IMPLEMENTATION OF THE FINITE-DIFFERENCE TIME-DOMAIN METHOD USING GRAPHICS PROCESSING UNITS

Approved by:

Dr. Marc Christensen

Dr. Nathan Huntoon

Professor Ira Greenberg

IMPLEMENTATION OF THE FINITE-DIFFERENCE TIME-DOMAIN METHOD USING GRAPHICS PROCESSING UNITS

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 Electrical Engineering

with a

Major in Electrical Engineering

by

S. David Lively

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

August 1, 2016

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

Implementation of the Finite-Difference Time-Domain
Method Using Graphics Processing Units

Advisor: Professor Marc Christensen
Master of Electrical Engineering degree conferred August 1, 2016
Thesis completed August 1, 2016

Traditionally, optical circuit design is tested and validated using software which implement numerical modeling techniques such as Beam Propagation, Finite Element Analysis and FDTD.

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.

Application of modern programmable GPGPUs 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 often-overlooked tool.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

LIST OF TABLES

Listings

To Audrey, Wyatt, Walter and Gwendolyn

Chapter 1

Introduction

The FDTD [1] algorithm is the underlying mechanism used by many commercial RF simulation packages, as well as open source software such as MIT's Meep.

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). 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 and artificial intelligence, medical research, signals analysis and other areas which require rapid analysis of large datasets.

Even modern consumer-grade GPUs offer thousands or tens of thousands of processing units (”cores”), while high-end CPUs typically offer 4-8 cores. While the two are not interchangeable, 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 FDTD. Later sections describe a CPU-based implementation (MIT's Meep simulator), 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.

1.1. FDTD Overview

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

1.1.1. Wave equation

In a TM_z time domain simulation, wave equation for E_z is of the form:

$$\frac{\partial E_z}{\partial t} = K * (\frac{\partial H_x}{\partial y} + \frac{\partial H_y}{\partial x}) \tag{1.1}$$

Equation 1.1 states that the temporal derivative (change in amplitude) of E_z is a function of the Y-axis spatial derivative of the H_x field and the X-axis spatial derivative of the H_y field.

In order to apply this equation to a computational domain, FDTD defines a cell-based discretization strategy.

1.1.2. Yee Cell

Yee [1] defines a computational unit known as a "cell." The cell describes how each field component within a domain is related to it's coupled fields. For instance, in a 2D TM_z simulation, E_Z depends on adjacent H_y and H_x components. The cell format used in such a simulation is of the form shown in 1.1.

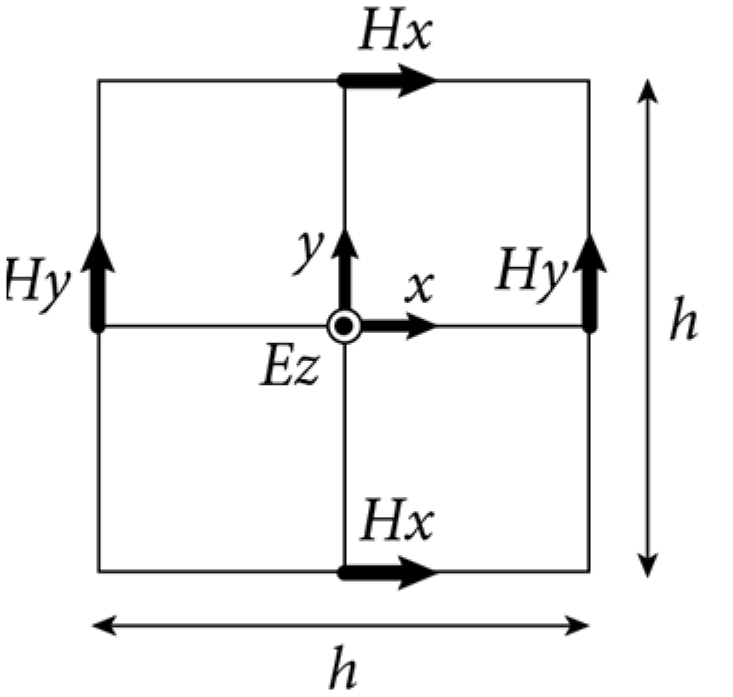

Figure 1.1. 2D TM_Z Yee Cell

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

$$E_{z i,j}^{t} = C_a * E_{z i,j}^{t-1} + C_b * (H_{x_{i,j+\frac{1}{2}}}^{t-\frac{1}{2}} - H_{x_{i,j-\frac{1}{2}}}^{t-\frac{1}{2}}) + C_b * (H_{y_{i+\frac{1}{2},j}}^{t-\frac{1}{2}} - H_x y_{i-\frac{1}{2},j}^{t-\frac{1}{2}}) \tag{1.2}$$

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

$$H_{x i,j}^{t} = D_a * H_{x i,j}^{t-1} + D_b * (E_{z_{i,j+\frac{1}{2}}}^{t-\frac{1}{2}} - E_{z_{i,j-\frac{1}{2}}}^{t-\frac{1}{2}}) \tag{1.3}$$

$$H_{y i,j}^{t} = D_a * H_{y i,j}^{t-1} + D_b * (E_{z_{i+\frac{1}{2},j}}^{t-\frac{1}{2}} - E_{z_{i-\frac{1}{2},j}}^{t-\frac{1}{2}}) \tag{1.4}$$

Chapter 2

Device Architecture

CPUs and GPUs each offer advantages for different computational tasks. Multi-core CPUs offer complete, semi-independent cores which are effectively discrete processsors. GPUs, however, offer large-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.

In a modern multi-core CPU, each core provides a dedicated ALU and register set. This allows each core to operate as an independent device. This architecture is advantageous when the device is required to perform disparate operations.

However, this approach also introduces some performance limitations. If we wish to perform identical operations on large datasets, we are limited to 4-8 threads. The same processor that excels at executing many different tasks at the same time performs sub-optimally.

When running FDTD on a CPU, each core's ALU executes essentially the same operations on a dedicated register set. Each ALU performs the same operation at the same time, indicating that the additional ALUs are redundant. Thus, the flexible, general-purpose nature of a CPU becomes a liability.

2.2. GPU

GPUs were initially designed for one thing: to determine, as quickly as possibly, what color a pixel should be.

.... more history stuff here...

2.2.1. SIMD

GPUs implement what is known as a single-instruction (SIMD), multiple-data processing model.

In a SIMD architecture, a core may consist of a single ALU, with multiple register banks. Separate ”threads” load different data into dedicated register banks. The ALU executes identitical operations on all registers simultaneously.

The approach provides some benefits and limitations:

- 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, eliminating the need to load or monitor cache behavior per thread.
- TODO

2.2.2. FDTD in SIMD

TODO: why FDTD maps well to GPUs

Chapter 3

Meep

Meep is a full-featured, open-source 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.

3.1. Modeling

One limitation of Meep is it's use of an obscure, uncommon scripting language, SCHEME. Models are defined in terms of constructive solid geometry (CSG) commands, whereby the user describes their model in terms of boolean operations and regular polyhedra.

While adequate for simple models, constructing an arbitrarily-shaped or dynamic structure in this way may be difficult. In practice, proprietary software may be used to convert more complex model definitions created in other software into a format usable by Meep.

It is worth noting that Meep provides a "material function" capability. This allows the user to dynamically determine the material properties of a point in space using their own algorithm rather than defining their model using CSG. However, to advantage of this capability, the user must employ additional software or custom programming, further increasing the complexity of an already complicated system.

3.2. Performance

Meep is a mature, highly-optimized suite of tools. While complex to configure and use, it scales well across multiple machines, relying on the MPI protocol to keep nodes within a simulation cluster in sync.

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 number of data points that can be 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 evenly across the resulting cluster.

This sort of cluster configuration incurs its own overhead. Although a domain may be divided into chunks and distributed across cluster nodes, FDTD boundary conditions require that, at some point, parts of the divided domain must be exchanged between nodes to maintain continuity. This necessitates installation of a high-speed local network and supporting hardware.

3.3. Limitations

The manner in which models are defined, poor performance on single machines and laptops, incompatibility with Microsoft Windows all indicate that this system is not suitable for rapid design iteration. To that end, we present an alternative, GPU-based approach. (See chapter 4)

Chapter 4

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.

4.1. 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 oft-underutilized 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. By building the software specifically for Microsoft Windows, we ensure that it is compatible with the most common desktop operating system.

4.2. Architecture

GoLightly comprises three primary application blocks:

- Model Processor 4.2.1
- Simulator 4.2.2
- Visualizer 4.2.3

FLOWCHART HERE?

4.2.1. 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 a max ϵ are specified.

Table 4.1. 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

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.

Once the model image is loaded, the MP iterates through each element in the dielectric array. (See 4.2.1)

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 $= B$

```
for(int j = 0; j < media.Size.y; j++)
{
  int sourceY = j * height / media.Size.y;
  for(int i = 0; i < media.Size.x; i++)
  {
    int sourceX = i * width / media.Size.x;
    unsigned int sourceOffset = channels * (sourceY * width + sourceX);
    unsigned int mediaOffset = j * media.Size.x + i;
    unsigned char red =   bytes[sourceOffset + 0];
    unsigned char green = bytes[sourceOffset + 1];
    unsigned char blue =  bytes[sourceOffset + 2];
    // is this pixel part of a source?
    if (red > 128)
      sourceOffsets.push_back(mediaOffset);

    /// fill default waveguide material (parameter n)
    if (green > 0)
    {
      // interpolate n based on green value.
      media.HostArray[mediaOffset] = epsilonMax * green * 1.f / 255;
    }
  }
}
```

Listing 4.1. 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.

4.2.2. 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 (4.2.2) 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 (4.2.2) is generated from the host descriptor and passed to the kernels.

```
struct FieldDescriptor
{
  /// <summary>
  /// describes a split-field boundary region for PML
  /// </summary>
  struct BoundaryDescriptor
  {
    FieldType Name;
    FieldDirection Direction;

    /// CPU-resident fields
    float *Amp;
    float *Psi;
    float *Decay;

    BoundaryDescriptor *DeviceDescriptor;

    unsigned int AmpDecayLength;

  private:
    CudaHelper *m_cuda;
  };


  float DefaultValue;
  FieldType Name;

  dim3 Size;
  dim3 UpdateRangeStart;
  dim3 UpdateRangeEnd;

  vector<float> HostArray;
  float *DeviceArray;

  DeviceFieldDescriptor *DeviceDescriptor;

  vector<GridBlock> GridBlocks;
  map<FieldType,shared_ptr<BoundaryDescriptor>> Boundaries;
```

```
};
```

Listing 4.2. Host Field Descriptor structure

asdfasdfasdf

```
enum class FieldDirection { X,Y,Z };

struct DeviceFieldDescriptor
{
  FieldType Name;
  dim3 Size;
  dim3 UpdateRangeStart;
  dim3 UpdateRangeEnd;

  float *Data;
};
```

Listing 4.3. 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:

```
__global__ void UpdateEz(
  dim3 threadOffset
  )
{
  unsigned int x = threadOffset.x + blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int y = threadOffset.y + blockIdx.y * blockDim.y + threadIdx.y;

```

```
if (y < 1 || x < 1)
  return;

float cb = Cb->Data[y * Cb->Size.x + x];

unsigned int center = y * Ez->Size.x + x;
float hxBottom = Hx->Data[y * Hx->Size.x + x];
float hxTop = Hx->Data[(y - 1) * Hx->Size.x + x];
float dhx = (hxBottom - hxTop);

float hyRight = Hy->Data[y * Hy->Size.x + x];
float hyLeft = Hy->Data[y * Hy->Size.x + x - 1];
float dhy = (hyLeft - hyRight);

float ezxPsi = 0.f;
float ezyPsi = 0.f;

// PML
if (x < 10 || x > Ez->UpdateRangeEnd.x - 10 || y < 10 || y > Ez->UpdateRangeEnd.y - 10)
{
  ezyPsi = Ezy->Decay[y] * Ezy->Psi[center] + Ezy->Amp[y] * dhx;
  Ezy->Psi[center] = ezyPsi;
  ezxPsi = Ezx->Decay[x] * Ezx->Psi[center] + Ezx->Amp[x] * dhy;
  Ezx->Psi[center] = ezxPsi;

}

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

```
}
```

Listing 4.4. 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),

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

corresponds to the FDTD E_Z equation. (See Equation 1.2)

```
__global__ void UpdateHx(dim3 threadOffset)
{
  unsigned int x = threadOffset.x + blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int y = threadOffset.y + blockIdx.y * blockDim.y + threadIdx.y;

  if (y >= Ez->Size.y - 1)
    return;

  unsigned int hxOffset = y * Hx->Size.x + x;
#ifdef USE_MAGNETIC_MATERIALS
  float db = Db->Data[y * Db->Size.x + x];
#else
  const float db = DbDefault;
#endif
  //float ezTop = Ez->Data[y * Ez->Size.x + x];
  //float ezBottom = Ez->Data[(y+1) * Ez->Size.x + x];

  float dEz = Ez->Data[(y + 1) * Ez->Size.x + x] - Ez->Data[y * Ez->Size
```

```
    .x + x];

  float hx = DA * Hx->Data[hxOffset] - db * dEz;

  if (y < 10 || y > Hx->UpdateRangeEnd.y - 10 || x < 10 || x > Hx->
    UpdateRangeEnd.x - 10)
  {
    /// update boundaries
    float decay = Hxy->Decay[y];
    float amp = Hxy->Amp[y];
    float psi = Hxy->Psi[hxOffset];

    psi = decay * psi + amp * dEz / Configuration->Dx;

    Hxy->Psi[hxOffset] = psi;
    hx = hx - db * Configuration->Dx * psi;
  }

  Hx->Data[hxOffset] = hx;
}


_
```

Listing 4.5. CUDA kernel for updating H_X

```
__global__ void UpdateHy(dim3 threadOffset)
{
  unsigned int x = threadOffset.x + blockIdx.x * blockDim.x + threadIdx.
    x;
  unsigned int y = threadOffset.y + blockIdx.y * blockDim.y + threadIdx.
    y;

  if (x >= Ez->Size.x - 1)
    return;

  unsigned int hyOffset = y * Hy->Size.x + x;

#ifdef USE_MAGNETIC_MATERIALS
  float db = Db->Data[y * Db->Size.x + x];
#else
  const float db = DbDefault;
#endif

  float ezLeft = Ez->Data[y * Ez->Size.x + x];
  float ezRight = Ez->Data[y * Ez->Size.x + x + 1];

  float dEz = ezRight - ezLeft;
  float hy = DA * Hy->Data[hyOffset] - db * (ezRight - ezLeft);

  if (x < 10 || y < 10 || x > Hy->UpdateRangeEnd.x - 10 || y > Hy->UpdateRangeEnd.y - 10)
  {

    float psi = Hyx->Psi[hyOffset];
    float decay = Hyx->Decay[x];
    float amp = Hyx->Amp[x];

    psi = decay * psi + amp * dEz / Configuration->Dx;

    hy = hy - db * Configuration->Dx * psi;

    Hyx->Psi[hyOffset] = psi;
  }

  Hy->Data[hyOffset] = hy;
}
```

Listing 4.6. 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.

4.2.3. Visualizer

4.3. Modeling approach

4.4. Implementation

4.5. Testing methodology

4.5.1. Test Model

4.5.2. Analytical Result

4.5.3. Numerical Result

4.5.4. Comparison

4.6. Additional Examples

4.6.1. Coupler

4.6.2. Splitter

Chapter 5

Conclusions

5.1. Meep performance

5.2. GoLightly performancec

5.3. Meep vs GoLightly

5.4. Results

5.5. Limitations

Chapter 6

FUTURE WORK

future work...

REFERENCES

[1] 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.