Finite Difference Time domain method on Cuda

A Project Report

submitted by

BHARATH M R

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY
and
MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2013

THESIS CERTIFICATE

This is to certify that the thesis titled **Finite Difference Time domain method on Cuda**, submitted by **Bharath M R**, to the Indian Institute of Technology, Madras, for the award of the dual degree of **Bachelor of Technology** and **Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Harishankar Ramachandran
Research Guide
Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 4th June 2013

ACKNOWLEDGEMENTS

I would like to thank my guide Prof. Harishankar Ramachandran for providing me guidance throughout the project and also giving me a lot of freedom to explore lot of ideas. I would like to thank Prof. Ananth Krishnan for all the insights into FDTD provided through the course Computational Electromagnetics. I am very lucky to have spent my last semester with my friends Bhargava, Sudharshan, Vignesh, Abishek, Siddharth, Numaan, and Prasad. The discussions were a lot of fun. I would also like to thank my wingmates for coming up with great ideas and making my insti life more enjoyable. Lastly I would like to thank my parents for their support throughout my life. They provided me with the best education possible and were ready to take a lot of risk for providing me with better education.

ABSTRACT

KEYWORDS: Maxwell's equations, Finite Difference Time Domain method, Graphical Processing units, Cuda, hardware acceleration

Numerical Solutions to Maxwell's equations are important in the case of photonics for verifying results before fabrication of devices. The Finite Difference Time Domain Method which models the Maxwell's equation as difference equations provides an accurate method for simulation of such problems. FDTD algorithms are very simple, but are also computationally intensive. The simple nature of the algorithm makes it highly parallelizable. Due to the parallel nature of the algorithm, an implementation of the algorithm on an GPU gives a large speedup. Various methods to achieve this speed up is explored during the project. A perfectly matched layer is also implemented. The program has the ability to handle dispersive and gain materials. .The speedup obtained is 15X on a capable GPU.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CUDA	Compute Unified Device Architecture
GPU	Graphical Processing Unit.
FDTD	Finite Difference Time Domain
PML	Perfectly matched layer

NOTATION

$\mathbf{E}$	Electric Field vector.
$\mathbf{H}$	Magnetic Field vector.
E_x, E_y, E_z	Electric field components in different cartesian directions.
H_x, H_y, H_z	Magnetic field components in different cartesian directions.
ε	Electrical permittivity
μ	Magnetic Permeability
σ	Electric conductivity
σ^*	Magnetic conductivity

CHAPTER 1

Introduction

FDTD(Finite Difference Time Domain) is a set of numerical methods for solving Maxwell's equations. FDTD methods are generally very simple and computationally intensive. FDTD does not involve linear algebra, which makes it a great target for parallelization. FDTD methods are powerful as they allows us to look at transient behaviours in materials, and can model dispersion and non - linear materials very effectively.

1.1 Motivation

The integrated optoelectronics group of IIT Madras uses commercial softwares for simulation before fabrication. The simulations of some problems can take large amounts of time to run. The motivation was to provide a open source alternative to the existing commercial softwares, which is flexible and highly extensible.

There are very few open source alternatives, the best one being MEEP [2] which is an implementation in C and uses OpenMP for execution in a network of nodes. There is a open source implementation called BCALM [5], which can simulate most of what MEEP does, on a GPU but suffers from a lot of divergence issues. The motivation was to develop an alternative open source project which is as powerful as MEEP but runs faster due to hardware acceleration.

1.2 Organization

In Chapter 2, the execution model of Cuda is explained. The memory hierarchy in a GPU and the various advantages they have one over the other is explained in this chapter.

Chapter 3 deals with the discretization of Maxwell's equation and reducing the problem into two dimensions.

Chapter 4 deals with the description and implementation of the 2 dimensional FDTD algorithm. It explaines various optimizations used, and explains why the optimizations were used. It also explores various methods which did not provide a speed up.

Chapter 4 deals with the implementation of a perfectly matched absorbing layer for handling boundary conditions.

Chapter 5 deals with materials which have their properties varying as a function of the frequency of the travelling wave. It explores various ways to adapt the FDTD algorithm so that it can simulate these materials.

Chapter 6 deals with further ideas that can be implemented to improve the project.

CHAPTER 2

Introduction to the GPU Architecture

2.1 Introduction to a GPU

Graphical processor units were initially special purpose parallel processors which were used to render graphics, which is inherently a highly parallel task. As the transistor budget for these GPU's increased, new functional units like the floating point units were added to these processors. The advantage of parallelism in these graphical processors was recognized by scientific researchers and they used the built in API to run large scientific simulations. Nvidia came up with a better execution model for these processors which made it easier to run general purpose programs on these graphical units.

2.2 Cuda Execution Model

NVidia's parallel programming model is known as Compute Unified Device Architecture. The architecture exploits parallelism by providing a virtual pool of infinite threads, which are then dynamically scheduled and run on the processors. The Cuda execution model allows us to separate the program execution into a series of serial code (runs on the cpu) and parallel code (runs on the gpu). The parallel code that is run on the gpu is called a kernel.

A kernel is executed by a grid of thread blocks. A thread block is a group of threads that share the same resources in one of the streaming processors. The threads in a block can communicate with other threads using the shared memory. All the threads in a block start at the same instruction address. If there is an divergence(a if statement) in the threads in a block, all the threads both take the divergent and the normal path.

A grid is made up of multiple blocks. Each block execution is completely independent of the other block. Hence the these blocks can be scheduled and executed inde-

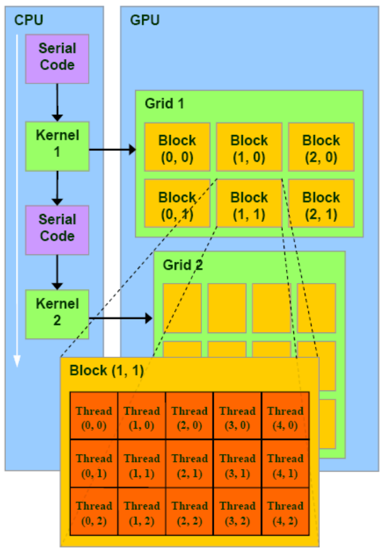

Figure 2.1: Execution model of a Cuda Program

pendent of each other. This allows the Cuda architecture to scale on different graphical processors. If the graphical processor has more number of streaming processors, then the more blocks can be executed making the process faster.

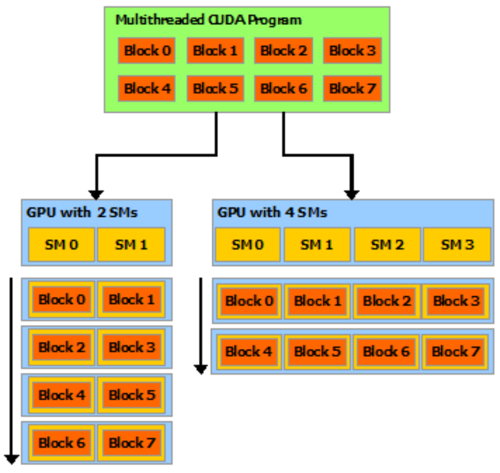

Figure 2.2: Execution model of a Cuda Program

2.2.1 Memory Model in Cuda

It is important to understand the memory model of a GPU before running a particular program. GPU's have a collection of different memories, each optimized for a special purpose. The memory model of Cuda consists of 5 types of memories.

1. Registers

2. Shared Memory
3. Global Memory
4. Constant Memory
5. Texture Memory

Registers

These are the CPU equivalent of register file. The registers are private to each thread. If the number of registers used in a program is large, then the extra registers are stored in the DRAM, slowing the process.

Shared Memory

Shared memory is used to communicate between threads. Shared memory also acts as an scratch pad where you store temporary results. The shared memory can be accessed by all the threads in a block. Shared memory is on chip memory and has the lowest latency. The shared memory behaves like a user managed cache.

Constant Memory

Each device has 64KB of constant memory. The constant memory resides in the global memory and it is cached. Constant memory reads can be broadcasted to all the threads in a half warp. Hence it results in a single access for half a warp of threads.

Texture Memory

Texture memory is specially implemented for use in graphical applications. Texture memory is read only. But the texture memory cache is optimized for 2D spatial locality.(Generally caches are optimized for 1D spatial locality).

Global Memory

Global memory is the non-local memory of the device. Every thread can read and write to the global memory.

CHAPTER 3

Introduction to the Finite Difference Time Domain method

The finite difference time method use finite difference as approximations to both spatial and temporal derivatives that appear in Maxwell's equations. This chapter discusses the discretization of the Maxwell's equations using the finite difference method.

3.1 Maxwell's Equation

Faraday's Law:

$$\frac{d\mathbf{B}}{dt} = -\nabla \times \mathbf{E} - \mathbf{J}_{M,source} - \sigma^* \mathbf{H} \tag{3.1}$$

Ampere's Law

$$\frac{d\mathbf{E}}{dt} = \nabla \times \mathbf{H} - \mathbf{J}_{E,source} - \sigma \mathbf{E} \tag{3.2}$$

Gauss' Law:

$$\nabla . \mathbf{D} = 0 \tag{3.3}$$

$$\nabla . \mathbf{B} = 0 \tag{3.4}$$

In the above equations, the magnetic current density and magnetic conductivity are not physically relevant, but are useful in implementing numerical techniques to avoid reflection from boundaries.

When we discretize the above equations, we would want our algorithm to implicitly handle the gauss' laws. We split the curl equations along 3 orthogonal axes to realize the following equations.

$$\frac{dH_x}{dt} = \frac{1}{\mu}\left[\frac{dE_y}{dz} - \frac{dE_z}{dy} - (J_{M,source_x} + \sigma^* H_x)\right] \tag{3.5}$$

$$\frac{dH_y}{dt} = \frac{1}{\mu}\left[\frac{dE_z}{dz} - \frac{dE_x}{dy} - \left(J_{M,source_y} + \sigma^* H_y\right)\right] \tag{3.6}$$

$$\frac{dH_z}{dt} = \frac{1}{\mu}\left[\frac{dE_x}{dz} - \frac{dE_y}{dy} - (J_{M,source_z} + \sigma^* H_z)\right] \tag{3.7}$$

$$\frac{dE_x}{dt} = \frac{1}{\mu}\left[\frac{dH_y}{dz} - \frac{dH_z}{dy} - (J_{M,source_x} + \sigma E_x)\right] \tag{3.8}$$

$$\frac{dE_y}{dt} = \frac{1}{\mu}\left[\frac{dH_z}{dz} - \frac{dH_x}{dy} - \left(J_{M,source_y} + \sigma E_y\right)\right] \tag{3.9}$$

$$\frac{dE_z}{dt} = \frac{1}{\mu}\left[\frac{dH_x}{dz} - \frac{dH_y}{dy} - (J_{M,source_z} + \sigma E_z)\right] \tag{3.10}$$

3.2 Yee's Algorithm

The FDTD method employs finite differences as an approximation to the temporal and spatial derivatives that appear in the Maxwell's equation. It uses central differences as the error in central difference is of the order Δ^2, while both right and left differences results in an error of the order Δ.

The FDTD method as proposed by Kane Yee in 1966 employs the second order central differences and staggered field structure to implement the finite difference scheme. In the spatial grid, the magnetic field is defined at every half step coordinate while the elctric fields are defined at every full step coordinate. The algorithm is summarized as follows:

1. Replace all the derivatives in Ampere's and Faraday's laws with finite differences. Discretize space and time so that they are staggered in both space and time.
2. Solve the discretized equations to obtain the update equations.

3. Evaluate the magnetic fields using the staggered electric field obtained before.
4. Evaluate the electric fields using the staggered magnetic field obtained in Step 3.

It can be proved that the following arrangement satisfies the integral form of gauss laws due to its inherent staggered structure. [4].

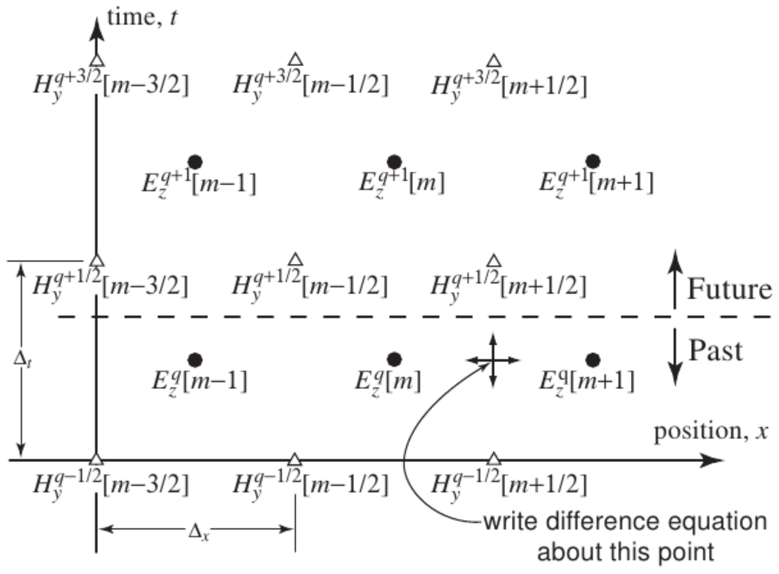

Figure 3.1: The arrangement of electric and magnetic field in space and time. The electric field are represented by circles and magnetic field is represented by triangles.

3.3 Reduction to lower dimensions

The curl's equation defined from 3.5 to 3.10 can be reduced into two dimensions by assuming there is no change in field along a particular direction. The update equations above had fields along each directions, hence resulting in 6 equations. We can reduce it to two dimensions by considering either the magnetic field of the electric field orthogonal to the plane of propagation i.e TM or TE polarization, respectively. This results in three update equations for each of the modes.

TM Mode:

$$\frac{dH_x}{dt} = \frac{1}{\mu}\left[-\frac{dE_z}{dy}\right] \tag{3.11}$$

$$\frac{dH_y}{dt} = \frac{1}{\mu}\left[\frac{dE_z}{dx}\right] \tag{3.12}$$

$$\frac{dE_z}{dt} = \frac{1}{\varepsilon}\left[\frac{dH_y}{dx} - \frac{dH_x}{dy} - (J_{E,source} + \sigma E_z)\right] \tag{3.13}$$

TE Mode:

$$\frac{dE_x}{dt} = \frac{1}{\varepsilon}\left[\frac{dH_z}{dy} - (J_{E,source_x} + \sigma E_x)\right] \tag{3.14}$$

$$\frac{dE_y}{dt} = \frac{1}{\varepsilon}\left[-\frac{dH_z}{dx} - \left(J_{E,source_y} + \sigma E_y\right)\right] \tag{3.15}$$

$$\frac{dH_z}{dt} = \frac{1}{\mu}\left[\frac{dE_x}{dy} - \frac{dE_y}{dx}\right] \tag{3.16}$$

The project primarily concentrates on the TM mode of propogation and implements most of the algorithms in TM mode.

CHAPTER 4

TM Mode implementation in Cuda

In this chapter, we are going to explore the design and the various optimizations in implementing the FDTD algorithm on a GPU.

The following notations will be used when we are discretizing the equations.

$$H_x(x,y,t) = H_x(m\Delta_x, n\Delta_y, q\Delta_t) = H_x^q[m,n]$$

$$H_y(x,y,t) = H_y(m\Delta_x, n\Delta_y, q\Delta_t) = H_y^q[m,n]$$

$$E_z(x,y,t) = E_z(m\Delta_x, n\Delta_y, q\Delta_t) = E_z^q[m,n]$$

The index m corresponds to a step in the x direction. The index n corresponds to a step in the y direction. q represens the temporal time step. The step sizes along the x and the y direction are represented by Δ_x and Δ_y respectively.

We discretize the TM mode equations to arrive at the following update equations.

$$H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}\right] = C_{hxh} H_x^{q-\frac{1}{2}}\left[m, n+\frac{1}{2}\right] - C_{hxe}\left(E_z^q[m,n+1] - E_z^q[m,n]\right) \quad (4.1)$$

$$H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n\right] = C_{hyh} H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}, n\right] + C_{hye}\left(E_z^q[m+1,n] - E_z^q[m,n]\right) \quad (4.2)$$

$$E_z^{q+1}[m,n] = C_{eze} E_z^q[m,n] + C_{ezh}\left\{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n\right] - H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2}, n\right]\right\}$$
$$\left\{H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}\right] - H_x^{q+\frac{1}{2}}\left[m, n-\frac{1}{2}\right]\right\} \quad (4.3)$$

where

$$C_{hxh}\left[m, n+\frac{1}{2}\right] = \left.\frac{1-\frac{\sigma_m \Delta t}{2\mu}}{1+\frac{\sigma_m \Delta t}{2\mu}}\right|_{m\Delta_x,(n+\frac{1}{2})\Delta y} \tag{4.4}$$

$$C_{hxe}\left[m, n+\frac{1}{2}\right] = \left.\frac{1}{1+\frac{\sigma_m \Delta t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{m\Delta_x,(n+\frac{1}{2})\Delta y} \tag{4.5}$$

$$C_{hyh}\left[m+\frac{1}{2}, n\right] = \left.\frac{1-\frac{\sigma_m \Delta t}{2\mu}}{1+\frac{\sigma_m \Delta t}{2\mu}}\right|_{(m+\frac{1}{2})\Delta_x,n\Delta y} \tag{4.6}$$

$$C_{hye}\left[m+\frac{1}{2}, n\right] = \left.\frac{1}{1+\frac{\sigma_m \Delta t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{(m+\frac{1}{2})\Delta_x,n\Delta y} \tag{4.7}$$

$$C_{eze}\left[m, n\right] = \left.\frac{1-\frac{\sigma\Delta_t}{2\varepsilon}}{1+\frac{\sigma\Delta_t}{2\varepsilon}}\right|_{m\Delta_x,n\Delta_y} \tag{4.8}$$

$$C_{ezh}\left[m, n\right] = \left.\frac{1}{1+\frac{\sigma\Delta_t}{2\varepsilon}}\frac{\Delta_t}{\varepsilon\delta}\right|_{m\Delta_x,n\Delta_y} \tag{4.9}$$

We look at how the above update equations are translated to Cuda code. The first step in starting the simulation is memory allocation. The constants μ, ε, σ, σ_m are read as input for the program and hence they allocated memory on the host. A linear memory is used, instead of an array of pointer to pointers, the reason for which will be explained. We need to calculate the coefficients from these constants. Calculating these constants is highly parallelizable. Also, once the coefficients are calculated, the constants can be deallocated, as they are not required. The first kernel that runs on the device is the kernel to calculate these coefficients. We need to decide how to split the whole array into blocks and threads. To see the performance, of the various divisions of blocks and threads, we run the kernel with different block and thread sizes and look at the time taken to execute the kernel.

Block Size	Time taken for 1024 x 1024 block(in ms)
32 x 32	2.83
256 x 1	2.36
128 x 16	3.24
128x8	3.23

Table 4.1: Time taken by different block sizes.

It is important to look at how the global memory is cached in order to understand what is a good way to split our array into blocks and threads. The single most important thing that affects performance in cuda applications is memory coalescing. Global memory loads and stores can be coalesced into a single load or store instruction, if the memory is coalesced properly. In Fermi architecture GPU's all memory accesses are cached through a L1 cache which are 128-byte lines. Hence a read instruction results in a read of 128 bytes. If all the bytes read in a single instruction are not used, then it results in bad performance.

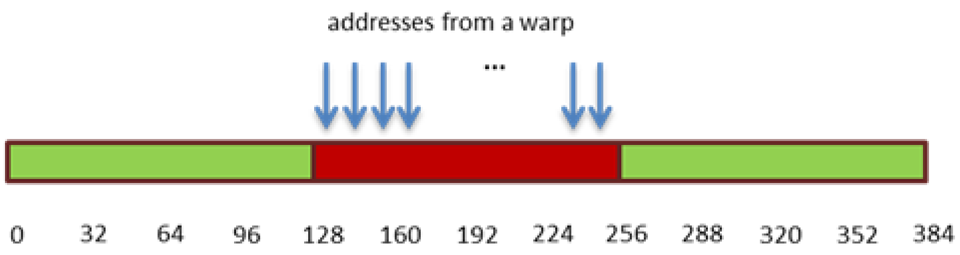

Figure 4.1: Access pattern which results in a single 128- byte transaction, indicated by the red rectangle. Source: [1]

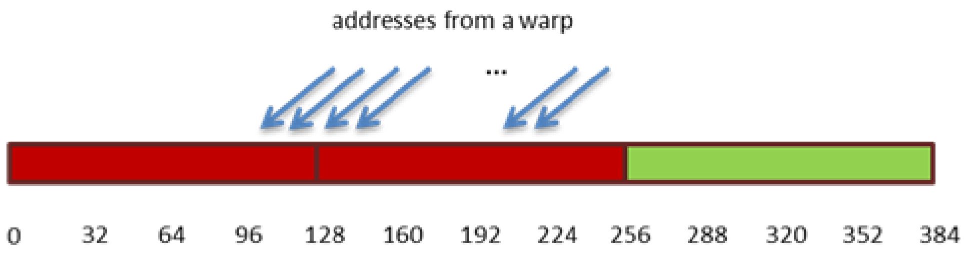

Figure 4.2: Access pattern which results in a two 128- byte transaction, indicated by the red rectangle, because of bad access pattern. Source: [1]

We will look at the kernel to update the field H_x.

```
__global__ void update_Hx(float *Hx, float *Ez,
                        float *coef1, float* coef2){
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * pitch / sizeof(float);
    int top = offset + pitch / sizeof(float);
    if(y < y_index_dim - 1)
        Hx[offset] = coef1[offset] * Hx[offset]
                   - coef2[offset] * (Ez[top] - Ez[offset]);
    __syncthreads();
}
```

Listing 4.1: update H_x kernel

The function `update_Hx` runs the kernel for the updating the x component of magnetic field. It is defined as `__global__` because its a device function that can be called by the program on the host. The structure `threadIdx` contains information about the identity of the thread in a two dimensional block. To obtain the actual x position of the value we are trying to calculate, we use the inbuilt structure values and obtain the x position using line 3. The y position is calculated similarly. As our field arrays and coefficient arrays are linear, we need to translate the obtained x and y position to the offset in the linear array. Line 5 calculates the required offset. We will discuss why we use the value `pitch` in 4.0.1. We also need access to the next value along the y direction. Line 6 calculates this value. The next few lines implement the update equations. The statement `__syncthreads()` is important because reads and writes in Cuda are non - blocking i.e. the execution of a line does not guarantee the completion of a write operation. Syncthreads waits for all the threads to synchronize and finish their write operation. This is very essential because the update on the electric fields has to happen after the update on the magnetic fields are complete.

4.0.1 Memory Coalescing visited again

The performance of a kernel goes down if a warp tries to access memory which does not start at a multiple of 128 bytes. This is illustrated in the following figure.

If the dimensions of our field array is not a multiple of 32(128 bytes), then we will have a large number of non - coalesced accesses when we try to access the second row of our grid array. Hence when the linear array is being allocated, we need to pad the array with unused memory, so that every row starts at a multiple of 128 bytes. Initially, this was done manually. But Cuda has an function `cudaMallocPitch()` which pads the linear array and returns the size_t variable `pitch` which represents the size of a memory padded row in bytes. The pitch used in the update equations is the value returned by `cudaMallocPitch`. It is used to calculate the offset in the program.

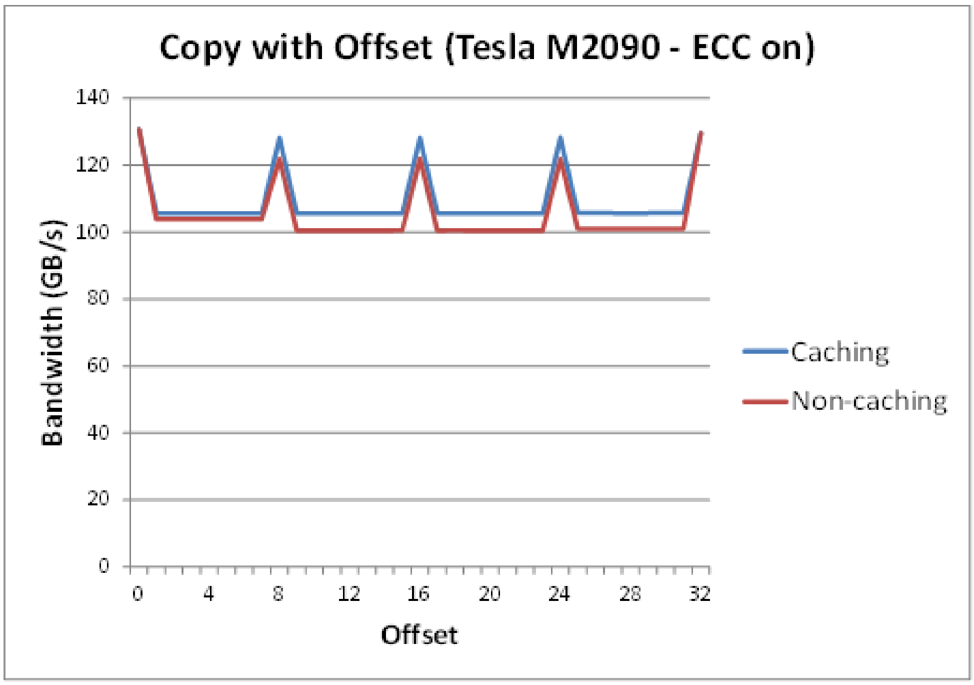

Figure 4.3: Memory bandwidth as a function of offset from a multiple of 128 bytes. Source: [1]

Another thing to consider with respect to memory coalescing is the thread structure in a block. While updating H_x, we have an access made to a node and node + offset. The threads should contain as many rows as possible, as it avoids reading a memory twice, as the read memory will be cached. But the columns should be sufficient so that we are making atleast 128 byte access. Hence a block should contain 32 x 16 threads.

4.0.2 H_y update

The H_y is similar to H_x updates except for the fact that it accesses the next value to the present index to update the H_y value. This introduces a memory coalescing problem as every kernel reads a single value outside of the 128 byte block. The only way to avoid this problem, is to hide the extra read, by increasing the column size of the block. Hence the size of the block used is 256×1.

```
__global__ void update_Hy(float *Hy, float *Ez,
                          float * coef1, float * coef2){

    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * pitch / sizeof(float);
    int right = offset + 1;
    if(x < x_index_dim -1)
        Hy[offset] = coef1[offset] * Hy[offset] +
                     coef2[offset] * (Ez[right] - Ez[offset]);
    __syncthreads();
}
```

Listing 4.2: update H_y kernel

4.0.3 E_z update

The E_z update is similar to the update_Hx and update_Hy, except for the fact both the $ix - 1$ and the $iy - 1$ values are used, where ix is the x index of the E_z array and iy is the y index of the E_z array. The kernel was run for different block sizes and the best size for the kernel block was 256×1.

```
__global__ void update_Ez(float *Hx, float *Hy,
                          float *Ez, float * coef1,
                          float *coef2){
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * pitch / sizeof(float);

    int left = offset - 1;
    int bottom = offset - pitch / sizeof(float);

    if (x > 0 && y > 0 &&
        x<x_index_dim - 1 && y < y_index_dim - 1){
        Ez[offset] = coef1[offset] * Ez[offset] +
                     coef2[offset] * ((Hy[offset] - Hy[left])
                     - (Hx[offset] - Hx[bottom]));
    }

    __syncthreads();
}
```

Listing 4.3: update E_z kernel

4.0.4 Other Ideas

There is an access to the dimensions `x_index_dim` and `y_index_dim` in each of the kernels. Loading these values for each kernel onto shared memory every time the kernel is run results in a single read to global memory in each kernel. The reads for these values should be fast and they should be cached. Anothere important property of these values is that once they are dynamically allocated, they are constant. Hence, constant memory makes a very good place to store these variables. Constant memory is cached and a read to constant memory can be broadcasted to all the threads in the warp.

4.1 Open GL Implementation

The above equations take care of calculating the updated values, but we need a way to visualize these values. This section introduces the use of OpenGL to handle the animations and the process to convert the floating point values to colors.

In order to visualize the floating point values, we have to convert it into a set of colors. The floating point values should be normalized between -1 and 1 for the color conversion to work. We take the maximum of the whole array and divide the whole array by the maximum. We arrive at a set of floating point value between -1 and 1. We use a kernel `float_to_color` to make this conversion. The floating point values are initially converted to HSL color scheme The luminosity value is set to the floating point value. The saturation is always 1. The HSL values are then converted to RGB values. The following program converts the values to a RGB array.

```
__global__ void float_to_color( unsigned char *optr,
                              const float *outSrc ) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * pitch / sizeof(float);

    float l = outSrc[offset];
    float s = 1;
    int h = (180 + (int)(360.0f * outSrc[offset])) % 360;
    float m1, m2;

    if (l <= 0.5f)
        m2 = l * (1 + s);
    else
        m2 = l + s - l * s;
    m1 = 2 * l - m2;

    optr[offset*4 + 0] = value( m1, m2, h+120 );
    optr[offset*4 + 1] = value( m1, m2, h );
    optr[offset*4 + 2] = value( m1, m2, h -120 );
    optr[offset*4 + 3] = 255;
}

__device__ unsigned char value( float n1, float n2,
                                int hue ) {
    if (hue > 360)      hue -= 360;
    else if (hue < 0)   hue += 360;

    if (hue < 60)
        return (unsigned char)(255 *
                    (n1 + (n2-n1)*hue/60));
    if (hue < 180)
        return (unsigned char)(255 * n2);
    if (hue < 240)
        return (unsigned char)(255 * (n1 +
                                    (n2-n1)*(240-hue)/60));
    return (unsigned char)(255 * n1);
}
```

Listing 4.4: Color Conversion kernel

The RGB values is stored as a bitmap and is rendered using openGL.

4.2 Storing Simulation Results

The openGL implementation provides a nice way of visualizing the information but it does not provides access to the simulation results once the simulation is over. There should be a way to store these simulation results once the simulation is done. The brute force way is to store these values as an comma seperated value file. But this requires a lot of memory as the values are stored as ascii characters and the size of the files for a 1024 X 1024 array is around 31Mb. The alternative was to store it as bytes but have a way to read these files. Hence the hierarchical data format (HDF5) was chosen to store the array data. HDF5 is a specially designed file format used to store large amounts of numerical data. The file format has extensive support in both python and matlab, hence its very easy to analyze results using python or matlab. Also, hdf5 comes with tools which converts the whole array into a bitmap, and hence visualization is a direct result of storing the arrays in this format. The HDF5 file format access to array values is also faster than accessing the values in a database.

4.3 Sources

Most of the simulations run with different kind of sources like a line source, point source etc. Also, the sources can be constant, sinusoid or gaussian. In the FDTD program, the sources are always point sources. A line sources can be implemented in terms of point sources.

Initially each source was implemented as a structure of following entities.

- **source type**: Can be constant, sinusoid or gaussian.
- **x position**
- **y position**
- **mean**: Mean acts as the frequency in the case of sinusoid signal. Mean is set to zero in the case of constant signal. In the case of Gaussian signal, it represents the center frequency.
- **variance**: This represents the variance of the gaussian signal. In the case of constant and sinusoid sources, it represents the amplitude of the signal.

The sources were initially represented as a array of structure. The kernel which initialized the source values became the longest running kernel when the number of structures became greater than 30. Hence a different way of representing sources was explored.

4.3.1 Array of structures vs Structure of arrays

An array of structures behaves like row major access. Hence an access to variable in a structure results in strided access, which has bad memory coalescing.

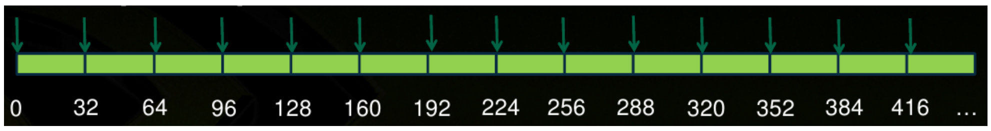

Figure 4.4: An access to structure[i].x results in strided access Source: [1]

Structure of arrays results in better memory coalescing when you are accessing an element of the structure.

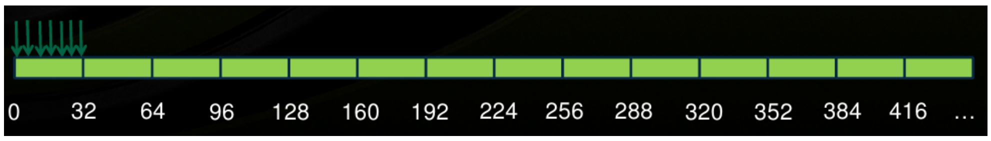

Figure 4.5: An access to structure[i].x results in coalesced access Source: [1]

4.4 Failed Ideas

This section deals with implementations which are supposed to work better than the basic implementation, but actually perform worse, or equivalent to the above implementation.

4.4.1 Using shared memory

Since every thread in the kernel accesses the next array value, either in the x direction or the y direction, it results in N redundant reads. Hence its a good idea to read the values in a kernel to the shared memory and then apply the algorthm on it. The first implementation was to ignore the value at the boundary and load the major values in the blocks to the shared memory. We access the boundary values from the global memory as they are accessed only once.

The initial implementation resulted in bank conflicts. Shared memory stores its values in a series of memory banks which can be accessed in parallel. A column major implementation results in access of values from the same memory bank while a row major implementation causes no bank conflicts. The bank conflict problem was resolved using row major access implementation, but direct accesses to global memory performs better than a shared memory implementation.

The next implementation involved reading the boundary values to shared memory. This had to be done in such a way that there is no bank conflict. The implementation performed equivalent to that of reading directly from global memory. Shared memory is great when there is a lot of calculations on the memory being read or there is repeated access to the memory being read.

Hence to increase the ratio of number of calculations to the number of memory access, shared memory was used to load memory, so that more than one iteration of the fdtd algorithm can be implemented in a single call to the kernel. The memory load for such a kernel is illustrated in figure 4.7. The kernels were tested with the number of iterations varying from 2 to 5. This did not increase the performance.

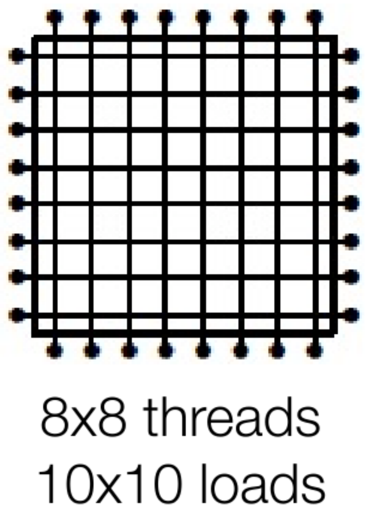

Figure 4.6: Shared memory access of the block and its boundaries

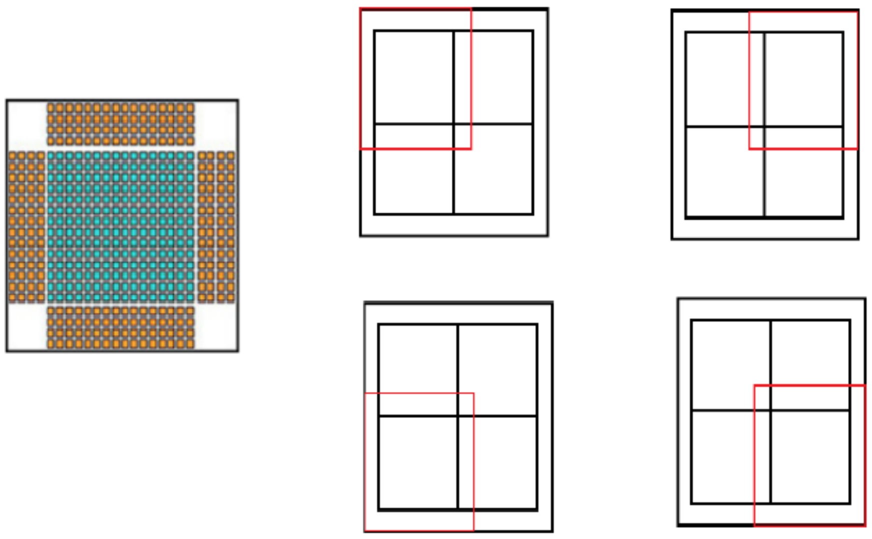

Figure 4.7: Loads to shared memory with fdtd algorithm running for more than one iteration

4.4.2 Using Texture memory

Texture memory is used in graphical applications to render the bitmap images. The advantage of texture memory is that the cache is optimized for 2 dimensional access. But texture memory is read only. The architecture does not really have a distinction between texture memory and global memory except for the fact that both use different types of caches. So an implementation which read from the texture memory but wrote into the global memory was implemented. There were two problems with this approach. Texture memory did not support a parameter pitch which would be used to access a element in the array. Even though the cache was optimizing for 2D access, there was no coalesced memory reads. The other problem was texture memory is to be used a read only memory. Constant change in the values, hindered the caching scheme of the texture memory..

This chapter summarized most of the ideas used to speed up the cuda implementation. We now extend these ideas to other modes.

4.5 Performance Comparisions

To compare the speedup obtained by implementing in a gpu, two different implementations of CPU code was used and the implementations were tested on two different GPUS. The first cpu implementation was a naive implementation of the difference method for FDTD. The second implementation involved boxed calculations so that the there is better usage of cache. The implementations were tested for 100 iterations of different grid structures.

Block sizes	CPU without -O3	CPU with -O3	GPU (Geforce GT 520)	GPU(GeForce GTX 560 Ti)
256	0.24	0.20	0.06	0.0076
512	0.97	0.26	0.248	0.025
800	2.4	0.74	0.674	0.063
1024	3.95	1.59	0.98	0.094

Table 4.2: Execution times for different implementations in seconds. The CPU tests were run on a Intel Core 2 Duo running at 3.16Ghz. GT 520 has 48 cuda cores running at 1.62Ghz. The GTX 560 Ti has 384 Cuda cores running at 1.66GHz

The speedup obtained on a gpu is about 17X.

CHAPTER 5

Absorbing boundary conditions

In most of the simulations, we simulate only a part of the region we are interested in and do not care about what the fields do once they are out of our region of interest. We need to mathematically devise methods so that there is no reflection from the boundaries when these fields hit them. One way to address the problem is to have large enough simulation region so that the reflected fields do not affect the simulation region we are interested in, during the time of the simulation. We explore one of the ways to implement boundary conditions, such that there is very little reflection from the boundaries.

A Berenger split field perfectly matched layer was implemented in order to handle boundary conditions. Berenger's PML provides an effective way to implement absorbing boundary conditions by splitting each field into two orthogonal components. The Maxwell's curl equations are also split and by choosing the right parameters, we can achieve a good perfectly matched layer.

When a electromagnetic way travelling in a region of permittivity $\varepsilon_{region1}$ and permeability $\mu_{region1}$ is normally incident on another region of permittivity $\varepsilon_{region2}$ and permeability $\mu_{region2}$ with electric conductivity σ and magnetic conductivity of σ^*, the reflection coefficient is given by

$$\Gamma = \frac{\eta_{region1} - \eta_{region2}}{\eta_{region1} + \eta_{region2}} \tag{5.1}$$

where:

$$\eta_{region1} = \sqrt{\frac{\mu_{region1}}{\varepsilon_{region1}}} \tag{5.2}$$

$$\eta_{region2} = \sqrt{\frac{\mu_{region2}\left(1 + \frac{\sigma^*}{j\omega\mu_{region2}}\right)}{\varepsilon_{region2}\left(1 + \frac{\sigma}{j\omega\varepsilon_{region2}}\right)}} \tag{5.3}$$

There won't be any reflection if

$$\mu_{region1} = \mu_{region2} \tag{5.4}$$

$$\varepsilon_{region1} = \varepsilon_{region2} \tag{5.5}$$

$$\frac{\sigma^*}{\mu_{region2}} = \frac{\sigma}{\varepsilon_{region2}} \tag{5.6}$$

But the above equations are valid only for normal incidence. Berenger PML provides a way to split a field into two orthogonal fields and allows us to apply the above equations in the orthogonal directions.

Berenger PML guarantees perfect transmission in the case of continuous media. Discretization of the media results in a lot of reflection if there is step discontinuity in conductivity. To reduce this reflection error we have to grade the conductivity in the PML layer from zero to a higher value.

$$\mu\frac{\partial H_x}{\partial t} + \sigma_y^* H_x = -\frac{\partial E_z}{\partial y} \tag{5.7}$$

$$\mu\frac{\partial H_y}{\partial t} + \sigma_x^* H_x = \frac{\partial E_z}{\partial z} \tag{5.8}$$

$$\varepsilon\frac{\partial E_{zx}}{\partial t} + \sigma_x E_{zx} = \frac{\partial H_y}{\partial x} \tag{5.9}$$

$$\varepsilon\frac{\partial E_{zy}}{\partial t} + \sigma_y E_{zy} = -\frac{\partial H_x}{\partial y} \tag{5.10}$$

In our simulation we use the results mentioned in [4] to grade our PML region using polynomial grading. The equation for polynomial grading is given by

$$\sigma(x) = \left(\frac{x}{d}\right)^m \sigma_{max} \tag{5.11}$$

where d is the thickness of the PML.

The reflection for such a graded PML is given by

$$R(\theta) = e^{2\eta\sigma_{max} d cos(\theta)/(m+1)} \tag{5.12}$$

For a desired reflection error, we can calculate σ_{max}

$$\sigma_{max} = -\frac{(m+1)ln(R(0))}{2\eta d} \tag{5.13}$$

We use a polynomial grading of order 3 in the polynomial graded PML implemented in the program. The pml layer is calculated using a python program which pads the provided structure with the pml layer. The padded structure is used by the Cuda program for the simulation.

CHAPTER 6

Dispersive, Non - Linear Material Simulation

Dispersive materials are materials, whose permeability and permittivity vary with the frequency of the waves travelling through them. In dispersive materials, different frequencies of the wave travel at different velocities. It is difficult to model the material properties at all frequencies. We model the properties of these materials piecewise in frequencies. Each region of interest is modeled as a single pole or two pole system. The assumption we make is, the region are far enough such that a pole in one region does not affect significantly the frequency response in the other region.

There are three different models in which the dependencies on frequency is modeled on

- Debye Media
- Lorentz Media
- Drude Media

6.1 Debye Media

Debye media are characterized by the susceptibility function which has a single real pole. For a single pole Debye media we have

$$\chi_p = \frac{\Delta\varepsilon_p}{1 + j\omega\tau_p} \tag{6.1}$$

The time domain response for Debye media is

$$\chi_p(t) = \frac{\Delta\varepsilon_p}{\tau_p} e^{\frac{-t}{\tau_p}} U(t) \tag{6.2}$$

At any particular **E** observation point, Ampere's law can be expressed as

$$\nabla \times \mathbf{H} = \varepsilon_0 \varepsilon_\infty \frac{\partial \mathbf{E}}{\partial t} + \sigma \mathbf{E} + \sum_{p=0}^{P} \mathbf{J_p} \tag{6.3}$$

where $\mathbf{J_p}$ is the polarization current associated with the p^{th} pole, The phasor polarization current associated with the each pole is given by

$$\mathbf{J_p} = \varepsilon_0 \Delta \varepsilon_p \left[\frac{j\omega}{1 + j\omega\tau_p} \right] \mathbf{E} \tag{6.4}$$

The update equations have an addition update variable J. The update equation for electric field is modified to include the polarization current.

$$\mathbf{J_p^{n+1}} = k_p J_p^n + \beta_p \left[\frac{\mathbf{E}^{n+1} - \mathbf{E}^n}{\Delta t} \right] \tag{6.5}$$

where:

$$k_p = \frac{1 - \frac{\Delta t}{2\tau_p}}{1 + \frac{\Delta t}{2\tau_p}} \tag{6.6}$$

$$\beta_p = \frac{\varepsilon_0 \Delta \varepsilon_p \frac{\Delta t}{\tau_p}}{1 + \frac{\Delta t}{2\tau_p}} \tag{6.7}$$

The electric field update equation is as follows.

$$\mathbf{E}^{n+1} = C_1 \mathbf{E^n} + C_2 \left[\nabla \times \mathbf{H^{n+\frac{1}{2}}} - \frac{1}{2} \sum_{p=0}^{P} (1 + k_p) \mathbf{J_p^n} \right] \tag{6.8}$$

where

$$C_1 = \left[\frac{2\varepsilon_0\varepsilon_\infty + \sum\limits_{p=0}^{P} \beta_p - \sigma\Delta t}{2\varepsilon_0\varepsilon_\infty + \sum\limits_{p=0}^{P} \beta_p - \sigma\Delta t} \right] \tag{6.9}$$

$$C_2 = \left[\frac{2\Delta t}{2\varepsilon_0\varepsilon_\infty + \sum\limits_{p=0}^{P} \beta_p - \sigma\Delta t} \right] \tag{6.10}$$

The magnetic field updates remain the same.

6.2 Lorentz Media

Lorentz media are generally modelled by two poles in their susceptibility. The permittivity in the frequency domain is given by.

$$\varepsilon(\omega) = \varepsilon_\infty + \sum_{p=1}^{P} \frac{\Delta\varepsilon_p \omega_p^2}{\omega_p^2 + 2j\omega\delta_p - \omega^2} \tag{6.11}$$

The polarization current update equations are given by

$$\mathbf{J_p^{n+1}} = \alpha_p \mathbf{J_p^n} + \xi_p \mathbf{J_p^{n-1}} + \gamma_p \left[\frac{\mathbf{E^{n+1}} - \mathbf{E^n}}{2\Delta t} \right] \tag{6.12}$$

where

$$\alpha_p = \frac{2 - \omega_p^2 (\Delta t)^2}{1 + \delta_p \Delta t} \tag{6.13}$$

$$\xi_p = \frac{\delta_p \Delta t - 1}{1 + \delta_p \Delta t} \tag{6.14}$$

$$\gamma_p = \frac{\varepsilon_0 \Delta\varepsilon_p \omega_p^2 (\Delta t)^2}{1 + \delta_p \Delta t} \tag{6.15}$$

The electric field updates are given by

$$\mathbf{E}^{n+1} = C_1\mathbf{E^{n-1}} + C_2\mathbf{E^{n}} \tag{6.16}$$

$$+ C_3\left\{\nabla \times \mathbf{H^{n+\frac{1}{2}}} - \frac{1}{2}\sum_{p=0}^{P}\left[(1+\alpha_p)\mathbf{J_p^n} + \xi_p\mathbf{J_p^{n-1}}\right]\right\} \tag{6.17}$$

where

$$C_1 = \frac{\frac{1}{2}\sum\limits_{p=0}^{P}\gamma_p}{2\varepsilon_0\varepsilon_\infty + \frac{1}{2}\sum\limits_{p=0}^{P}\gamma_p + \sigma\Delta t} \tag{6.18}$$

$$C_2 = \frac{2\varepsilon_0\varepsilon_\infty - \sigma\Delta t}{2\varepsilon_0\varepsilon_\infty + \frac{1}{2}\sum\limits_{p=0}^{P}\gamma_p + \sigma\Delta t} \tag{6.19}$$

$$C_3 = \frac{2\Delta t}{2\varepsilon_0\varepsilon_\infty + \frac{1}{2}\sum\limits_{p=0}^{P}\gamma_p + \sigma\Delta t} \tag{6.20}$$

The coefficients are calculated and cached before and rest of the updates follow a similar pattern as that of TM mode. The kernels implemented for magnetic field updates can be reused.

6.3 Drude Media

The Drude model is used to model electron motion in metal and their influence on the properties of the material. The relative permittivity in drude model in the frequency domain is given by

$$\varepsilon(\omega) = \varepsilon_\infty - \sum_{p=0}^{P}\frac{\omega_p^2}{\omega^2 - j\omega\gamma_p} \tag{6.21}$$

The polarization equation updates are given by

$$\mathbf{J_p^{n+1}} = k_p\mathbf{J_p^n} + \beta_p(\mathbf{E}^{n+1} + \mathbf{E}^n) \tag{6.22}$$

where

$$k_p = \frac{1 - \gamma_p \frac{\Delta t}{2}}{1 + \gamma_p \frac{\Delta t}{2}} \tag{6.23}$$

$$\beta_p = \frac{\omega_p^2 \varepsilon_0 \Delta t/2}{1 + \gamma_p \frac{\Delta t}{2}} \tag{6.24}$$

The electric field updates are given by

$$\mathbf{E}^{n+1} = C_1 \mathbf{E^n} + C_2 \left[\nabla \times \mathbf{H^{n+\frac{1}{2}}} - \frac{1}{2} \sum_{p=0}^{P} (1 + k_p) \mathbf{J_p^n} \right] \tag{6.25}$$

where

$$C_1 = \left[\frac{2\varepsilon_0 \varepsilon_\infty + \sum\limits_{p=0}^{P} \beta_p - \sigma \Delta t}{2\varepsilon_0 \varepsilon_\infty + \sum\limits_{p=0}^{P} \beta_p - \sigma \Delta t} \right] \tag{6.26}$$

$$C_2 = \left[\frac{2\Delta t}{2\varepsilon_0 \varepsilon_\infty + \sum\limits_{p=0}^{P} \beta_p - \sigma \Delta t} \right] \tag{6.27}$$

The magnetic field updates don't change and hence the same kernels can be reused. Separate kernels are used to implement the electric and magnetic field updates.

The number of updates increase with the number of poles present in our model. Hence the time taken by a model increases with the number of update equations.

CHAPTER 7

Further Ideas

This chapter introduces different ideas to extend the project.

7.1 Convolutional PML's

We have implemented a split field PML for all the simulations. Though split field PML's provide us with negligible reflections, split field PML's cannot be used with dispersive or gain materials. Also, split field PML's increase the number of kernels throughout the grid. Convolutional PML's [3],are much powerful than split field PML's. They work with all kinds of FDTD algorithms including dispersive and gain simulations. They also require less number of grid cells to provide the same reflectivity coefficient. Hence an implementation of Convolutional PML's can increase the speed of the program.

7.2 A Domain specific language for structure design

The present program works with a text file which specifies the epsilon, mu and sigma values. Though this allows us to simulate the program, it does not allow us to implement advanced methods for smoothening ladder like structures, optimizing using symmetry of the structure etc. Also, a domain specific language will help us to design a graphical interface in the future. A crude implementation was implemented during the project which is present in the branch `interface`.

7.3 Python Interface

It would be easier to simulate, if the fdtd engine can be wrapped with a python function. This allows us to create the structure in python and run the program on a faster fdtd engine.

APPENDIX A

Using the program

The source of the program is git tracked. There are two different versions of the program, one with OpenGL and the other with saving data through .h5 files. The h5 version is the version that is on master i.e. if you do `git checkout master`, you will be using this version. If you want to use the OpenGL version, you need to do `git checkout interfacenew`. If you want to look at other implementations like, shared memory implementation, then you need to do `git checkout shared`. A tutorial for how to use the program is at a github wiki.

A.1 Generating Documentation

The program has documentation written in the comments. It uses a tool called `doxygen` to convert the documentation in the comments of the source, to formatted html pages. The command used to generate the documentation is `doxygen Doxyfile`. The documentation is generated in the docs folder. The main page of the documentation can be accessed by opening the file `index.html` or running a python server using `python -m SimpleHTTPServer` in th same folder.

REFERENCES

[1] **Nvidia** (2013 –). Cuda toolkit documentation. URL `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`.

[2] **Oskooi, A. F.**, **D. Roundy**, **M. Ibanescu**, **P. Bermel**, **J. D. Joannopoulos**, and **S. G. Johnson** (2010). MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method. *Computer Physics Communications*, **181**, 687–702.

[3] **Roden, J. A.** and **S. D. Gedney** (2000). Convolution PML (CPML): An efficient FDTD implementation of the CFS-PML for arbitrary media. *Microwave and Optical Technology Letters*, **27**(5), 334–339.

[4] **Taflove, A.** and **S. C. Hagness**, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. 2005.

[5] **Wahl, P.**, **D. S. Ly Gagnon**, **C. Debaes**, **J. Van Erps**, **N. Vermeulen**, **D. A. Miller**, and **H. Thienpont** (2013). B-calm: an open-source multi-gpu-based 3d-fdtd with multi-pole dispersion for plasmonics. *Progress In Electromagnetics Research*, **138**, 467âĂŞ478.