now enjoying wide adoption in fields such as machine learning[15] and artificial intelligence[20], medical research[17] and other areas which require rapid analysis of large datasets.

Multi-core CPUs excel at quickly performing disparate operations on potentially-unrelated data. This is a requirement in traditional desktop computing. However, the flexible architecture that provides this capability can be a liability when repeatedly executing large batches of identical operations.

GPUs trade this flexibility for increased throughput. Modern consumer-grade GPUs offer thousands or tens of thousands of processing units ("cores"). Some algorithms, such as FDTD, require little or no data interdependence, no branching logic (a severe performance impediment on GPUs) and consist of short cycles of simple

operations. The power of the GPU lies in performing these simple operations at large scale, with thousands of threads running in parallel.

The following sections detail the FDTD algorithm. Later sections describe a CPU-based implementation (MIT’s Meep), and our GPU-based GoLightly simulator. We verify the GPU solution numerically, and compare performance between CPU- and GPU-based implementations. Finally, we consider future applications and enhancements.

Chapter 2

Device Architecture

CPUs and GPUs each offer advantages for different computational tasks. Multicore CPUs offer independent cores which are effectively discrete processors. GPUs offer larger-scale parallelization, but require strong data and code coherence in order to achieve acceptable performance.

2.1. CPU

Users typically run many different applications in parallel: a web browser, music player, word processor and email client are a common combination. Multicore CPUs facilitate this by providing separate ALUs, register files and cache, minimizing dependencies between cores on the same physical device.

2.2. GPU

Unlike CPUs, GPUs are capable of running thousands of threads in parallel using a SIMD (single-instruction, multiple-data) model. This architecture allows rapid processing of large datasets wherein each datum exhibits little or no interdependence. This independence is crucial to GPU performance as detailed in subsection 2.2.1.

2.2.1. SIMD

In a SIMD[8] architecture, a core may consist of a single ALU with multiple register files. Separate threads load different data into dedicated register banks. The ALU executes identitical operations on each lane's registers simultaneously. In this

way, each register bank is analogous to a CPU "thread." However, since the same instruction must be executed on all register sets at every step, this design is less flexible than a system where each core is truly independent.

That said, the SIMD approach provides some benefits, including:

- Fewer components are required per core since fewer ALUs are required. Leads to reduced die space requirements.
- Better code caching. A single ALU, and corresponding cache, are used for many threads. This simplifies cache controller architecture.

2.3. Resource Limitations

When comparing CPUs and GPUs, it is important to consider available resources. Modern GPUs may provide as much as 16GB (or more) of onboard RAM. However, if a simulation requires more RAM than is available on the device, the simulator will be responsible for swapping data between the GPU and the host system over the PCIe bus. Due to the relatively slow speed of the PCIe bus versus the speed of onboard RAM, this incurs a severe performance penalty which can negate the speedup provided by the GPU.

In contrast, CPUs potentially have much more memory available. A modern consumer desktop may have 32GB or 64GB of host RAM. Operating systems such as Microsoft Windows, Linus and Apple MacOS offer the ability to swap RAM contents to disk when an application's requirements exceed available RAM. While this incurs its own performance penalty, it makes it possible to simulate very large domains that would not fit on a GPU.

Chapter 3

FDTD

FDTD expresses Maxwell's equations as a set of discretized time-domain equations[22]. These equations describe each electric field component in terms of its orthogonal, coupled magnetic fields, and each magnetic field component as a function of its coupled electric fields.

3.1. Wave equation

The TM wave equations for E_z , H_x , and H_y are of the form:

$$\frac{\partial E_z}{\partial t} = K * \left(\frac{\partial H_x}{\partial y} + \frac{\partial H_y}{\partial x} \right) \quad (3.1)$$

$$\frac{\partial H_x}{\partial t} = K * \left(\frac{\partial E_z}{\partial y} \right) \quad (3.2)$$

$$\frac{\partial H_y}{\partial t} = K * \left(\frac{\partial E_z}{\partial x} \right) \quad (3.3)$$

These equations state that the temporal derivative of a field is a function of the sum of the spatial derivatives of the coupled orthogonal fields.

In order to apply these equations to a computational domain, FDTD defines discretization strategies for simulated time and space. The simulation domain is divided into cells, and each frame is updated using a fixed time step derived from parameters such as source wavelength and simulation dimensionality.

3.2. Yee Cell

Yee [22] defines a computational unit known as a "cell." The cell describes how each field component within a domain is related to its coupled fields. For instance, each E_z field component depends on adjacent H_y and H_x components. The cell format in a TM_Z simulation is of the form shown in Figure 3.1.

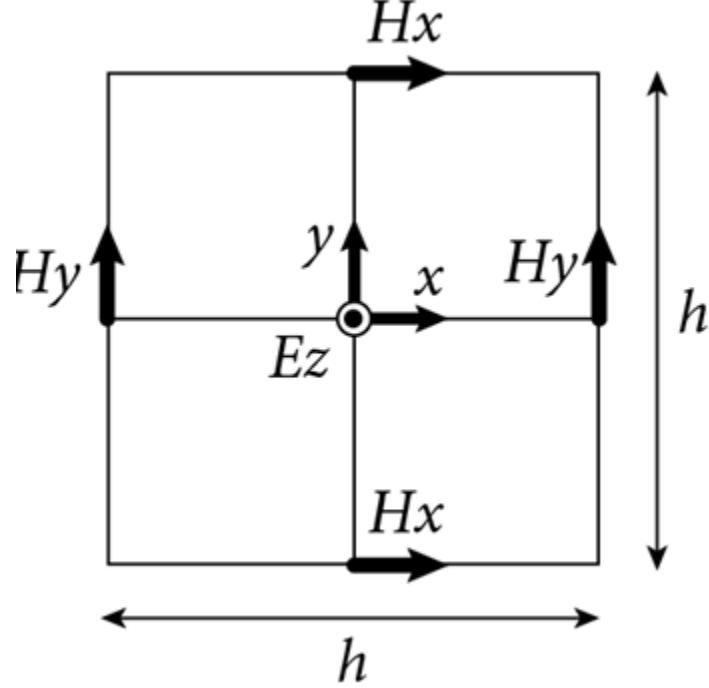


Figure 3.1. 2D TM_Z Yee Cell

More formally, we may expand the E_z wave equation, arriving at:

$$E_{z,i,j}^{t+1} = C_a * E_{z,i,j}^t + C_b * (H_{x,i,j+\frac{1}{2}}^{t+\frac{1}{2}} - H_{x,i,j-\frac{1}{2}}^{t+\frac{1}{2}}) + C_b * (H_{y,i+\frac{1}{2},j}^{t+\frac{1}{2}} - H_{y,i-\frac{1}{2},j}^{t+\frac{1}{2}}) \quad (3.4)$$

Similarly, the 2D equations for the coupled fields H_x and H_y may be expressed as:

$$H_{x,i,j}^{t+1} = D_a * H_{x,i,j}^t + D_b * (E_{z,i,j+\frac{1}{2}}^{t+\frac{1}{2}} - E_{z,i,j-\frac{1}{2}}^{t+\frac{1}{2}}) \quad (3.5)$$

$$H_{y,i,j}^{t+1} = D_a * H_{y,i,j}^t + D_b * (E_{z,i+\frac{1}{2},j}^{t+\frac{1}{2}} - E_{z,i-\frac{1}{2},j}^{t+\frac{1}{2}}) \quad (3.6)$$

Table 3.1. TM_Z FDTD Equation Terms

Symbol	Definition	Description
dx	$\frac{\lambda}{10}$	Spatial step as a function of max source fundamental frequency
dt	$\frac{dx}{\sqrt{n}} * 0.9$	Time step between frame updates, where n is the domain dimensionality
C_a	$\frac{1}{\epsilon_0}$	Permittivity of free space
C_b	$\frac{dt}{dx} \frac{1}{\epsilon}$	Permittivity at the location i, j
D_a	$\frac{1}{\mu_0}$	Permeability of free space
D_b	$\frac{dt}{dx} \frac{1}{\mu}$	Permeability at the location i, j
i, j		Field element location within the domain
t		Current time step ($t_E = t_H \pm \frac{1}{2}$)
E_Z		Electric field amplitude in Z
H_X		Magnetic field amplitude in X
H_Y		Magnetic field amplitude in Y

Since Maxwell's equations are scale-invariant, GoLightly substitutes 1 for constants such as C , μ_0 and ϵ_0 . The $\frac{0.9}{\sqrt{2}}$ scalar in dt prevents aliasing, and corrects for simulation dimensionality¹

In equations 3.4, 3.5 and 3.6, note that each $E_{i,j}^{t+1}$ field update depends only upon the previous E value ($E_{i,j}^t$), and the previous adjacent H values ($H_X^{t+\frac{1}{2}}$ and $H_Y^{t+\frac{1}{2}}$). This independence allows each field component to be updated without regard for any other value in the same field.

¹This value should be $\frac{0.9}{\sqrt{n}}$, where n is 3 for a 3D domain, 2 for a 2D domain and 1 for a 1D domain.

3.3. Leap Frog: Stepping in Space and Time

In equations 3.4, 3.5 and 3.6, note the presence of a "half step" in time ($F^{t+\frac{1}{2}}$) and space $F_{i-\frac{1}{2},j-\frac{1}{2}}$.

This $t_{-}^{\pm\frac{1}{2}}$ represents the temporal step size between an E -field update and the next H -field update, and visa-versa. Similarly, the spatial offset $x_{-}^{\pm\frac{1}{2}}$ represents the distance between an E -field component and its adjacent, interdependent H values.

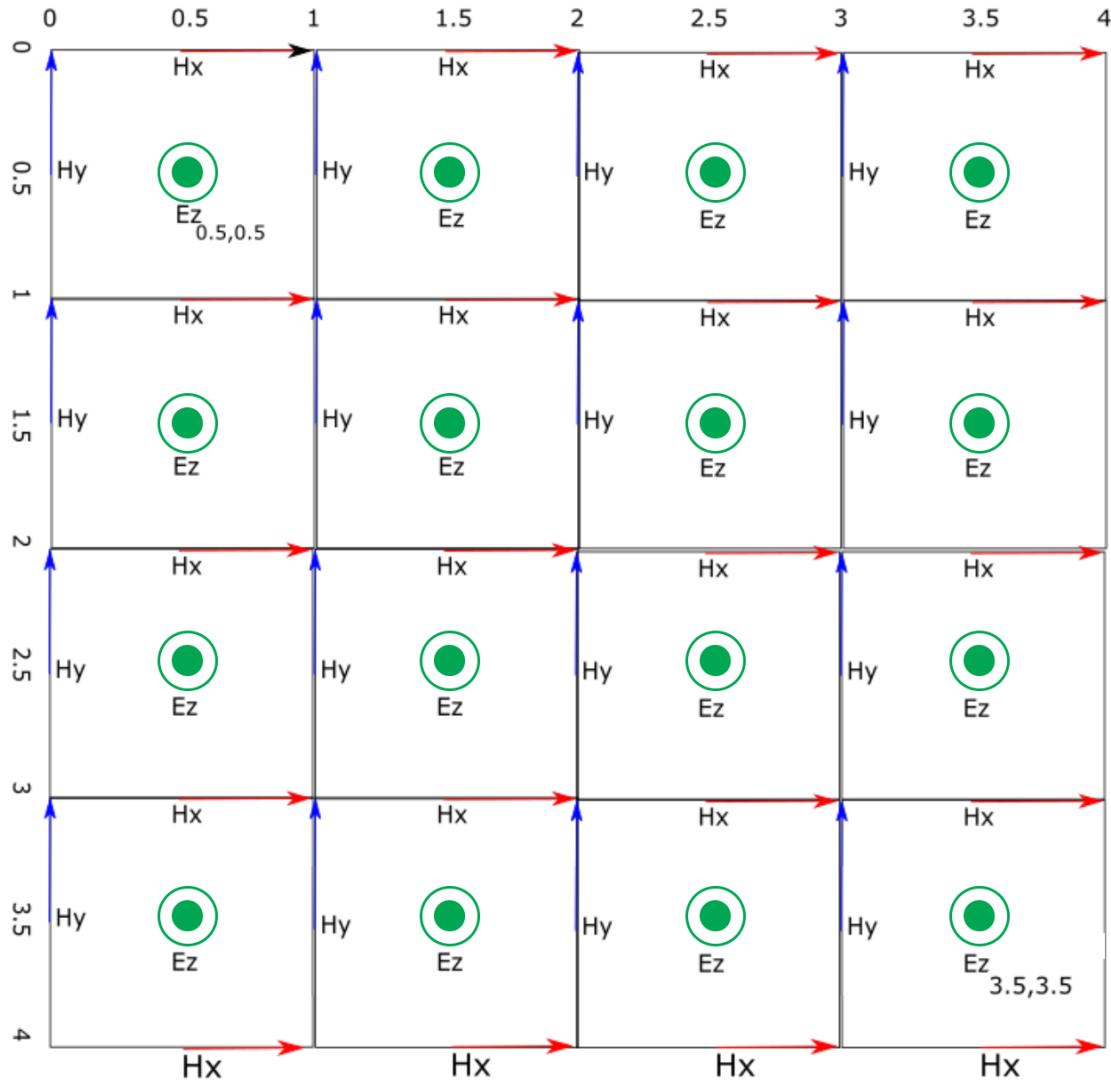


Figure 3.2. 4x4 Yee Lattice

The spatial relationship between E and H grids is illustrated in the Yee lattice in Figure 3.2. Note that each E_Z component is in the middle of a cell, at the $(i + \frac{1}{2}, j + \frac{1}{2})$ position where (i, j) is the upper-left corner of the cell. H components, however, are positioned at integer coordinates.

This arrangement reflects the manner in which a wave will propagate through the domain. In the half time step between E and H updates, the wave moves one half-cell. If the E and H components were coincident, the simulation would degrade to a large collection of individual, disjoint points rather than a discretized, connected domain.

3.4. Boundary Conditions

Recall the H_Y update equation Equation 3.6, and note that it depends on two adjacent E_Z values on the X axis. The finite grid does not contain enough information to update the H values that lie on the edge of the domain. For instance, $H_{Y0,\frac{1}{2}}$ requires $E_{Z\frac{1}{2},\frac{1}{2}}$ and $E_{Z-\frac{1}{2},\frac{1}{2}}$.

Since the position $(-\frac{1}{2}, \frac{1}{2})$ does not exist, the simulator cannot update this value. The simulator must take this case into account by implementing a boundary condition, or the wave will reflect from the boundaries.

We implement the Perfectly Matched Layer (PML) as described in [1]. A detailed explanation of PML is beyond the scope of this thesis. In practice, PML adds imaginary hyperplanes orthogonal to each field in the simulation. In the boundary regions, power couples between E and H as expected, but *also* couples into those hyperplanes. Unlike the E and H interdependence, the transfer is one-way. Dielectric values in the cells in the PML describe non-physically realizable materials which force the coupled power to decay over a number of layers. In our implementation, we use 10 PML layers as experimentation showed this to provide the most satisfactory results.

3.5. FDTD in SIMD

FDTD's leap-frog update method, whereby E fields and H fields are successively calculated, is well-suited to a GPU implementation. E field values depend on adjacent H field values, and visa-versa. Since the E-field update equation requires knowledge only of the H field state and previous E field state, each field component can be calculated independently with no opportunity for a pipeline stall or race condition.

Chapter 4

Meep

Meep[14] is a full-featured, open-source electromagnetic simulator produced by the Massachusetts Institute of Technology. In addition to its core FDTD-based simulation engine, it provides a scripting interface for defining models and simulation parameters, recording results, and other tasks.

4.1. Modeling

One limitation of Meep is it's CSG¹-based modeling language. Construction of arbitrarily-shaped or dynamic structures using CSG is a complex process. It is worth noting that Meep provides a "material function" capability. This allows the user to dynamically determine the material properties of any point in space using their own algorithm rather than defining their model using CSG. However, to take advantage of this capability, the user must employ additional software or custom programming.

4.2. Performance

Meep is a mature, highly-optimized suite of tools. It scales well across multiple machines, relying on the MPI protocol to synchronize nodes within the cluster.

While performant when compared to other FDTD software, Meep suffers from the same architecture-imposed limitations of all CPU-based implementations. The limited number of processing cores available on a general-purpose CPU restricts the

¹Constructive Solid Geometry. A method of describing manifolds as a series of boolean operations of convex polyhedra.

number of data points that can be processed within a given time frame. This problem can be solved by provisioning additional computers which would run in parallel, distributing the computational load across the resulting cluster.

This sort of cluster configuration incurs its own overhead. Although a domain may be divided into discrete parts and distributed across cluster nodes, the state of cells located at the interfaces between subdomains must be communicated between nodes in order to maintain continuity. This exchange must occur for every calculation frame and may necessitate use of a high-speed local network and supporting hardware.

4.3. Popularity

Meep has been widely adopted by many institutions, and is frequently cited in journals such as Nature[19][5], Computer Physics Communication[7], Physical Review Letters[6] and others. In fact, a web search revealed over 1200 citations of the original[14] paper. This speaks to its reliability and accuracy, and indicates that it is a trusted, useful tool.

Chapter 5

GoLightly

GoLightly is the GPU-based FDTD simulator application that is the focus and product of this thesis. Written using a combination of C++, CUDA and OpenGL, it provides a lightweight yet complete FDTD solution.

5.1. Goals

GoLightly is intended to address deficiencies common to CPU-based solutions. In particular, it is designed to be fast, friendly and portable.

- **Fast.** An iterative design process requires rapid feedback from the simulator. Long simulation times necessitated by existing solutions inhibit this process.
- **Friendly.** Definition of models and other simulation parameters should not require expertise in software development or quasi-proprietary scripting languages.
- **Portable.** Ideally, the simulator should run on a high-end consumer grade laptop and support the most common desktop operating systems (Microsoft Windows and Apple OS X).

To meet those goals, GoLightly takes advantage of the programmable GPU available in common desktop and laptop computers, resulting in a dramatic speedup. Rather than relying on a proprietary model definition language or obscure, limited scripting system, we use industry-standard image and geometry file formats so that models may be defined using robust, familiar, readily-available tools.

5.2. Architecture

GoLightly comprises three primary application blocks:

- Model Processor 5.3
- Simulator 5.4
- Visualizer 5.5

5.3. Model Processor

The model processor (MP) is responsible for initialization of the simulator. When launching the simulator, a domain size and image file containing a coded image of the desired dielectric as well as ϵ_{max} are specified.

Table 5.1. Model processor inputs

Symbol	Data Type	Meaning	Typical value
Width	int	Domain size in X	1024
Height	int	Domain size in Y	1024
Media	float	ϵ_{max}	9
Model	string	Model definition stored as a bitmap	filename
ϵ_{max}	float	Maximum ϵ value for this model	

The MP allocates arrays to hold the dielectric properties for each Yee cell. These arrays are of the same dimensions as the domain, which may be different than the dimensions of the model to accomodate boundary cells.

Once the model image is loaded, the MP iterates through each pixel in the image. (See 5.3), and parses each pixel to populate data structures describing the dielectric contents of the simulation as well as source and monitor locations and parameters.

For each element:

1. Determine the normalized texel coordinate that corresponds to the current cell position
2. Read the red (R), green (G) and blue (B) color components from the image
3. If $R > 128$, this texel is part of a source. Add the cell to the list of sources
4. If $G > 0$, this texel has non-unity dielectric. Set $C_{bi,j} = \epsilon_{max} * \frac{G}{255.0}$
5. If $B > 0$, this texel is part of a monitor. Add its position to the monitor definition with $ID = B$

Source code for the image processor is shown in listing 5.3.

```

1 for( int j = 0; j < media.Size.y; j++)
2 {
3     int sourceY = j * height / media.Size.y;
4     for( int i = 0; i < media.Size.x; i++)
5     {
6         int sourceX = i * width / media.Size.x;
7         unsigned int sourceOffset = channels * (sourceY * width + sourceX);
8         unsigned int mediaOffset = j * media.Size.x + i;
9         unsigned char red = bytes[sourceOffset + 0];
10        unsigned char green = bytes[sourceOffset + 1];
11        unsigned char blue = bytes[sourceOffset + 2];
12        // is this pixel part of a source?
13        if (red > 128)
14            sourceOffsets.push_back(mediaOffset);
15
16        // fill default waveguide material (parameter n)
17        if (green > 0)
18        {
19            // interpolate n based on green value.
20            media.HostArray[mediaOffset] = epsilonMax * green * 1.f / 255;

```

```
21     }
22 }
23 }
```

Listing 5.1. Generating a model from an image

Once the dielectric, sources and monitors are derived from the model image, the model processor transfers control to the simulator.

5.4. Simulator

The simulator block implements the FDTD algorithm. Given the dielectric, source and monitor configurations from the model processor, the simulator initializes the GPU, transfers required data from host memory to the GPU, and begins the simulation loop.

In addition to the dielectric and field arrays, the simulator generates a descriptor (5.4) for each field that will be updated. This structure is used by the kernels to assist in handling boundary conditions (PML) and other housekeeping duties. A similar, more compact descriptor (5.4) is generated from the host descriptor and passed to the kernels.

```
1 struct FieldDescriptor
2 {
3     /// <summary>
4     /// describes a split-field boundary region for PML
5     /// </summary>
6     struct BoundaryDescriptor
7     {
8         FieldType Name;
9         FieldDirection Direction;
10    }
```

```

11 // CPU-resident fields
12 float *Amp;
13 float *Psi;
14 float *Decay;
15
16 BoundaryDescriptor *DeviceDescriptor;
17
18 unsigned int AmpDecayLength;
19
20 private:
21 CudaHelper *m_cuda;
22 };
23
24
25 float DefaultValue;
26 FieldType Name;
27
28 dim3 Size;
29 dim3 UpdateRangeStart;
30 dim3 UpdateRangeEnd;
31
32 vector<float> HostArray;
33 float *DeviceArray;
34
35 DeviceFieldDescriptor *DeviceDescriptor;
36
37 vector<GridBlock> GridBlocks;
38 map<FieldType, shared_ptr<BoundaryDescriptor>> Boundaries;
39 }

```

Listing 5.2. Host Field Descriptor structure

```

1 enum class FieldDirection { X,Y,Z };
2
3 struct DeviceFieldDescriptor
4 {
5     FieldType Name;
6     dim3 Size ;
7     dim3 UpdateRangeStart ;
8     dim3 UpdateRangeEnd ;
9
10    float *Data ;
11 };

```

Listing 5.3. Device Field Descriptor

For each loop iteration, the simulator launches a CUDA kernel to update all E fields. Once the E update is complete, the simulator launches kernels to update all H fields.

The three kernels required for a TM_Z simulation are detailed below:

```

1 __global__ void UpdateEz(
2     dim3 threadOffset
3 )
4 {
5     unsigned int x = threadOffset.x + blockIdx.x * blockDim.x + threadIdx.x;
6     unsigned int y = threadOffset.y + blockIdx.y * blockDim.y + threadIdx.y;
7
8     if (y < 1 || x < 1)
9         return;
10
11    float cb = Cb->Data[y * Cb->Size.x + x];

```

```

12
13     unsigned int center = y * Ez->Size.x + x;
14     float hxBottom = Hx->Data[y * Hx->Size.x + x];
15     float hxDTop = Hx->Data[(y - 1) * Hx->Size.x + x];
16     float dhx = (hxBottom - hxDTop);
17
18     float hyRight = Hy->Data[y * Hy->Size.x + x];
19     float hyLeft = Hy->Data[y * Hy->Size.x + x - 1];
20     float dhy = (hyLeft - hyRight);
21
22     float ezxPsi = 0.f;
23     float ezyPsi = 0.f;
24
25 // PML
26 if (x < 10 || x > Ez->UpdateRangeEnd.x - 10 || y < 10 || y > Ez->
27     UpdateRangeEnd.y - 10)
28 {
29     ezyPsi = Ezy->Decay[y] * Ezy->Psi[center] + Ezy->Amp[y] * dhx;
30     Ezy->Psi[center] = ezyPsi;
31     ezxPsi = Ezx->Decay[x] * Ezx->Psi[center] + Ezx->Amp[x] * dhy;
32     Ezx->Psi[center] = ezxPsi;
33 }
34
35 Ez->Data[center] = CA * Ez->Data[center] + cb * (dhy - dhx) + cb * (
36     ezxPsi - ezyPsi);
}

```

Listing 5.4. CUDA kernel for updating E_Z

The majority of each kernel's source performs setup and bounds checking tasks. In each kernel, the FDTD equation implementation can be isolated to one or two

lines of code.

For example, the line (from the E_Z update kernel),

```
1 Ez->Data[ center ] = CA * Ez->Data[ center ] + cb * ( dhy - dhx ) + cb * (  
2   ezxPsi - ezyPsi );
```

corresponds to the FDTD E_Z equation. (See Equation 3.4)

```
1 __global__ void UpdateHx(dim3 threadOffset)  
2 {  
3     unsigned int x = threadOffset.x + blockIdx.x * blockDim.x + threadIdx.  
4         x;  
5     unsigned int y = threadOffset.y + blockIdx.y * blockDim.y + threadIdx.  
6         y;  
7  
8     if (y >= Ez->Size.y - 1)  
9         return;  
10  
11    unsigned int hxOffset = y * Hx->Size.x + x;  
12 #ifdef USE_MAGNETIC_MATERIALS  
13    float db = Db->Data[y * Db->Size.x + x];  
14 #else  
15    const float db = DbDefault;  
16 #endif  
17  
18    float dEz = Ez->Data[(y + 1) * Ez->Size.x + x] - Ez->Data[y * Ez->Size  
19         .x + x];  
20  
21    float hx = DA * Hx->Data[ hxOffset ] - db * dEz;  
22  
23    if (y < 10 || y > Hx->UpdateRangeEnd.y - 10 || x < 10 || x > Hx->
```

```

UpdateRangeEnd.x - 10)

23 {
24     /// update boundaries
25     float decay = Hxy->Decay[y];
26     float amp = Hxy->Amp[y];
27     float psi = Hxy->Psi[hxOffset];
28
29     psi = decay * psi + amp * dEz / Configuration->Dx;
30
31     Hxy->Psi[hxOffset] = psi;
32     hx = hx - db * Configuration->Dx * psi;
33 }
34
35 Hx->Data[hxOffset] = hx;
36 }
37
38 -

```

Listing 5.5. CUDA kernel for updating H_X

```

1 --global-- void UpdateHy(dim3 threadOffset)
2 {
3     unsigned int x = threadOffset.x + blockIdx.x * blockDim.x + threadIdx.x;
4     unsigned int y = threadOffset.y + blockIdx.y * blockDim.y + threadIdx.y;
5
6     if (x >= Ez->Size.x - 1)
7         return;
8
9     unsigned int hyOffset = y * Hy->Size.x + x;
10

```

```

11 #ifdef USE_MAGNETIC_MATERIALS
12     float db = Db->Data[y * Db->Size.x + x];
13 #else
14     const float db = DbDefault;
15 #endif
16
17     float ezLeft = Ez->Data[y * Ez->Size.x + x];
18     float ezRight = Ez->Data[y * Ez->Size.x + x + 1];
19
20     float dEz = ezRight - ezLeft;
21     float hy = DA * Hy->Data[hyOffset] - db * (ezRight - ezLeft);
22
23     if (x < 10 || y < 10 || x > Hy->UpdateRangeEnd.x - 10 || y > Hy->
24         UpdateRangeEnd.y - 10)
25     {
26         float psi = Hyx->Psi[hyOffset];
27         float decay = Hyx->Decay[x];
28         float amp = Hyx->Amp[x];
29
30         psi = decay * psi + amp * dEz / Configuration->Dx;
31
32         hy = hy - db * Configuration->Dx * psi;
33
34         Hyx->Psi[hyOffset] = psi;
35     }
36
37     Hy->Data[hyOffset] = hy;
38 }
```

Listing 5.6. CUDA kernel for updating H_Y

Note that all E updates occur simultaneously, as do all H fields. However, given the dependence between the E and H fields, the E field update kernels must complete before the H fields are updated.

The simulator repeats this operation until the application is closed, or the desired number of frames are completed.

Finally, the completed field arrays are copied to the host from the GPU, and saved to disk in bitmap and CSV format for later analysis.

5.5. Visualizer

If enabled¹, the visualizer application block provides interactive display of the simulation.

When running a simulation, the user may optionally specify a visualizer update frequency, indicating the number of simulation frames that should complete between visualizer updates. This aids in reducing the visualizer's performance impact.

A window and OpenGL context are created using GLFW and the glLoadGen OpenGL extension loader. An OpenGL pixel buffer object (PBO) is allocated to contain a copy² of the field that the user wishes to observe³.

The visualizer also creates a screen-aligned quad on which the field texture will be rendered, and allocates a texture object which is then bound to the PBOFigure 5.1.

¹The visualizer requires some GPU overhead. As such, its use may affect simulator performance.

²The PBO may be of different dimensions than the simulation domain. Since the PBO is used only for visualization, it is not necessary for it to contain the full-resolution field.

³The visualizer provides the ability to dynamically select which field(s) should be displayed

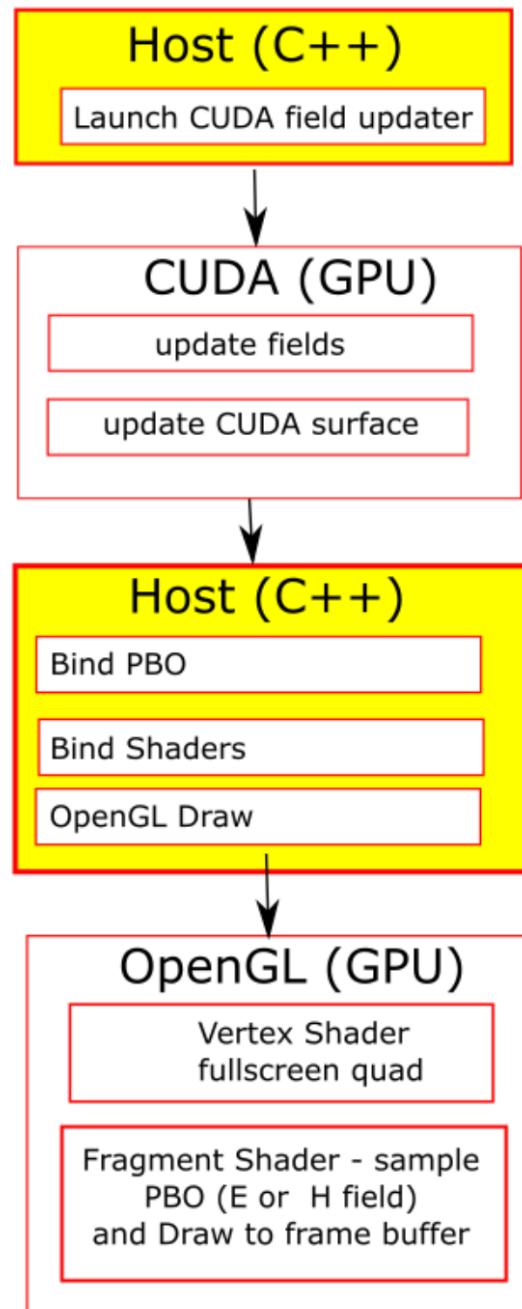


Figure 5.1. Visualizer Update Pipeline

After the required number of frames have been completed, the visualizer launches a CUDA kernel which samples the selected field and populates the PBO.

```

1 --global__ void visualizerUpdatePreviewTexture(
2     cudaSurfaceObject_t image
3     , int imageWidth
4     , int imageHeight
5     , float *fieldData
6     , int fieldWidth
7     , int fieldHeight
8     , float *materials
9     )
10 {
11     unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
12     unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
13     int readX = (int)(x * fieldWidth * 1.f / imageWidth);
14     int readY = (int)(y * fieldHeight * 1.f / imageHeight);
15     float fieldValue = fieldData[readY * fieldWidth + readX];
16     float cb = materials[readY * fieldWidth + readX];
17     float4 color = make_float4(fieldValue, cb, 0, 1); color.w = threadIdx.
18         x == 0 || threadIdx.y == 0;
19     surf2Dwrite(color, image, x * sizeof(float4), y, cudaBoundaryModeClamp
    );
}

```

Listing 5.7. CUDA kernel for updating visualizer pixel buffer object

```

1 #version 430 core
2 uniform sampler2D CudaTexture;
3 uniform vec2 TextureSize;
4 uniform vec2 WindowSize;
5 in vec3 Position;
6 out vec4 vColor;
7 out vec3 vWorldPosition;
8 out vec2 vTexCoord;

```

```

9 void main()
10 {
11     vec4 worldPos = vec4(Position, 1);
12     /// scale down the quad coordinates to match the texture aspect ratio
13     if (TextureSize.x > TextureSize.y)
14         worldPos.y *= TextureSize.y / TextureSize.x;
15     else
16         worldPos.x *= TextureSize.x / TextureSize.y;
17     float windowAspect = WindowSize.x / WindowSize.y;
18     if (windowAspect > 1)
19         worldPos.y *= windowAspect;
20     else if (windowAspect < 1)
21         worldPos.x /= windowAspect;
22     if (abs(worldPos.x) > 1) worldPos.xy /= abs(worldPos.x);
23     if (abs(worldPos.y) > 1) worldPos.xy /= abs(worldPos.y);
24     vWorldPosition = worldPos.xyz;
25     vTexCoord = vec2(Position.x, -Position.y);
26     gl_Position = worldPos;
27
28     vColor = vec4(1);
29 }
```

Listing 5.8. GLSL Vertex Shader

```

1 #version 430 core
2 uniform sampler2D CudaTexture;
3 uniform float ColorScale = 100.f;
4 in vec2 vTexCoord;
5 in vec3 vWorldPosition;
6 in vec4 vColor;
7 out vec4 fragmentColor;
8 vec4 saturate(vec4 val) { return clamp(val, vec4(0), vec4(1)); }
```

```

9 void main() {
10    vec2 texCoord = vTexCoord / 2 + vec2(0.5);
11    vec4 texel = texture2D(CudaTexture, texCoord);
12    float r = texel.r * ColorScale;
13    float b = -r;
14    float dielectric = texel.g;
15    float g = (dielectric >= 0.5) ? 0 : dielectric * 2;
16    vec4 t = saturate(vec4(r,g,b,1));
17    fragmentColor = saturate(vec4(r,g,b,1));
18 }
```

Listing 5.9. GLSL Fragment Shader

A completed PBO is bound to a uniform input for a shader (Listing 5.8, Listing 5.9) listing which renders the texture to the visualizer window. This process runs continuously. However, the texture is only updated at the frequency requested by the user when the simulation was launched.

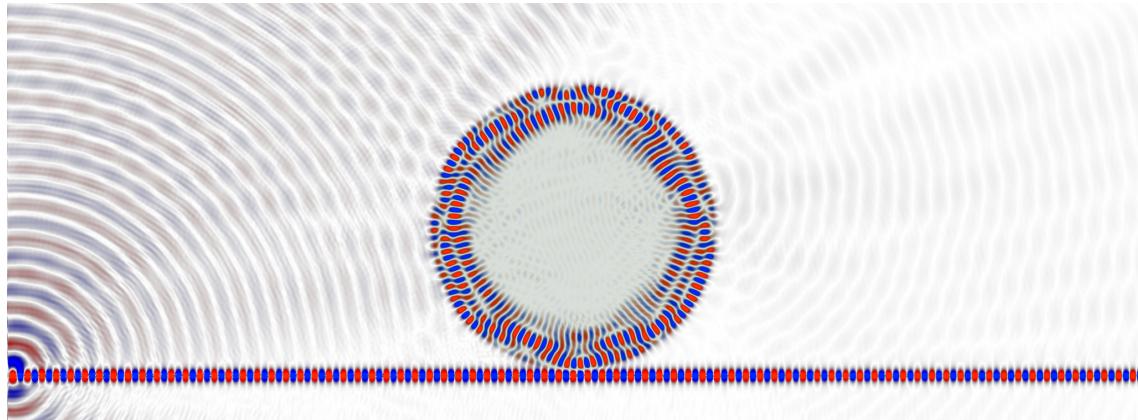


Figure 5.2. 2D Whispering Gallery Mode Sensor

5.6. Modeling approach

For simplicity, models are defined in any of a number of standard 32-bit color image formats. In a 32-bit per pixel image, each color has 8-bit red, green, blue and alpha values. As mentioned in the model processor (section 5.3) section, each component is used to indicate some data about a given point in the simulation domain:

Table 5.2. Color component usage

Component	Meaning	Interpretation
Red	Source	normalized wavelength of the source
Green	Dielectric	$\epsilon_r = \text{green} * \frac{\epsilon_{max}}{255.0}$
Blue	Monitor	ID of the monitor to which this texel belongs
Alpha	Reserved	Reserved for future use.

Whenever a non-zero blue (monitor) value is encountered, it is treated as an identifier ("ID") of a monitor. If the given ID has not yet been encountered, a new monitor is created. Any subsequent locations with the same ID are added to the corresponding monitor. This allows monitors to be of any shape or size.

Using a tool such as Adobe Photoshop or Microsoft Paint, the user can specify all required simulation input - sources, monitors and dielectric - in an intuitive fashion. Alternatively, these bitmaps could be generated by a custom tool which would voxelize a CAD model, assigning color components based on the model's metadata or object properties.

A significant advantage of this approach over the CSG method used in Meep is that all constructs - sources, monitors and dielectric - can be of any shape that can be drawn in a bitmap. In theory, any voxel-based modeling tool could be used to build models in higher dimensions.



Figure 5.3. Arbitrarily-shaped source (Red pixels)

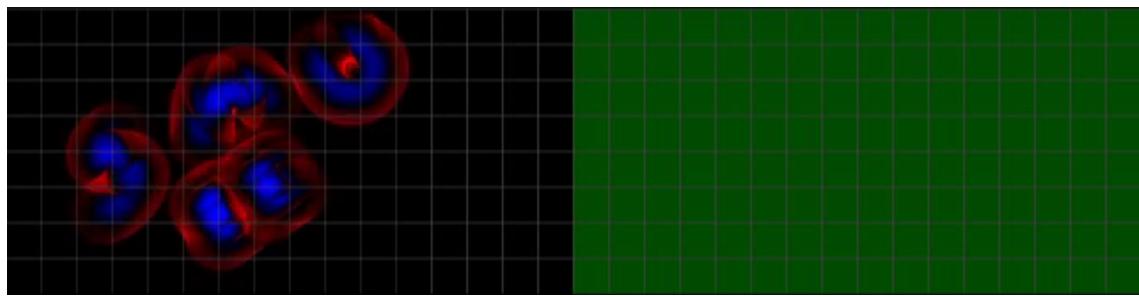


Figure 5.4. Arbitrarily-shaped source after 20 frames

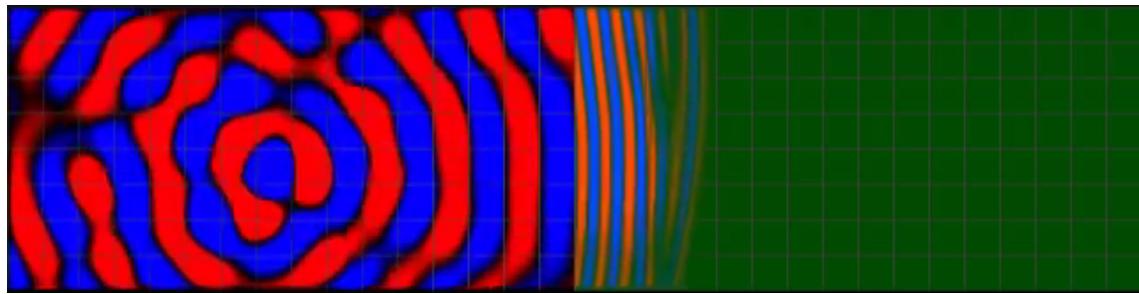


Figure 5.5. Arbitrarily-shaped source after 100 frames

5.7. Testing and Validation Methodology

In order to validate the GPU simulation results, a 2D TM_Z simulation was executed.

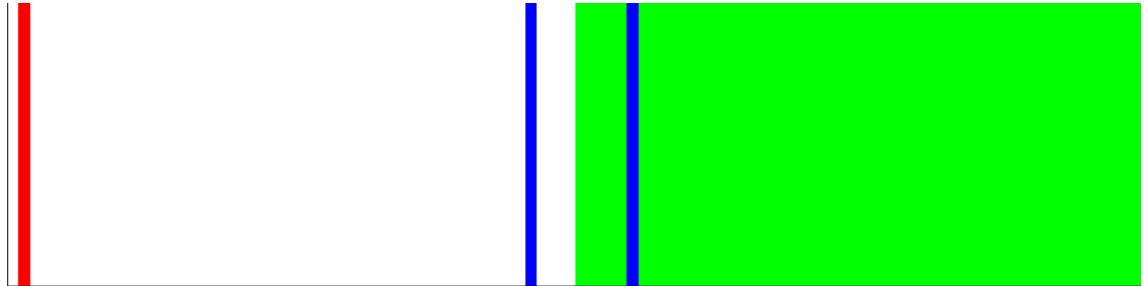


Figure 5.6. TM_Z Test Model

As can be seen in Figure 5.6, the simulation includes a plane wave source (red line), monitors for incident and transmitted waves (blue), and a dielectric slab with $\epsilon_R = 9$ (green). An additional monitor at the source location is not clearly visible due to its relatively low ID ($B = 5$).

The simulation was first run with $\epsilon_R = 1$ to record the incident magnitude in absence of any reflective interfaces. The simulation was then run with $\epsilon_R = 9$ to record combined incidence and reflection, as well as transmittance within the dielectric.

In a post-processing step, the reflective magnitude was found by subtracting the incident wave magnitude obtained from the $\epsilon_R = 1$ simulation from the combined incidence and reflection magnitudes obtained from the $\epsilon_R = 9$ simulation.

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

5.7.1. Analytical Result

In the test configuration, a normalized ($\lambda = 1$) TM_Z plane wave is normally incident upon a dielectric interface with $\epsilon_R = 9$.

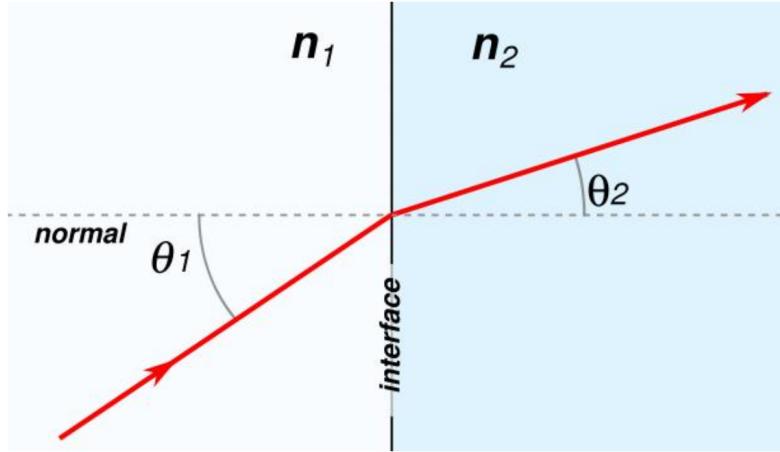


Figure 5.7. Snell's Law

Snell's Law states that the ratio of the refractive indices of the media at an interface, along with the angle of incidence, determine the angle of transmittance. (See Figure 5.7)

Mathematically, this relationship may be expressed as

$$n_1 \sin \theta_1 = n_2 \sin \theta_2 \quad (5.1)$$

In the test model, the index n_2 is calculated using the formula:

$$n = \sqrt{\epsilon_0 \epsilon_R \mu_0 \mu_R} \quad (5.2)$$

In our simulator, ϵ_0 and μ_0 are normalized to 1. Similarly, in the non-magnetic media used in this case,

$$\mu_R = 1 \quad (5.3)$$

Using our test value of $\epsilon_R = 9$ gives

$$n_2 = \sqrt{9} = 3 \quad (5.4)$$

For a normally-incident plane wave, the incident and refraction angles are

$$\theta_I = 0 \quad (5.5)$$

and

$$\theta_T = 0 \quad (5.6)$$

Evaluating the Fresnel equations for the reflection and transmission of a TM_Z wave,

$$r = \frac{n_2 \cos \theta_I - n_1 \cos \theta_T}{n_1 \cos \theta_T + n_2 \cos \theta_I} \quad (5.7)$$

$$t = \frac{2n_1 \cos \theta_I}{n_1 \cos \theta_T + n_2 \cos \theta_I} \quad (5.8)$$

gives the coefficients:

$$r = \frac{3 * 1 - 1 * 1}{1 * 1 + 3 * 1} = \frac{1}{2} \quad (5.9)$$

$$t = \frac{2 * 1 * 1}{1 * 1 + 3 * 1} = \frac{1}{2} \quad (5.10)$$

In this case, given a dielectric constant $\epsilon_R = 9$, the reflection and transmission coefficients are equal.

5.7.2. Numerical Result

The observed steady-state output of the simulation for the baseline case ($\epsilon_R = 1$) is shown in Figure 5.8.

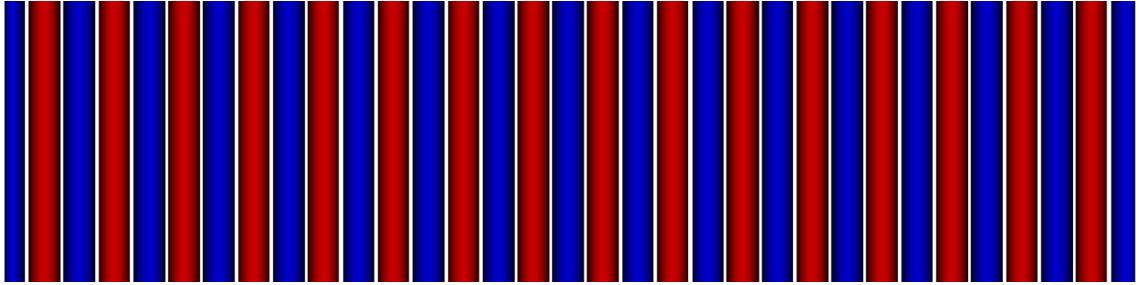


Figure 5.8. Plane wave with $\epsilon_R = 1$

The time-averaged (RMS) numerical output for the incident and transmitted monitors is

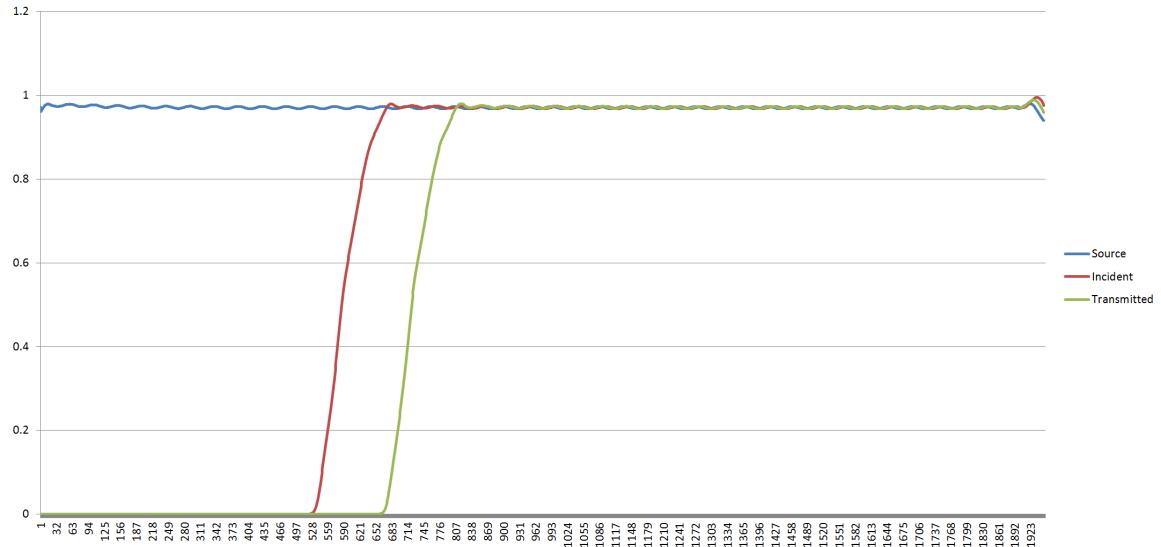


Figure 5.9. Time-averaged (RMS) output in free space $\epsilon_R = 1$

As expected, the incident and transmitted magnitudes equal the source magnitude, offset by the time it takes for the wave to propagate from the source to the monitors.

The observed output of the same simulation, executed with dielectric constant $\epsilon_R = 9$,

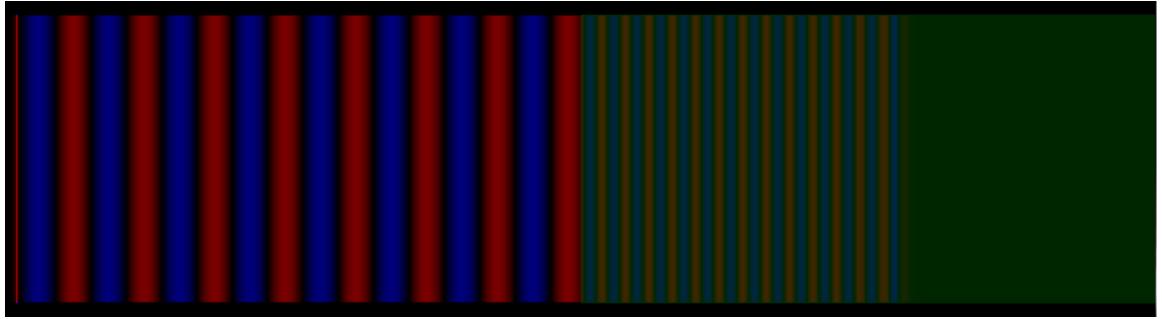


Figure 5.10. Steady state with $\epsilon_R = 9$

Note the difference in λ once the wave enters the dielectric (green area). This simulation produces the following raw output:

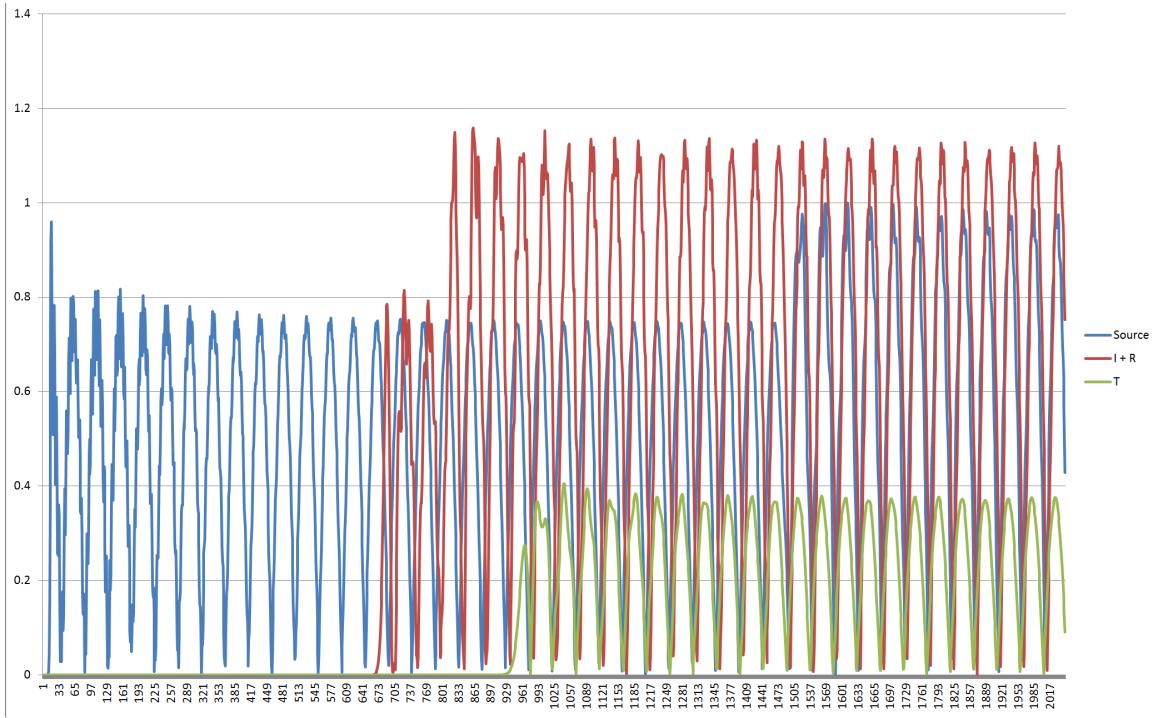


Figure 5.11. Raw monitor output with $\epsilon_R = 9$

The time-averaged power for the monitors is shown in Figure 5.12.

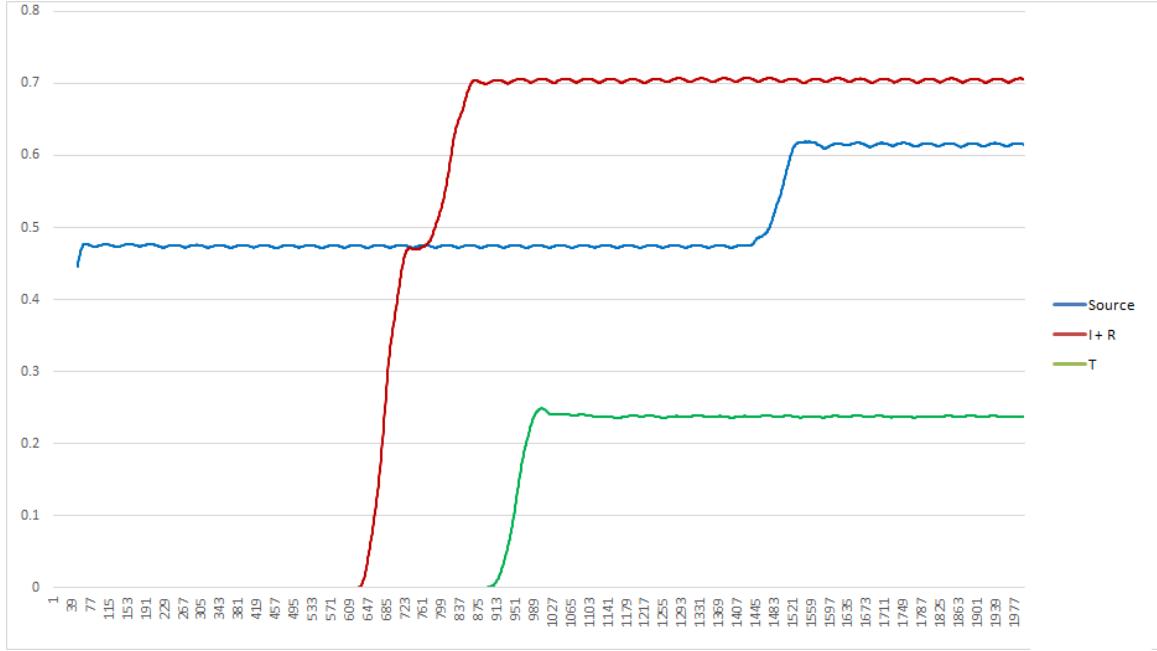


Figure 5.12. RMS output with $\epsilon_R = 9$

Subtracting the incident magnitude from the simulation with $\epsilon_R = 1$ from the second simulation with $\epsilon_R = 9$ gives the following result (Figure 5.13)

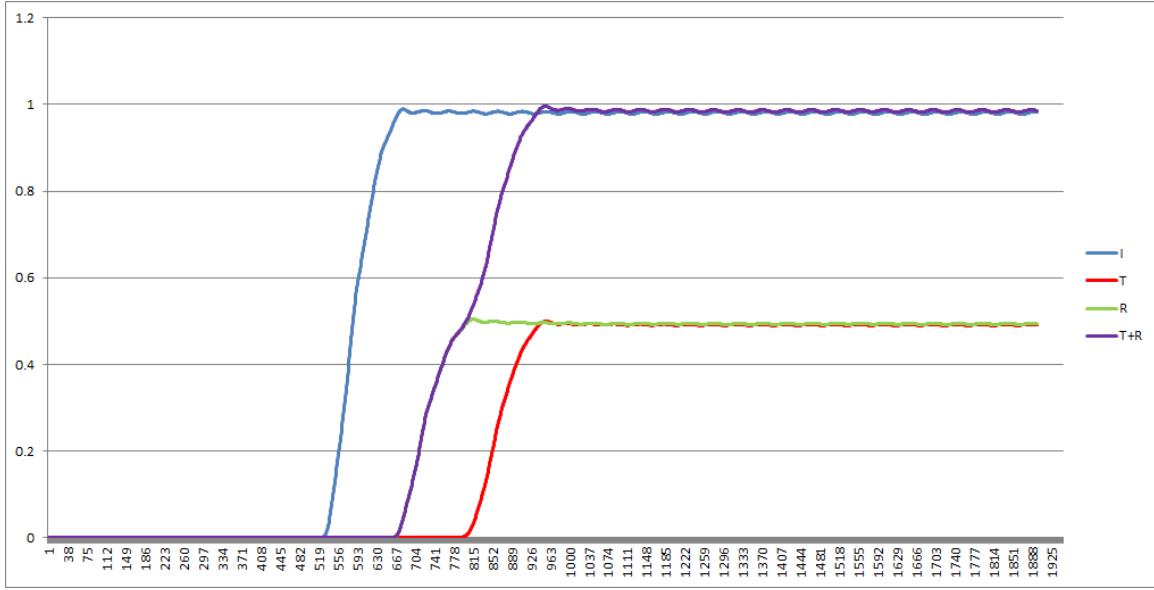


Figure 5.13. Normalized output with $\epsilon_R = 9$

In this case, the average of each of the relevant values are determined to be:

$$I_{avg} = 0.96214\dots$$

$$T_{avg} = 0.4882\dots$$

$$R_{avg} = 0.484\dots$$

Conservation of energy requires that the sum of the transmitted (T) and reflected (R) magnitudes equals the incident magnitude (I).

$$I = T + R \quad (5.11)$$

The computed error of the analytical result is

$$I_{error} = \left| \frac{I_{avg} - (T_{avg} + R_{avg})}{I_{avg}} \right| \quad (5.12)$$

Gives

$$\left| \frac{0.9621 - (0.4882 + 0.4843)}{0.9621} \right| = 1.011\%$$

Comparing to the analytically-derived Fresnel coefficients,

$$T = \frac{T_{avg}}{I_{avg}} = \frac{0.4882219...}{0.962135...} = 0.507 \quad (5.13)$$

$$R = \frac{R_{avg}}{I_{avg}} = \frac{0.484332...}{0.962135...} = 0.503 \quad (5.14)$$

The error contribution of each component may be expressed as:

$$T_{error} = \left| 1 - \frac{T_{computed}}{T_{analytical}} \right| = \left| 1 - \frac{0.507}{0.5} \right| = 1.4\% \quad (5.15)$$

$$R_{error} = \left| 1 - \frac{R_{computed}}{R_{analytical}} \right| = \left| 1 - \frac{0.503}{0.5} \right| = 0.6\% \quad (5.16)$$

Careful analysis of different simulation results indicates that this error is due to floating-point precision errors. As every frame output depends on the state of the previous frame, small errors are compounded over time.

Chapter 6

Results

In order to determine the effectiveness of the GPU-based FDTD implementation, we compare performance of GPU and CPU implementations. We measure performance as a function of total execution time for a given domain size and number of time steps, as well as number of cell operations completed per second.

6.1. Test Environment

Tests were performed on a 2015 Dell XPS 9550 with 32GB of RAM, Intel i7 2600 CPU and NVIDIA 960M GPU. GoLightly GPU tests were executed under Microsoft Windows 10, while Meep CPU tests were run under Ubuntu Desktop version 16.04.

6.2. Performance Metrics

Metrics were calculated using domain sizes ranging from 128x64 to 8192x4096 over 5000 frames. (In this case, a frame represents a complete time step wherein all E and H fields are updated.) For benchmarking purposes, the visualizer was disabled.

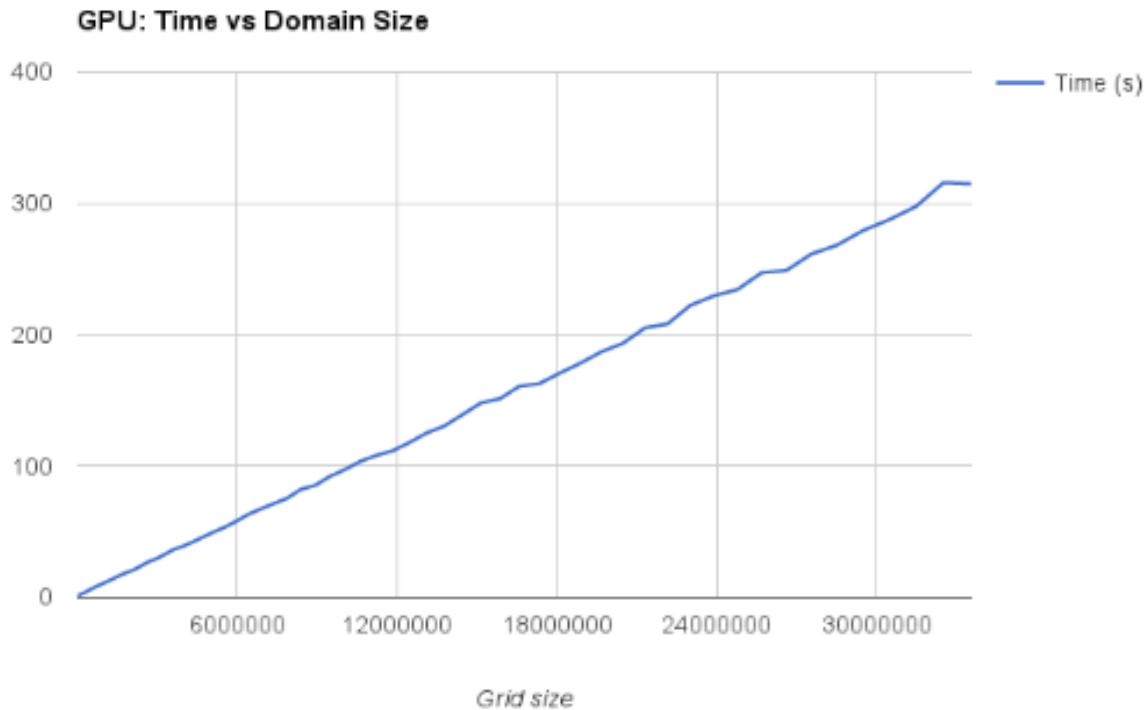


Figure 6.1. GoLightly: seconds for 5000 frames with the given domain size

Figure 6.1 shows that computation time increases linearly as a function of simulation domain size.

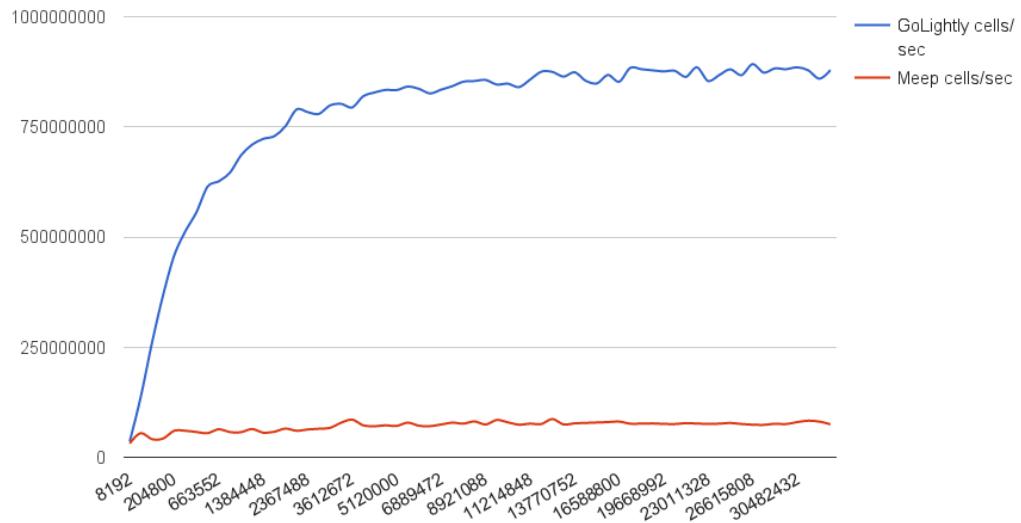


Figure 6.2. GoLightly: Completed E and H updates per second

Note that GPU throughput (represented in Figure 6.2 as “cell” operations per second) increases dramatically as the domain size increases, until GPU initialization overhead is overcome by computation time.

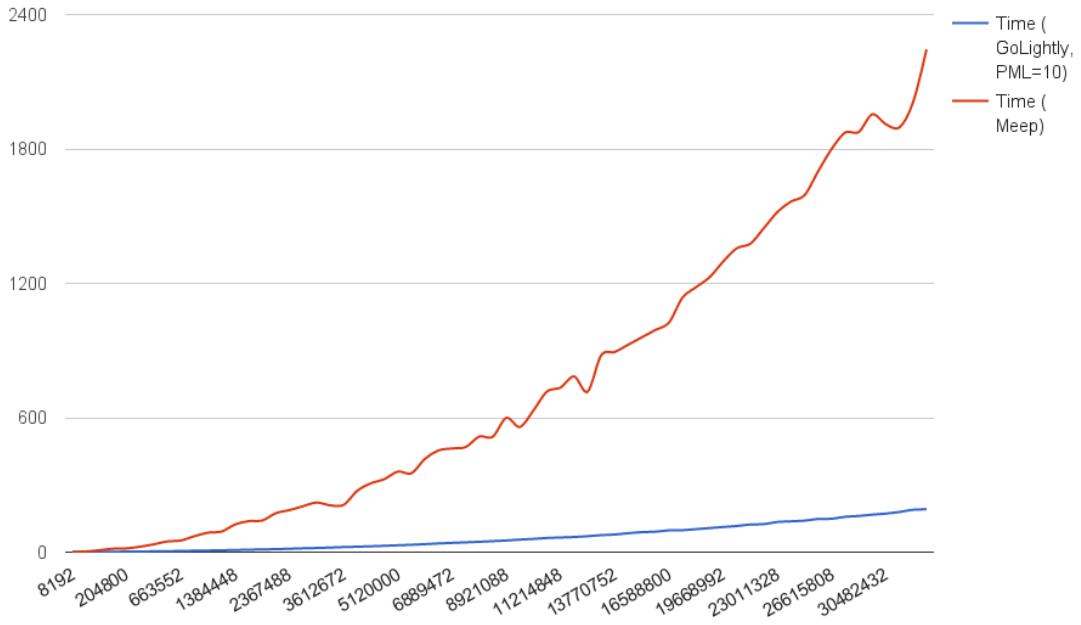


Figure 6.3. GoLightly vs Meep

Figure 6.3 In Figure 6.3, note some CPU performance variance near the middle and end of the graph. Multiple runs consistently exhibited this behavior. While the cause of these variances is not clear, it has been reproduced on different hardware and operating system versions. This may be caused by several factors, including:

- Memory alignment. When reading or writing from memory, CPUs perform best when dealing with word-aligned data sizes. The dimensions of the simulation domain affect this alignment.
- Multi-tasking preemption. In multitasking operating systems such as Linux or Windows, background tasks may preempt foreground tasks, resulting performance degradation.
- Memory latency. Relatively slow access to out-of-core memory may cause a pipeline stall.

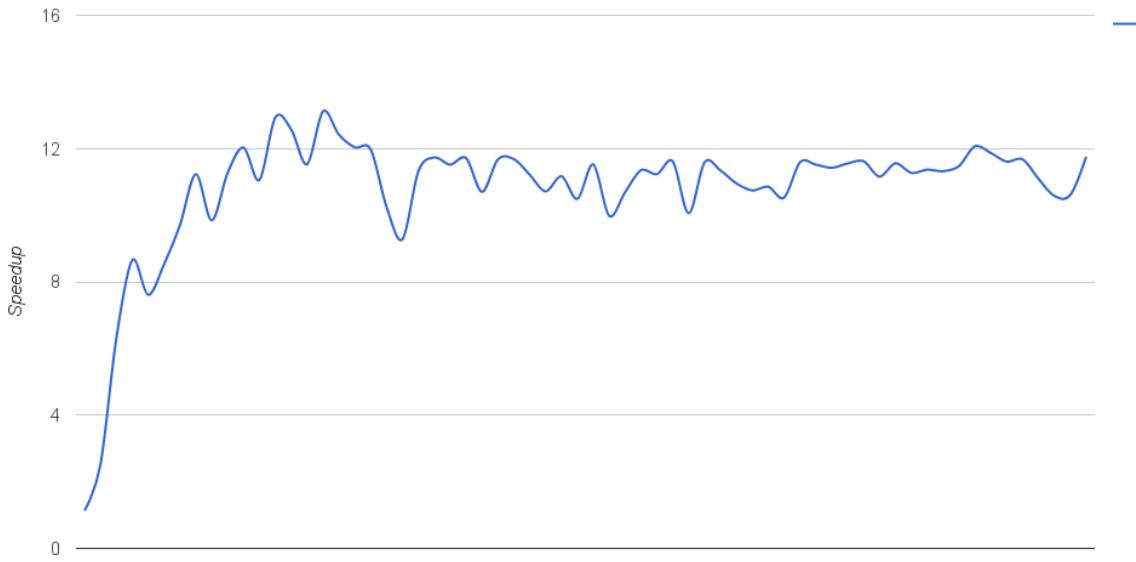


Figure 6.4. Speedup - Meep Time / GoLightly Time

Figure 6.4 shows a speedup ranging from 1.2 to 12, depending on the domain size. At lower resolutions, the overhead of initializing assets on the GPU can take more time than the simulation. In that case, the CPU may outperform the GPU solution.

6.3. Optimization and Enhancements

Although a 1200% speed increase is significant, there is much room for improvement. GPUs provide different memory spaces that vary in size and access speed. In addition to global device memory, each wave¹ has shared memory² and local memory³.

While global memory is the most flexible and plentiful - typically on the order of

¹A wave, wavefront or warp is a group of 32 or 64 threads allocated to a common stream processor unit.

²Shared memory is physically local to the ALU and accessible by all threads within the warp.

³Each thread has local memory which is not accessible to any other thread

gigabytes on current generation hardware - it is also the slowest.

Shared memory is common to all threads in a wave. It can be used for intra-thread synchronization and resource sharing, and is significantly faster than global memory.

Finally, local memory provides thread-local storage. Local memory provides the lowest latency of all memory spaces.

In its current form, GoLightly makes little use of shared or local memory. Modifying the application to take advantage of those would improve performance.

Chapter 7

Conclusions

Using the GoLightly simulator, we have shown a speedup potential of up to 12 times that achieved by CPU-based solutions.

7.1. Usability

Although the software, in its current state, may not be suitable for production, it is clear that a GPU-based approach promises substantially improved performance. This results in increased productivity and, ultimately, better products. Eliminating the script-only modeling approach enforced by other packages makes utilization of this application significantly easier, and therefore accessible to a wider audience.

7.2. Future Work

The vastly-improved performance afforded by this system presents some interesting opportunities:

7.2.1. GoLightly Improvements

Care must be taken during the design process to avoid nonsensical monitor configurations, such as disjoint or outlying pixels, or inconsistent thickness due to shape aliasing. A model validation step could correct model errors. Sub-pixel averaging would increase effective domain resolution near dielectric boundaries, and reduce structure aliasing.

7.2.2. Genetic Algorithms

Genetic algorithms[9][4] are intended to let software take over a part of the design process. By defining a problem in terms of a number of inputs and creating a fitness function, an application can potentially test many different designs and use a feedback loop to suggest new permutations. This approach has been shown to be successful in such diverse fields as antennae design[3], turbine design[10] and pharmaceutical research[2].

A fast FDTD implementation will facilitate application of this technique to optical circuit design. By defining a problem domain in terms of desired package size, available inputs, allowed waveguide shape, available dielectric properties, and designing an appropriate fitness function, software will be able to rapidly evaluate different designs, and suggest new permutations. This may also dramatically reduce time-to-market and suggest new research avenues.

7.2.3. Arbitrary Domain Shape and PML Sinks

Given the more flexible voxel-based model definition used in GoLightly, it becomes possible to create completely arbitrary domain shapes. Non-rectangular domains with PML "sinks" at any desired position within the domain would allow the designer to more tightly fit the computational domain to the circuit in question, and ignore irrelevant or uninteresting areas.

PML sinks would potentially increase performance by reducing memory requirements. If large sections of a domain could be surrounded by PML, those sections are effectively disjoint from the rest of the domain and therefore may be ignored and, in fact, removed from the simulation.

Although the current implementation treats the domain as a regular grid of identical blocks, this is a programming convenience and is not dictated by the FDTD

algorithm.

In addition to non-rectangular domains, non-rectangular cell shapes could improve simulation fidelity. For example, in a TM_Z simulation, each E_Z component could be treated as the center of a hexagonal grid cell. As such, it would be updated using three derivatives instead of two. While this would increase the computation requirement for each E cell by roughly 50% (in a two-dimensional domain), the improved fidelity may justify the cost in experiments where improved accuracy is valuable. Inclusion of the afore-mentioned PML sink capability could offset this cost.

7.2.4. Load Balancing

Although we have shown that a GPU implementation of FDTD can outperform a CPU implementation, CPUs should not be ignored. It is possible to divide computational load between the CPU and GPU, in a manner analogous to that used to distribute load between machines in a cluster.

This technique, when combined with load balancing between separate machines, would allow the GPU to act as an additional cluster node, providing a more ideal solution which, rather than trading a CPU for a GPU, provides the power of both.

7.3. Final Words

The field of general-purpose GPU computing offers potential yet to be realized in many of the areas where it shows the most promise. Leveraging this commonly available, underutilized resource will enable shorter, more robust design cycles, and facilitate exploration of more sophisticated waveguide architectures.

Appendix A

APPENDIX

A.1. Meep scripts

The shell script shown in listing A.1 calls Meep in a loop, increasing the domain size from 128x64 cells (not including PML layers) to 8192x4096 cells in increments of 128x64 cells.

```
1 for ((w=128; w <= 8192; w+=128))
2 do
3 echo "width " $w;
4 echo "height " ${((w / 2))};
5 meep dw=$w dh=${((w/2))} benchmark.ctl
6 done;
```

Listing A.1. Meep shell script

The Meep CTL code in listing A.2 defines a simulation with a dielectric block with $\epsilon_{max} = 9$. The simulation is run for 5000 frames. During testing, the simulation is timed and the results are written to a CSV file for later analysis.

```
1 (define-param dw 4096)
2 (define-param dh 2048)
3 (set! geometry-lattice (make-lattice (size dw dh no-size)))
4 (set! geometry (list
5   (make-block (center 256 0) (size 512 infinity infinity)
6   (material (make-dielectric (epsilon 9))))))
7 (set! sources (list
```

```
8   (make source
9     (src (make continuous-src (frequency 0.15)))
10    (component Ez)
11    (center 128 0))))
12 (set! pml-layers (list (make pml (thickness 1.0))))
13 (set! resolution 1)
14 (run-until 5000)
```

Listing A.2. Meep simulation configuration

REFERENCES

- [1] BERENGER, J.-P. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics* 114, 2 (1994), 185 – 200.
- [2] CHI, H.-M., MOSKOWITZ, H., ERSOY, O. K., ALTINKEMER, K., GAVIN, P. F., HUFF, B. E., AND OLSEN, B. A. Machine learning and genetic algorithms in pharmaceutical development and manufacturing processes. *Decis. Support Syst.* 48, 1 (Dec. 2009), 69–80.
- [3] GLOBUS, A., HORNBY, G., LINDEN, D., AND LOHN, J. Automated antenna design with evolutionary algorithms.
- [4] GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
- [5] KROGSTROP, P., JØRGENSEN, H. I., HEISS, M., DEMICHEL, O., HOLM, J. V., AAGESEN, M., NYGARD, J., AND MORRAL, A. F. Single-nanowire solar cells beyond the shockley-queisser limit. *Nature Photonics* 7, 4 (2013), 306–310.
- [6] LEVIN, M., McCUALEY, A. P., RODRIGUEZ, A. W., REID, M. H., AND JOHNSON, S. G. Casimir repulsion between metallic objects in vacuum. *Physical review letters* 105, 9 (2010), 090403.
- [7] LIU, V., AND FAN, S. S 4: A free electromagnetic solver for layered periodic structures. *Computer Physics Communications* 183, 10 (2012), 2233–2244.
- [8] MASSINGILL, B. L., MATTSON, T. G., AND SANDERS, B. A. Simd: An additional pattern for plpp (pattern language for parallel programming). In *Proceedings of the 14th Conference on Pattern Languages of Programs* (New York, NY, USA, 2007), PLOP ’07, ACM, pp. 6:1–6:15.
- [9] MITCHELL, M. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.
- [10] MOSETTI, G., POLONI, C., AND DIVIACCO, B. Optimization of wind turbine positioning in large windfarms by means of a genetic algorithm. *Journal of Wind Engineering and Industrial Aerodynamics* 51, 1 (1994), 105 – 116.

- [11] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable parallel programming with cuda. *Queue* 6, 2 (Mar. 2008), 40–53.
- [12] NVIDIA CORPORATION. NVIDIA CUDA C programming guide, 2010. Version 3.2.
- [13] NVIDIA CORPORATION. U.s. to build two flagship supercomputers for national labs, 2014.
- [14] OSKOOI, A. F., ROUNDY, D., IBANESCU, M., BERMEL, P., JOANNOPOULOS, J. D., AND JOHNSON, S. G. MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method. *Computer Physics Communications* 181 (January 2010), 687–702.
- [15] RAINA, R., MADHAVAN, A., AND NG, A. Y. Largescale deep unsupervised learning using graphics processors. In *International Conf. on Machine Learning* (2009).
- [16] RODRIGUEZ, P., PATTICHIS, M., AND JORDAN, R. Parallel single instruction multiple data (simd) fft: Algorithm and implementation. Tech. Rep. HPCERC2003-002, January 2003.
- [17] SHI, L., LIU, W., ZHANG, H., XIE, Y., AND WANG, D. A survey of gpu-based medical image computing techniques. *Quantitative Imaging in Medicine and Surgery* 2, 3 (2012).
- [18] STRZODKA, R., COHEN, J., AND POSEY, S. Gpu-accelerated algebraic multi-grid for applied {CFD}. *Procedia Engineering* 61 (2013), 381 – 387. 25th International Conference on Parallel Computational Fluid Dynamics.
- [19] VYNCK, K., BURRESI, M., RIBOLI, F., AND WIERSMA, D. S. Photon management in two-dimensional disordered media. *Nature materials* 11, 12 (2012), 1017–1022.
- [20] WU, R., ZHANG, B., AND HSU, M. Clustering billions of data points using gpus. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop* (2009), ACM, pp. 1–6.
- [21] XIE, J., YU, H., AND MA, K.-L. Interactive ray casting of geodesic grids. *Computer Graphics Forum* 32, 3pt4 (2013), 481–490.
- [22] YEE, K. Numerical solution of initial boundary value problems involving maxwell’s equations in isotropic media. *IEEE Transactions on Antennas and Propagation* 14, 3 (1966), 302–307.