The University of Maine
DigitalCommons@UMaine

Electronic Theses and Dissertations Fogler Library

Fall 12-2021

Computer Modeling Using The Finite-Difference Time-Domain (FDTD) Method for Electromagnetic Wave Propagation

Atheer A. Oufi
atheer.oufi@maine.edu

Follow this and additional works at: https://digitalcommons.library.umaine.edu/etd

Part of the Electrical and Electronics Commons, and the Electromagnetics and Photonics Commons

Recommended Citation

Oufi, Atheer A., "Computer Modeling Using The Finite-Difference Time-Domain (FDTD) Method for Electromagnetic Wave Propagation" (2021). *Electronic Theses and Dissertations*. 3510.
https://digitalcommons.library.umaine.edu/etd/3510

This Open-Access Thesis is brought to you for free and open access by DigitalCommons@UMaine. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of DigitalCommons@UMaine. For more information, please contact um.library.technical.services@maine.edu.

COMPUTER MODELING USING THE FINITE-DIFFERENCE TIME-DOMAIN (FDTD) METHOD FOR ELECTROMAGNETIC WAVE PROPAGATION

By

Atheer A. Oufi

B.SC University of Technology, 2000

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Electrical Engineering)

The Graduate School

The University of Maine

December 2021

Advisory Committee:

Bruce Segee, Professor of Electrical and Computer Engineering, Advisor

Vincent Weaver, Associate Professor of Electrical and Computer Engineering

Mauricio Pereira da Cunha, Professor of Electrical and Computer Engineering

© 2021 Atheer A. Oufi

All Rights Reserved

COMPUTER MODELING USING THE FINITE-DIFFERENCE TIME-DOMAIN (FDTD) METHOD FOR ELECTROMAGNETIC WAVE PROPAGATION

By Atheer A. Oufi

Thesis Advisor: Dr. Bruce Segee

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Electrical Engineering)
December 2021

The Finite-Difference Time-Domain (FDTD) technique is a numerical analysis modeling method to find the solutions of the partial derivatives in Maxwell's equations to electromagnetic problems. In FDTD the electrical and magnetic fields components staggered in time and space by a method developed by Yee. The approximation of the solutions can be found using a set of updated equations.

In every simulation that utilizes the FDTD method, the factors of time and memory size are the two significant considerations. This study focused on reducing the computation time, as the time required to time-march the components of the electrical and magnetic fields at each of the FDTD problem cells is computationally expensive.

Based on the findings of this study, the issue of time can be solved by parallelizing the code. Since the structures of the FDTD field's components are independent, the algorithm of the FDTD can be divided into small tasks that can be executed concurrently. Two approaches were taken to parallelize the one- and two-dimensional FDTD code: The Compute Unified Device Architecture (CUDA) approach and Open Computing Language (OpenCL) approach.

The serial FDTD C code was implemented and accelerated using CUDA. The result of the comparison between the serial and parallel algorithms (C, CUDA, MATLAB) showed a speed-up of 505 speed factor with the GPU-GPU method and 5 speedup factor with the CPU-GPU method. This was the case for a one-dimensional space problem.

The FDTD code was implemented and executed with the OpenCL (Open Computing Language) software as well. The OpenCL software is important since it is open-source and freely available. In contrast to CUDA, which only supports NVIDIA and enabled GPUs, the code written in OpenCL is portable and can be executed on any parallel processing platforms such as CPUs, GPUs, DSPs, FPGAs, and others. Total time's Speedup of 22X has been recorded with OpenCL (PCL) with respect to CPU-C, with 10000 iterations and a 150000 cells grid size.

DEDICATION

This work is dedicated to my parents; my wife, Rawaa; my daughters, Feed, Aya and Baneen; and my son, Fadhl who gave me support and encouragement throughout each step of my life.

ACKNOWLEDGEMENTS

I would like to thank Professor Bruce Segee for making this master thesis possible, as well as for his supervision and guidance.

Atheer Oufi, August 2021

TABLE OF CONTENTS

DEDICATION…… iv

ACKNOWLEDGEMENTS…… v

LIST OF TABLES…… ix

LIST OF FIGURES…… x

Chapter

1. INTRODUCTION 1

1.1. Brief Review of Electromagnetics 1

1.1.1 Green's, Gauss's and Stokes' Theorems 1

1.1.2. Gauss' Divergence Theorem 2

1.1.3. Stokes's Theorem 3

1.1.4. Gauss's Law for Electrical and Magnetic Fields 3

1.1.5. Faraday's Law 4

1.1.6. Ampere's Law with Maxwell's Correction 5

1.2. Divergence and Curl 5

1.3. Summary Of the Time-Varying Fields 7

2. FINITE-DIFFERENCE TIME DOMAIN METHOD FDTD......8

2.1. Introduction......8

2.2. Approximation of the Partial Derivatives using Finite Differences......9

2.3. The Yee Algorithm 11

2.4. FDTD Updating Equations of Three-, Two-, and One-Dimensional Problems 15

2.4.1. Three-Dimensional Problems 19

2.4.2. Two-Dimensional Problems 27

2.4.3. One-Dimensional Problems 30

2.5. Conclusion 32

3. HARDWARE ACCELERATION OF FDTD METHOD – CUDA 34

3.1. Introduction 34

3.2. GPU Programming using CUDA 35

3.2.1. Compute Unified Device Architecture 36

3.2.2. Programming Model 37

3.2.2.1 CUDA Kernel 37

3.2.2.2 Thread Hierarchy. 38

3.2.2.3 Memory Hierarchy. 38

3.3. Hardware Testing Platform 42

3.4. CUDA Implementation of the One-, and Two-dimensional Problems 43

3.4.1 Main Time Loop 44

3.4.2. Programming GPGPU CUDA 46

3.5. Verification 48

3.6. Speedup, Parallel Efficiency, and Strong and Weak Scaling 54

3.7. Timing 55

3.8. Performance of the FDTD Parallel Code 56

3.8.1 GPU-GPU Performance 56

3.8.1.1. Grid Size 217 cells. 56

3.8.1.2. Grid Size 617 cells. 60

3.8.1.3. Grid Size 4017 cells. 61

3.8.2. CPU-GPU Performance 62

3.9. Conclusion 64

4. HARDWARE ACCELERATION OF THE FDTD METHOD – OpenCL...66

4.1. Introduction...66

4.2. GPU Programming Using OpenCL...66

4.2.1. Platform Model 66

4.2.2. Memory Model..67

4.2.2. Execution Model..69

4.2.3. Programming Model...71

4.2.3.1 Initialization. 71

4.2.3.2 Allocating memory 73

4.2.3.3. Program and Kernel Objects...73

4.2.3.4 Execution 75

4.3. Performance of One-, and Two-Dimensional FDTD Problems 75

5. Conclusions AND FUTURE WORK...79

5.1 FDTD Method 79

5.2 Future Work 81

5.2.1. FDTD Approximation 81

5.2.2. CUDA 82

5.2.3. OpenCL 82

5.2.4. Parallelizing Techniques 82

5.3 Conclusion 82

BIBLIOGRAPHY 84

APPENDIX A 86

BIOGRAPHY OF THE AUTHOR 93

LIST OF TABLES

Table 3.1. Specification of CUDA Quadro P400 40

Table 3.2. The error of the Hx values when running the 1D-FDTD code 55

Table 3.3. Performance of CUDA with 217 cells and 400 steps 61

Table 3.4. Performance of CUDA with 400,8193, and 10000 steps 61

Table 3.5. Performance of CUDA with 400,8193, and 10000 steps 62

Table 3.6. Performance of CUDA with 400,8193, and 10000 steps 62

Table 3.7. GPU-GPU Performance with 10000 iterations in different grid sizes 63

Table 3.8. CPU-GPU Performance with 10000 iterations in different grid sizes.......... 64

Table 4.1. Description of OpenCL device’s memories [20] 69

Table 4.2. OpenCL Platforms 78

Table 4.3. OpenCL Performance 78

LIST OF FIGURES

Figure 1.1. Flux-Divergence relation in Green' Theorem. 2

Figure 1.2. Flow-Curl relation in Green' Theorem. 2

Figure 1.3. Examples of Different type of divergence of a field D 6

Figure 1.4. Circulation of a vector field. 6

Figure 2.1a Forward, Backward, and Central Finite Differences Error with a step size of 0.8 10

Figure 2.1b Forward, Backward, and Central Finite Differences with a step size of 0.08 11

Figure 2.2. Electrical and Magnetic Components staggered without offset. 12

Figure 2.3. Electrical and Magnetic components staggered within half time step offset. 13

Figure 2.4. Electrical and Magnetic components staggered in Yee Algorithm. 14

Figure 2.5. Electrical and Magnetic components staggered in (one-dimensional) Yee Algorithm. 15

Figure 2.6. The Electric Dipole Moment when an external E field applied. 17

Figure 2.7. Magnetic field components on Yee algorithm. 21

Figure 2.8. Arrangement of field's components on a (3-D) Yee cell indexed at (*i*, *j*, *k*) 25

Figure 2.9. Unit cell Indexed as (*i*, *j*, *k*) Showing the Two Modes of Wave Propagation 28

Figure 2.10. Hz-Mode Sliced figure (2.9.) at z=0 28

Figure 2.11. 2D-FDTD Hz- Mode 29

Figure 2.12. Ez- Mode Sliced figure (2.9.) at z=0.5 29

Figure 2.13. 2D-FDTD Ez-Mode 30

Figure 2.14. Unit cell Indexed as (i, j, k) Showing the Two Modes of Propagation 31

Figure 2.15. One-Dimensional FDTD-Mode 1 31

Figure 2.16. One-Dimensional FDTD-Mode 2 31

Figure 2.17. Summery of the Formulation of the FDTD equations. 33

Figure 3.1. FDTD Field update equations in series 34

Figure 3.2. FDTD Updated Equations in Parallel 35

Figure 3.3. distributions of chip resources for CPU versus a GPU [10] 36

Figure 3.4. available memories for thread execution period [10] 39

Figure 3.5. An example of initializing and defining Two- Dimensional FDTD problem in MATLAB. 44

Figure 3.6. One-dimensional FDTD algorithm. 45

Figure 3.7. Two-dimensional FDTD algorithm (CPU). 46

Figure 3.8. An example of FDTD problem 49

Figure 3.9. An example of FDTD code in MATLAB 50

Figure 3.10. An example of the 1D-FDTD sequential code in C 50

Figure 3.11. CUDA function to call the kernel of the same code of figure (3.10.) 57

Figure 3.12. C, MATLAB, and CUDA running time with 10000 iterations and different grid size 63

Figure 3.13. Speedup of the CUDA with different grid sizes 64

Figure 3.14. An example of the times that CUDA needed to perform FDTD code 64

Figure 3.15. speedup of the CUDA with different grid sizes 65

Table 4.1. description of OpenCL device's memories [20] 68

Figure 4.2. Memory Configuration [20] 68

Figure 4.3. An example of two dimensional NDRange [17] 70

Figure 4.4. An example of three-dimensional NDRange (AMD)[18] 70

Figure 4.5. Create buffer function 73

Figure 4.6. Create and Build (program and kernel) functions 74

Figure 4.7. FDTD Updating Kernel 74

Figure 4.8. Kernel's Arguments 75

Figure 4.9. OpenCL platforms 76

Figure 4.10. OpenCL platforms 77

CHAPTER 1

1. INTRODUCTION

This thesis explores the use of computer models to calculate and visualize electromagnetic fields using discrete time and discrete space modeling, also known as the Finite Difference Time Domain methods (FDTD). This thesis derives the necessary FDTD equations and shows how to model electromagnetic wave propagation in 1D, 2D and 3D. Calculations using a conventional CPU are compared to calculations using Graphis Processing Units (GPUs) for accuracy, speedup, and efficiency. It is found that equally accurate results can be obtained in less time using GPUs in parallel as compared to a single CPU. Both speedup and efficiency increase with increasing problem size.

1.1. Brief Review of Electromagnetics

1.1.1 Green's, Gauss's and Stokes' Theorems

In vector calculus, Green's, Gauss', and Stokes' identities, have widely been used to simplify complex electromagnetism calculations by relating local properties to global properties. Local properties include things like the degree of spinning, whether faster or slower, or clockwise or counterclockwise; vector magnitude; and the direction of each individual point inside the region that is contained by the vector. Global properties are those such as flow or flux of the fields around or across the boundary of that region. Theorems are also used to relate (n-dimensional into (n-1) dimensional) integral problems.

Green's theorem (Divergence-flux and circulation-curl form) states that for C, a smooth, simple, closed curve enclosing R, a vector field **F** having continuous first partial in an open region containing R, as shown in figure (1.1.), the total flux across the boundary of R region and sum of the divergence of the field inside R, are equal.

$$\oint_C (\boldsymbol{F} \cdot \hat{n})\, ds = \iint_R (\nabla . \boldsymbol{F})\, dA = \iint_R \left(\frac{\partial M}{\partial x} + \frac{\partial N}{\partial y}\right) dxdy \qquad (1.1)$$

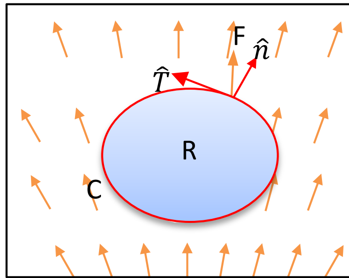


Figure 1.1. Flux-Divergence relation in Green' Theorem.

Green's theorem of circulation-curl form, figure (1.2.), states that the circulation around a closed region R and the curl of the vector field inside of R, are related according to equation 1.2, below:

$$\oint_C (\boldsymbol{F} \cdot \hat{T})\, ds = \iint_R (\nabla \times \boldsymbol{F})\, dA = \iint_R \left(\frac{\partial N}{\partial x} - \frac{\partial M}{\partial y}\right) dxdy \qquad (1.2)$$

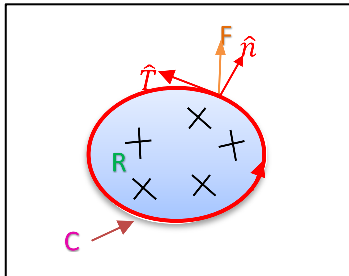


Figure 1.2. Flow-Curl relation in Green' Theorem.

1.1.2. Gauss' Divergence Theorem

The integral of a continuously differentiable vector field across a closed surface, which is called the flux through the surface, is equal to the integral of the divergence of that vector field within the region enclosed by the boundary [21], Let V be a sold region on R3 and let S be a surface of V (S must be oriented with an outward pointing normal vector) Gauss's Theorem states that for a given vector field **F**, the following equation holds:

$$\oiint_S (\boldsymbol{F} \cdot \hat{n})\, dA = \iiint_V \nabla . \boldsymbol{F}\, dv \qquad (1.3)$$

$\hat{n}$ is a unit normal vector at each point on **S**. Using electric field density **D** in Equation (1.3)**,** Gauss's theorem states:

$$\oiint_S (\boldsymbol{D} \cdot \hat{n})\, dS = \iiint_V \nabla . \boldsymbol{D}\, dv \qquad (1.4)$$

1.1.3. Stokes's Theorem

Stokes' theorem states that the integral of a vector field over any closed path is equal to the integral of the curl of that field over a surface which has that path as its border. Again, this holds for any vector field, but using the electric field as an example one can write:

$$\oint \boldsymbol{E} \,.\, dl = \iint_{S} \boldsymbol{\nabla} \times \boldsymbol{E} \;\, d\boldsymbol{s}$$

The surface normal is assumed to follow the right-hand convention so that when the fingers of the right hand are oriented along the path of the loop, the thumb points in the positive direction of the surface normal.

1.1.4. Gauss's Law for Electrical and Magnetic Fields

Carl Friedrich Gauss, in 1813, stated the law concerning the electric flux and electric charges. The law states that the amount of charge inside a closed volume V is equal to the total amount of electric flux (**D**) exiting the surface S. Mathematically, Gauss's law can be expressed in integral form and differential form. The integral form is:

$$Q_{enc} = \oiint_S \boldsymbol{D} \cdot d\boldsymbol{S} = \iiint_V \rho_e \, dv \qquad (1.6)$$

where **D** is electric flux density (coulombs/m^2), ***S*** is the closed surface (arbitrary three-dimensional surface), ***dS*** is differential normal vector that characterizes surface S (m^2), and Q_{enc} is the enclosed charge (coulombs). The differential form is:

$$\nabla . \boldsymbol{D} = \rho_e \qquad (1.6a)$$

where $\nabla \cdot \mathbf{D}$ is the divergence of the electric flux density, ρ_e is the electric charge density (coulombs/m^3).

Equation (1.6a) can be rewritten by using Gauss's theorem in Equation (1.6), Then, the equation (1.6) becomes:

$$\oiint_S \boldsymbol{D} \cdot d\boldsymbol{S} = \iiint_V \nabla . \boldsymbol{D} \, dv$$

$$\oiint_S \boldsymbol{D} \cdot d\boldsymbol{S} = \iiint_V \rho_e \, dv \xrightarrow{yields} \iiint_V \rho_e \, dv = \iiint_V \nabla . \boldsymbol{D} \, dv \xrightarrow{yields} \nabla . \boldsymbol{D} = \rho_e$$

The second law of Gauss states that the net magnetic flux through a closed surface is zero. In other words, the divergence of magnetic field **B** within a closed surface is equal to zero. The fact that **B** has no divergence makes sense, since the magnetic monopole does not exist.

$$\oiint_S \boldsymbol{B} \cdot d\boldsymbol{S} = 0 \qquad (1.7)$$

$$\nabla . \boldsymbol{B} = 0 \qquad (1.7a)$$

1.1.5. Faraday's Law

In 1831, Michael Faraday studied the electromagnetic induction properties and formed what was later named Faraday's Law of Induction. This law states that, "The electromotive force around a closed path is equal to the negative of the time rate of change of the magnetic flux enclosed by the path"[22].

$$\varepsilon = -\frac{d\phi_B}{dt} \qquad (1.8)$$

($\varepsilon(\text{volt}) = \oint \boldsymbol{E} \, . \, dl$) Is the electromotive force, and ($\phi_B = \iint_S \boldsymbol{B} \cdot d\boldsymbol{S}$) is the magnetic flux (Weber).

The integral form of Equation (1.8) can be written as:

$$\oint \boldsymbol{E} \, . \, dl = -\frac{\partial}{\partial t} \iint_S \boldsymbol{B} \cdot d\boldsymbol{S} \qquad (1.8a)$$

Where (d**S**) is an element of surface area of the moving surface S, **B** is magnetic flux density (Wb/m^2), **E** is electric field (volts/meter). Applying **Stokes' theorem** on Equation (1.8a) gives

$$\oint \boldsymbol{E} \, . \, dl = \iint_S \boldsymbol{\nabla} \times \boldsymbol{E} \; d\boldsymbol{S}$$

And eq (1.8a) becomes:

$$\iint_S \boldsymbol{\nabla} \times \boldsymbol{E} \; d\boldsymbol{S} = -\frac{\partial}{\partial t} \iint_S \boldsymbol{B} \cdot d\boldsymbol{S}$$

$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} \qquad (1.8b)$$

1.1.6. Ampere's Law with Maxwell's Extension

Ampere's law relates the closed line integral of the magnetic field ($\boldsymbol{H}$) (in amperes/meter) around the closed line L, to the total electric current crossing the area S enclosed by the path L.

The integral form of Ampere's law is:

$$\oint_L \boldsymbol{H}.dl = \iint_S J \cdot d\boldsymbol{S} = I_{enc} \qquad (1.9)$$

Where L is closed contour that bounds surface S, dl is differential length vector that characterizes contour L (meters), ds = $\hat{\mathrm{n}} \times \mathrm{d}S$. $\mathrm{I_{enc}}$ is the current that passes through the surface S, which is bounded by the loop L.

Since the differential form of Ampere's law states that the divergence of magnetic field is equal to zero, applying Stokes' theorem $\oint_L \mathbf{H}.\mathrm{d}\mathbf{l} = \iint_S (\nabla \times \mathbf{H}).\mathrm{d}\mathbf{S}$ and Equation (1.9) becomes:

$$\iint_S (\nabla \times \mathbf{H}).\mathrm{d}\mathbf{S} = \iint_S \mathbf{J} \cdot \mathrm{d}\mathbf{S}$$

$$\nabla \times \mathbf{H} = \mathbf{J} \qquad (1.9b)$$

Equation (1.9b) is known as Ampere's Law. Maxwell later added a temporal derivative of the displacement current, density $\frac{\partial \boldsymbol{D}}{\partial t}$, to the right-hand side of the equation to make it applicable to time-varying current [23]. And the corrected equation is expressed as:

$$\nabla \times \boldsymbol{H} = \mathbf{J} + \frac{\partial \boldsymbol{D}}{\partial t} \qquad (1.9c)$$

$$\oint_L \boldsymbol{H}.d\boldsymbol{l} = \iint_S (\nabla \times \mathbf{H}).\mathrm{d}\mathbf{S} = \iint_S \mathbf{J}.\mathrm{d}\mathbf{S} + \iint_S \frac{\partial \boldsymbol{D}}{\partial t} \cdot \mathrm{d}\mathbf{S} \qquad (1.9d)$$

1.2. Divergence and Curl

The divergence operator of a vector field is the measure of the vector flow out/into of an imaginary surface surrounding a point P (x, y, z). In other words, it is a measure of the rate of change of the vector field in the x, y, and z directions.

$$\nabla. = \frac{\partial}{\partial x} + \frac{\partial}{\partial y} + \frac{\partial}{\partial z} \qquad (1.10)$$

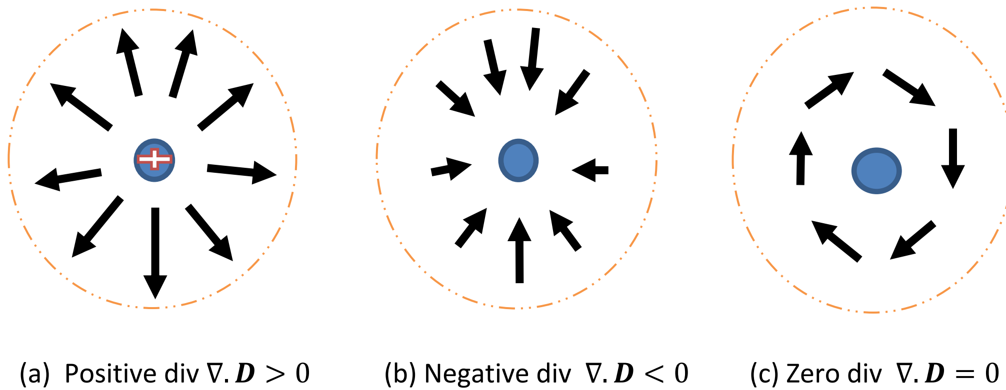
Let us assume that the density of electrical field **D** is equal to

$$\mathbf{D} = \begin{bmatrix} D_x \\ D_y \\ D_z \end{bmatrix} = D_x \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + D_y \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + D_z \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The divergence of **D** field is the sum of how fast the vector function is changing:

$$\nabla . \boldsymbol{D} = \frac{\partial D_x}{\partial x} + \frac{\partial D_y}{\partial y} + \frac{\partial D_z}{\partial z}$$

The divergence of any vector field could be positive, negative, or zero (no divergence).



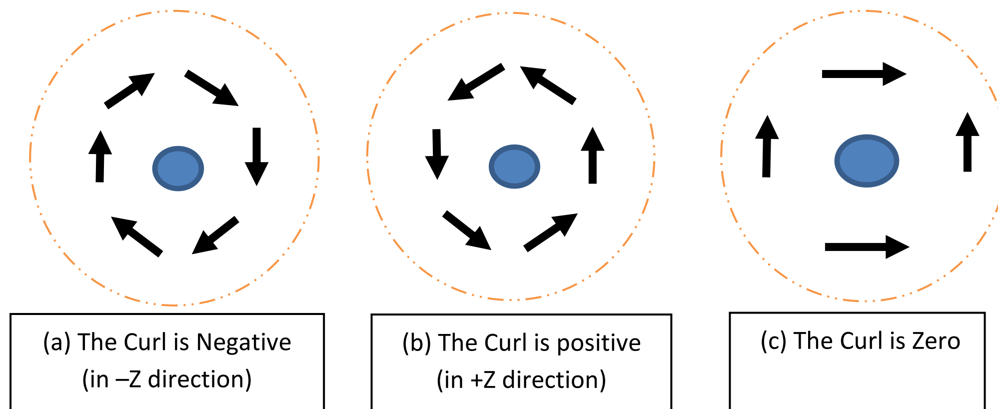
(a) Positive div $\nabla . \boldsymbol{D} > 0$ (b) Negative div $\nabla . \boldsymbol{D} < 0$ (c) Zero div $\nabla . \boldsymbol{D} = 0$

Figure 1.3. Examples of Different type of divergence of a field D

The curl operator ($\nabla \times$) Is a measure of the rotation of a vector field. Let **(E)** be a vector field with

(E_x, E_y, E_z) function components in (x, y, z) directions. That is:

$$\boldsymbol{E} = E_x \hat{x} + E_y \hat{y} + E_z \hat{z}$$



(a) The Curl is Negative (in –Z direction)

(b) The Curl is positive (in +Z direction)

(c) The Curl is Zero

Figure 1.4. Circulation of a vector field

The curl of E field:

$$\nabla \times = \left(\frac{\partial}{\partial y} - \frac{\partial}{\partial z}\right) \hat{x} + \left(\frac{\partial}{\partial z} - \frac{\partial}{\partial x}\right) \hat{y} + \left(\frac{\partial}{\partial x} - \frac{\partial}{\partial y}\right) \hat{z} \qquad (1.11)$$

$$\nabla \times \boldsymbol{E} = \begin{bmatrix} \hat{x} & \hat{y} & \hat{z} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ E_x & E_y & E_z \end{bmatrix} = \left(\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z}\right)\hat{x} + \left(\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x}\right)\hat{y} + \left(\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y}\right)\hat{z}$$

1.3. Summary Of the Time-Varying Fields

$\nabla . \boldsymbol{D} = \rho e$ (1.12)

$\nabla . \boldsymbol{B} = 0$ (1.13)

$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} - M$ (1.14)

$\nabla \times \boldsymbol{H} = \mathbf{J} + \frac{\partial \boldsymbol{D}}{\partial t}$ (1.15)

The following symbols are defined:

E : electric field (volts / meter)

D : electric flux density (coulombs / meter^2)

H : magnetic field (amperes / meter)

B : magnetic flux density (webers / meter^2)

M : equivalent magnetic current density (volts / meter^2)

1- For the static case ($\frac{\partial}{\partial t} = 0$), the first two equations, the divergence equations, will have no change since they do not have any terms that are time dependent. The curl equations (Equation (1.14) and (1.15)) both have time dependent terms that will not change for the static case so these terms will be zero and the curl equations becomes:

$\nabla \times \boldsymbol{E} = 0$ (1.16)

$\nabla \times \boldsymbol{H} = \mathbf{J}$ (1.17)

The curl equations (1.14) and (1.15) coupling both electrical and magnetic fields and will be used to drive the necessary equations for the FDTD technique.

CHAPTER 2

2. FINITE-DIFFERENCE TIME DOMAIN METHOD FDTD

The Finite Difference Time Domain method (FDTD) is a simple but powerful, accurate, and robust technique in numerical computational methods. Currently, and in the recent past, a plethora of applications related to electromagnetics phenomena has needed to solve Maxwell's equations in time and space[5]. These applications include but are not limited to, radar technology (generating, sending, cand receiving, analyzing of electromagnetic waves), antenna and waveguides design, medical applications, wi-fi, cellular telephone, and non-linear and frequency-dependent materials[1]. Frequency domain techniques were commonly used but the rapid development of computer and memory technology and the basic mathematics equations for the formulation of the FDTD techniques has changed the emphasis toward FDTD. FDTD allows one to visualize the propagation of the electrical **E** field and the magnetic **H** field before, during, and after interacting with structures of interest (or free space) giving the developer an understanding of the response of the system in every step of the testing time.

2.1. Introduction

The differential forms of Maxwell's curl equations are the backbone of the FDTD method. The original idea was first introduced by Kane Yee in 1966 [1]. Yee used the second-order finite central-differences to approximate the time partial derivative that appear in Maxwell's equations. The elegance of FDTD is due to the way that the electrical and magnetic components of the E and H fields are distributed in Yee's geometry space.

This chapter will explain the formulation of the FDTD equations the second order finite difference approximation method, the compute time and spatial steps ($\triangle t$ and $\triangle x$) and the compute update coefficients.

2.2. Approximation of the Partial Derivatives using Finite Differences

The partial derivative of a function in the x direction can be computed using the Taylor series expansion of the function over an infinitesimally small interval in the x-direction (holding all other dimensions constant). Let $f(x)$ be a function of x then Taylor series expansion of $f(x + \Delta x)$ when $\lim_{\Delta x \to 0} f(x \mp \Delta x) = f(x)$ is given by:

1- The forward finite differences:

$$f(x + \Delta x) = f(x) + \frac{\Delta x}{1!}\frac{\partial f(x)}{\partial x} + \frac{(\Delta x)^2}{2!}\frac{\partial^2 f(x)}{\partial x^2} + \frac{(\Delta x)^3}{3!}\frac{\partial^3 f(x)}{\partial x^3} + \cdots, \qquad (2.1)$$

solve for $\frac{\partial f(x)}{\partial x}$, and assume $O(\Delta x) = \frac{(\Delta x)}{2!}\frac{\partial^2 f(x)}{\partial x^2} + \frac{(\Delta x)^2}{3!}\frac{\partial^3 f(x)}{\partial x^3} + \cdots,$

$$\frac{\partial f(x)}{\partial x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} + O(\Delta x) \qquad (2.2)$$

2- The backward finite differences approximation:

$$f(x - \Delta x) = f(x) - \frac{\Delta x}{1!}\frac{\partial f(x)}{\partial x} + \frac{(\Delta x)^2}{2!}\frac{\partial^2 f(x)}{\partial x^2} - \frac{(\Delta x)^3}{3!}\frac{\partial^3 f(x)}{\partial x^3} + \cdots, \qquad (2.3)$$

solve for $\frac{\partial f(x)}{\partial x}$, and assume $O(\Delta x) = \frac{\Delta x}{2!}\frac{\partial^2 f(x)}{\partial x^2} - \frac{(\Delta x)^2}{3!}\frac{\partial^3 f(x)}{\partial x^3} + \cdots,$

$$\frac{\partial f(x)}{\partial x} = \frac{f(x) - f(x-\Delta x)}{\Delta x} + O(\Delta x) \qquad (2.4)$$

3- The central finite differences approximation is the Taylor series expansion of $f(x + \Delta x) - f(x - \Delta x)$

$$f(x + \Delta x) - f(x - \Delta x) = \left(f(x) + \frac{\Delta x}{1!}\frac{\partial f(x)}{\partial x}\right) - \left(f(x) - \frac{\Delta x}{1!}\frac{\partial f(x)}{\partial x}\right) + \frac{2(\Delta x)^3}{3!}\frac{\partial^3 f(x)}{\partial x^3} - \cdots,$$

$$\frac{f(x + \Delta x) - f(x - \Delta x)}{2\,\Delta x} = \frac{\partial f(x)}{\partial x} + O(\Delta x^2)$$

taking $(\frac{\Delta x}{2})$ Sufficiently small, the term $O(\frac{\Delta x}{2})^2$ can be neglected and the equation becomes:

$$\frac{\partial f(x)}{\partial x} = \frac{f\left(x+\frac{\Delta x}{2}\right) - f\left(x-\frac{\Delta x}{2}\right)}{\Delta x} \qquad (2.5)$$

The term $O(\Delta x^n)$ represents all the terms that are neglected. If Δx is sufficiently small, then $O(\Delta x^n)$ is the lowest order of (Δx) in these neglected terms. For the case of forward and backward finite differences, this term is $O(\Delta x^1)$ since the lowest power of the neglected terms is 1 and the

forward and backward finite differences are said to have first order approximation accuracy. For the central finite differences, the term is $O(\Delta\times^2)$. Since the lowest power of hidden terms in $O(\Delta\times^n)$ is 2, the central finite differences then said to have second order accuracy. If (Δ×) is reduced by a factor of 10 then the error in approximation should be reduced by a factor of 100. As (Δ×) approaches zero the approximation becomes exact.

Let f(x) = sin(x), consider computing $\frac{\partial f(x)}{\partial x}$ using forward, backward, and central differences methods with two steps sizes $\Delta\times = 0.8, and\ 0.08$. The actual derivative is known to be $\frac{\partial f(x)}{\partial x}$ = cos(x). Figure (2.1. a) and (2.1. b) show the error between the exact and approximation derivatives with steps (0.8,0.08). The forward and backward error are reduced by a factor of 10 (0.4 with $\Delta\times = 0.8 \longrightarrow 0.04\ with\ \Delta\times = 0.08$) and are reduced by a factor of 100 with the central differences' method (error 0.1 with $\Delta\times = 0.8 \longrightarrow 0.001\ with\ \Delta\times = 0.08$).

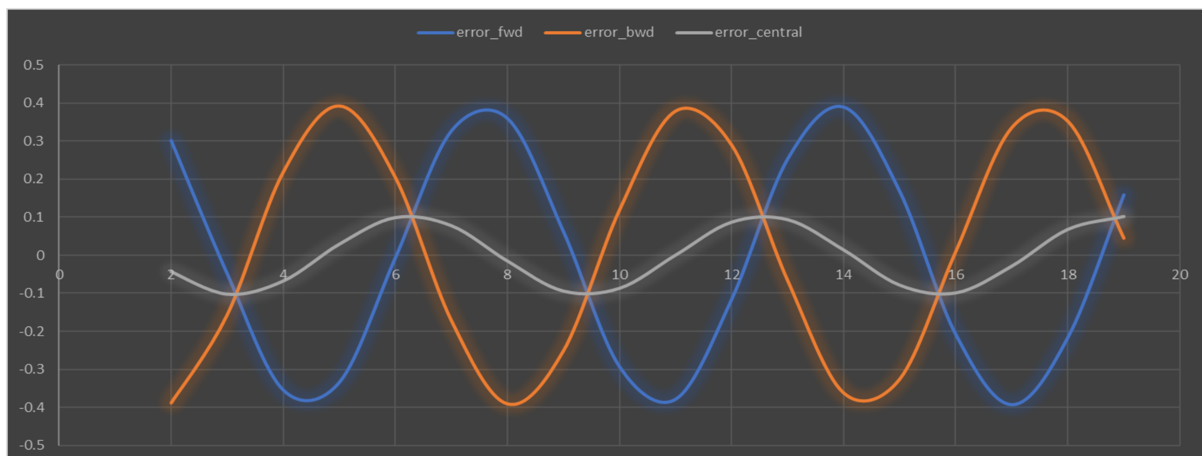


Figure 2.1a Forward, Backward, and Central Finite Differences Error with a step size of 0.8

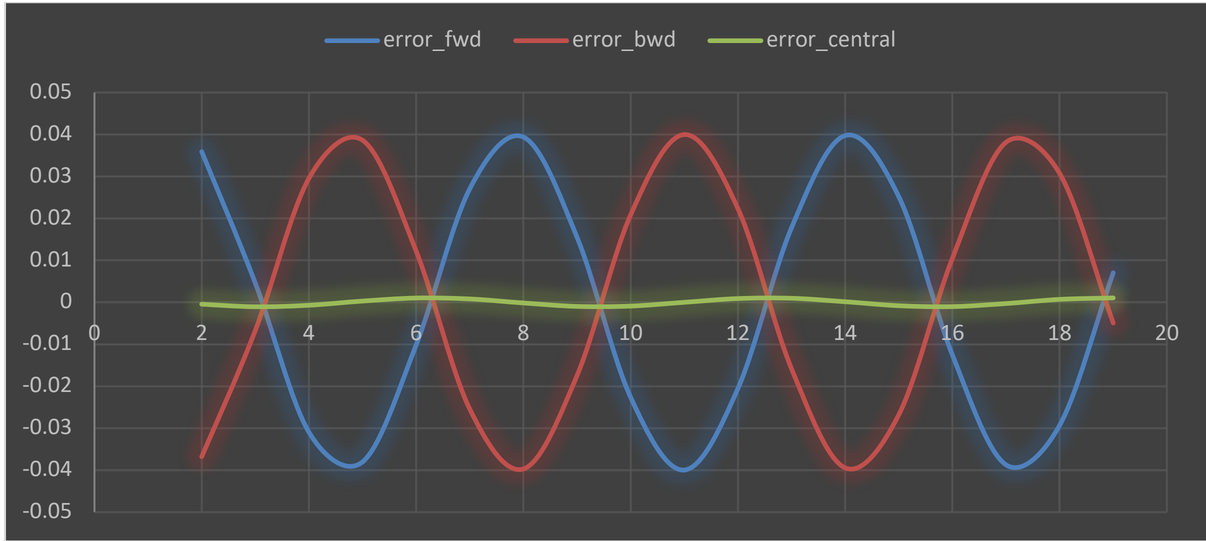


Figure 2.1 b Forward, Backward, and Central Finite Differences with a step size of 0.08

2.3. Yee's Algorithm

Consider one dimensional space case with $\left(\frac{\partial \boldsymbol{H}}{\partial y}, \frac{\partial \boldsymbol{H}}{\partial z}, \frac{\partial \boldsymbol{E}}{\partial y}, \frac{\partial \boldsymbol{E}}{\partial z}, E_y, H_x, H_z = 0\right)$ $and\ \boldsymbol{J} = 0$. Then apply Faraday's law using Equation (1.14):

$$\nabla \times \boldsymbol{E} = -\mu \frac{\partial \boldsymbol{H}}{\partial t}$$

$$\begin{bmatrix} \hat{x} & \hat{y} & \hat{z} \\ \frac{\partial}{\partial x} & 0 & 0 \\ E_x & 0 & E_z \end{bmatrix} = (0-0)\hat{x} + \left(0 - \frac{\partial E_z}{\partial x}\right)\hat{y} + (0-0)\hat{z} = -\mu\left(\frac{\partial \boldsymbol{H_x}}{\partial t}\hat{x} + \frac{\partial \boldsymbol{H_y}}{\partial t}\hat{y} + \frac{\partial \boldsymbol{H_z}}{\partial t}\hat{z}\right)$$

$$\mu \frac{\partial \boldsymbol{H_y}}{\partial t} = \frac{\partial \boldsymbol{E_z}}{\partial x} \qquad (2.6)$$

Similarly, Ampere's Law, Equation (1.15) becomes:

$$\varepsilon \frac{\partial \boldsymbol{E_z}}{\partial t} = \frac{\partial \boldsymbol{H_y}}{\partial x} \qquad (2.7)$$

Equations (2.6) and (2.7) hold if they have been taken for the same space-time points. The first equation will be used to find the magnetic field value in the next future step while the second equation will be used to find the electric field value in the next future step.

Electrical and magnetic fields need to be sampled in space and time domains. Let

$$E_z(x,t) = E_z(i+\Delta x, t+\Delta t) = E_z|_{i+\Delta x}^{t+\Delta t}$$

and

$$H_y(x,t) = H_y(i+\Delta x, t+\Delta t) = H_y\Big|_{i+\Delta x}^{t+\Delta t}$$

The components will be calculated at discrete time steps separated by delta t, as in figure (2.2.)

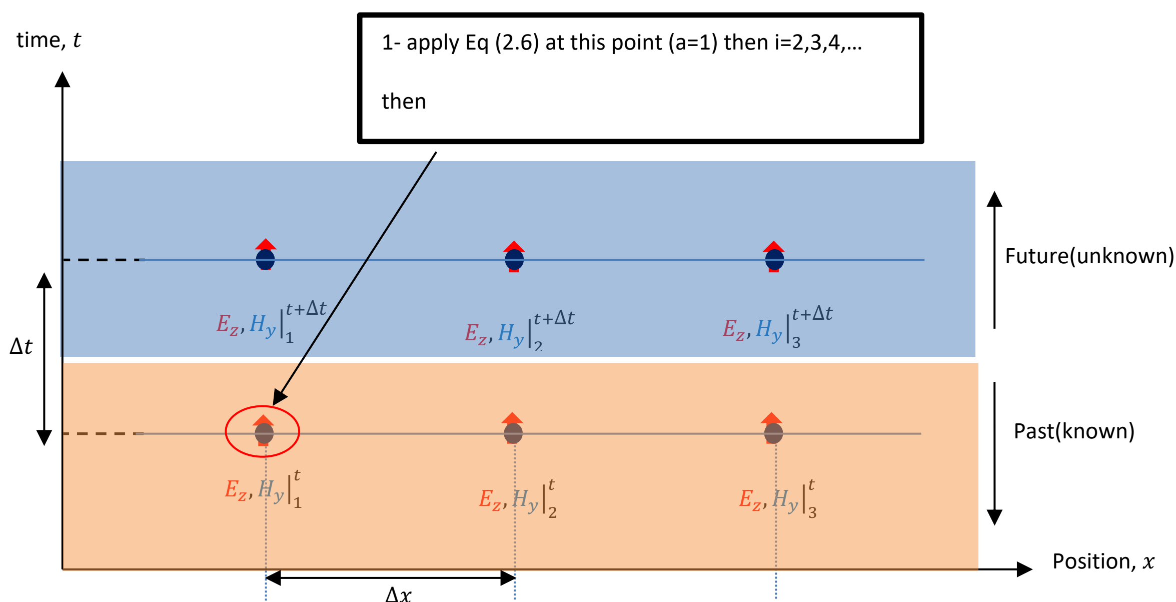


Figure 2.2. Electrical and Magnetic Components staggered without offset.

One can notice that after Eq (2.6) is solved to get the future $H_y\big|_i^{t+\Delta t}$ for all H_y components, and Equation (2.7) is solved to obtain the future $E_z|_i^{t+\Delta t}$ components by using the past $E_z|_i^t$ and the past $H_y\big|_i^t$ components (not the updated ones we got after solving Eq (2.6)), nothing will change in the simulation overall. We need to update the values of the H_y field and use it to find the values of the future E_z field. Later we will use the updated E_z values to find the future H_y field's values, and so on. To do that we will stagger the E_z components by one half time step ($\Delta t/2$) offset as shown in figure (2.3.)

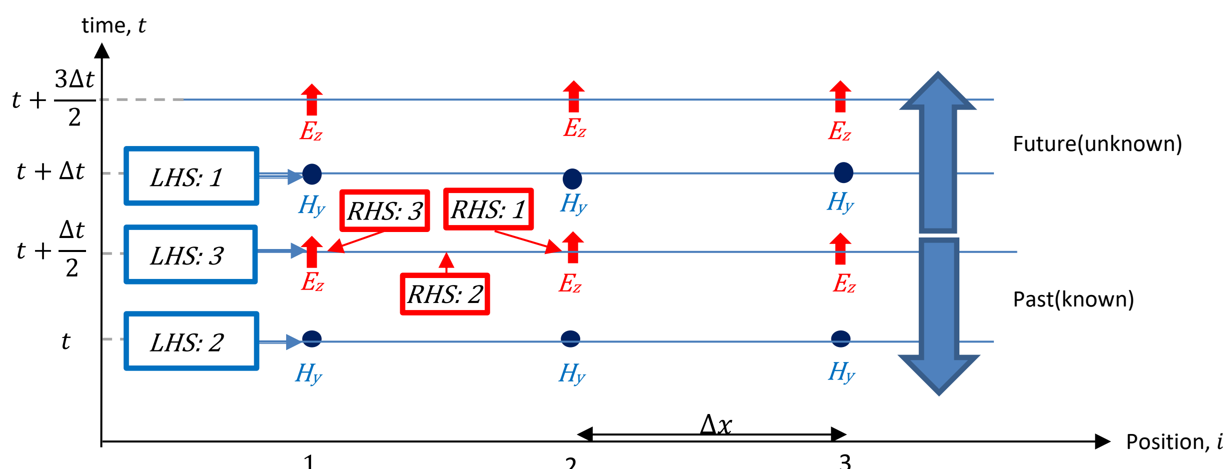


Figure 2.3. Electrical and Magnetic components staggered within half time step offset.

Notice that $(H_y(i,t), E_z\left(i, t+\frac{\Delta t}{2}\right),\ and\ E_z(i+1, t+\frac{\Delta t}{2}))$ are defined values (known), and by solving the $\mu\frac{\partial \boldsymbol{H_y}}{\partial t} = \frac{\partial \boldsymbol{E_z}}{\partial x}$, the value of the future $H_y\ at\ (i, t+\Delta\ t)$ will be computed.

A- LHS: To approximate the temporal derivative there are three choices at points (LHS:1, LHS:2, LHS:3)

1- **LHS:1** - Backward Finite Differences ($\frac{\partial f(x)}{\partial x} = \frac{f(x)-f(x-\Delta x)}{\Delta x}$): $\frac{\partial \boldsymbol{H_y}}{\partial \boldsymbol{t}}$ at point$(i, t+\Delta\ t)$ = $\frac{H_y|_i^{t+\Delta t} - H_y|_i^t}{\Delta t}$.

The approximation of the derivative in this approach has been taken at the y-axis ($t+\Delta\ t$) which is different than both terms of the spatial derivative in RHS $E_z\left(i, t+\frac{\Delta t}{2}\right), and\ E_z(i+1, t+\frac{\Delta t}{2})$. Therefore, this approach has obvious drawbacks.

2- **LHS:2** - Forward Finite Differences ($\frac{\partial f(x)}{\partial x} = \frac{f(x+\Delta x)-f(x)}{\Delta x}$): $\frac{\partial \boldsymbol{H_y}}{\partial \boldsymbol{t}}$ at the point(i, t) = $\frac{H_y|_i^{t+\Delta t} - H_y|_i^t}{\Delta t}$.

Again, this solution is not desirable for the same reason above.

3- **LHS:3** Central Finite Differences ($\frac{\partial f(x)}{\partial x} = \frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x}$): $\frac{\partial \boldsymbol{H_y}}{\partial \boldsymbol{t}}$ at point $(i, t+\frac{\Delta t}{2})$ = $\frac{H_y|_i^{(t+\frac{\Delta t}{2})+\frac{\Delta t}{2}} - H_y|_i^{(t+\frac{\Delta t}{2})-\frac{\Delta t}{2}}}{2\frac{\Delta t}{2}} = \frac{H_y|_i^{t+\Delta t} - H_y|_i^t}{\Delta t}$.

This solution has the property that the terms exist on the same y-axis as other terms in the RHS of the equation.

B- RHS: to approximate the spatial derivative we have three choices:

1- RHS:1 - Backward Finite Differences: $\frac{\partial E_z}{\partial x}$ at point$(i+1, t+\frac{\Delta t}{2})$ and

2- RHS:2 - Central Finite Differences: $\frac{\partial E_z}{\partial x}$ at point$(i+\frac{1}{2}, t+\frac{\Delta t}{2})$.

Both approaches will not match the x-axis of the LHS of the equation.

3- RHS:3 - Forward Finite Differences ($\frac{\partial f(x)}{\partial x} = \frac{f(x+\Delta x)-f(x)}{\Delta x}$): $\frac{\partial E_z}{\partial x}$ (At point$(i, t+\frac{\Delta t}{2})$) = $\frac{E_z|_{i+1}^{t+\frac{\Delta t}{2}} - E_z|_{i}^{t+\frac{\Delta t}{2}}}{\Delta x}$.

So, the correct solution to solve the equation $\mu\frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x}$ for the future $H_y's$ can be found by approximating the temporal derivative using central finite differences and the spatial derivative by forward finite differences. The only problem with this solution is the accuracy of the forward finite differences method which is order one and the error will accumulate during the time of the simulation and potentially lead to instability.

Kane Yee, in 1966, solved the partial derivatives of Maxwell's equations using the second-order central finite differences approximation by staggering the fields' components in half time-space steps as shown in the following figure (2.4.).

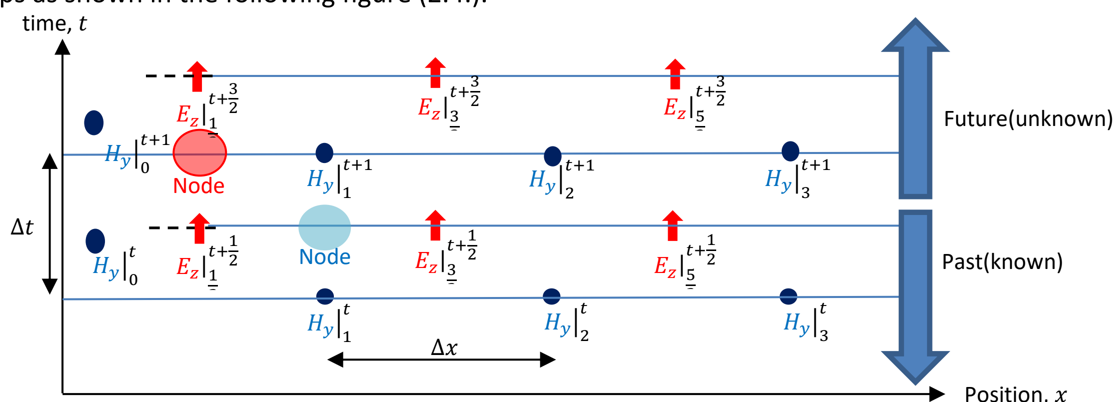


Figure 2.4. Electrical and Magnetic components staggered in Yee Algorithm.

1- at Node 1: we solve for at $\mu \frac{\partial H_y}{\partial t}\Big|_i^{t+\frac{1}{2}} = \frac{\partial E_z}{\partial x}\Big|_i^{t+\frac{1}{2}}$ by using central finite differences.

$$\mu \frac{H_y|_i^{(t+\frac{1}{2})+\frac{1}{2}} - H_y|_i^{(t+\frac{1}{2})-\frac{1}{2}}}{2\times\frac{\Delta t}{2}} = \frac{E_z|_{i+\frac{1}{2}}^{t+\frac{1}{2}} - E_z|_{i-\frac{1}{2}}^{t+\frac{1}{2}}}{2\times\frac{\Delta x}{2}}$$

$$\frac{H_y|_i^{t+1} - H_y|_i^t}{\Delta t} = \frac{1}{\mu\Delta x}(E_z|_{i+\frac{1}{2}}^{t+\frac{1}{2}} - E_z|_{i-\frac{1}{2}}^{t+\frac{1}{2}})$$

$$H_y[i]\Big|^{t+1} = H_y[i]\Big|^t + \frac{\Delta t}{\mu\Delta x}(E_z[i+\tfrac{1}{2}]\Big|^{t+\frac{1}{2}} - E_z[i-\tfrac{1}{2}]\Big|^{t+\frac{1}{2}})$$

2- at Node 2: we solve for at $\boldsymbol{\varepsilon \frac{\partial E_z}{\partial t}\Big|_{i+\frac{1}{2}}^{t+1} = \frac{\partial H_y}{\partial x}\Big|_{i-\frac{1}{2}}^{t+1}}$ by using central finite differences**.**

$$\boldsymbol{E_z|_{i+\frac{1}{2}}^{t+1+\frac{1}{2}} = E_z|_{i+\frac{1}{2}}^{t+1-\frac{1}{2}} + \frac{\Delta t}{\varepsilon\Delta x}(H_y|_{(i-\frac{1}{2})+\frac{1}{2}}^{t+1} - H_y|_{(i-\frac{1}{2})-\frac{1}{2}}^{t+1})}$$

$$E_z[i+\tfrac{1}{2}]\Big|^{t+\frac{3}{2}} = E_z[i+\tfrac{1}{2}]\Big|^{t+\frac{1}{2}} + \frac{\Delta t}{\mu\Delta x}(H_y[i]\Big|^{t+1} - H_y[i-1]\Big|^{t+1})$$

In C and MATLAB, the electrical and magnetic fields are stored in separate arrays. The elements of each array are accessed by using integer indices. One can imagine $E_z[i+\frac{1}{2}]$ as $E_z[i+1]$ which is at the right of $H_y[i]$ and $E_z[i-\frac{1}{2}]$ as $E_z[i]$ which is at the left of $H_y[i]$. As shown in Figure (2.5.).

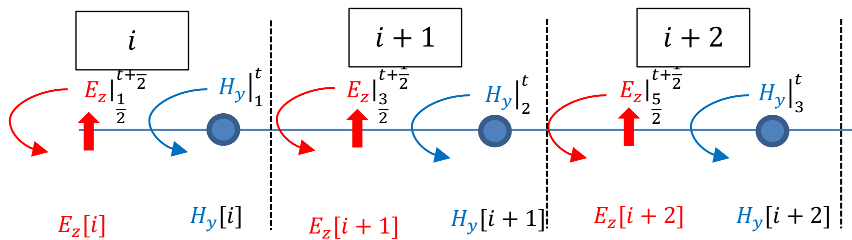


Figure 2.5. Electrical and Magnetic components staggered in (one-dimensional) Yee's Algorithm.

Hy(i)= Hy(i) + mHy *(Ez(i+1) - Ez(i))/dx.

Ez(i) = Ez(i) + mEy*(Hy(i) - Hy(i - 1))/dx.

2.4. FDTD Updating Equations of Three-, Two-, and One-Dimensional Problems

The Finite Differences Time Domain (FDTD) Method uses Maxwell's equations to formulate updated equations. The first two equations of Maxwell are called the divergence equations while the other

two are called the curl equations. These equations together describe how the electric field (E-field) and magnetic field (H-fields) behave and couple in nature.

$\nabla.\boldsymbol{D} = \rho e$ (1.12)

$\nabla.\boldsymbol{B} = 0$ (1.13)

$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} - \boldsymbol{M}$ (1.14)

$\nabla \times \boldsymbol{H} = \mathbf{J} + \frac{\partial \boldsymbol{D}}{\partial t}$ (1.15)

All materials are made up of charged particles. Different materials have different responses to the applied electrical and/or magnetic fields. The response will depend on the strength of the applied field (i.e., it may tear positive and negative bound charges apart), the frequency-dependent behavior of dispersive materials, and/or the unique structure of some materials such as crystals that have a directionally dependent response. For all of that, we need to relate **E** and **B** to **D** and **H,** respectively.

$$\boldsymbol{D}(r,t) = \varepsilon_0 \boldsymbol{E}(r,t) + \boldsymbol{P}(r,t) \quad (2.8)$$

$$\boldsymbol{H}(r,t) = \frac{1}{\mu_0}\boldsymbol{B} - \boldsymbol{M}(r,t) \quad (2.9)$$

Where ***M*** is the magnetization or magnetic moment per unit volume, and ***P*** is the polarization density which could be defined as the ratio of the electric dipole moments $\boldsymbol{p}$ (Coulombs-meter) per unit volume (cubic meter).

$\boldsymbol{P} = \frac{\boldsymbol{p}}{dV} = \frac{q_b}{dV}\boldsymbol{d}$, where **P** is polarization density (coulombs/square meters), **p** polarization of the material, electric dipole moment, (coulombs-meter), q_b bounded charges (coulomb), **d** displacement vector pointing from -q to +q. See Figure (2.6.)

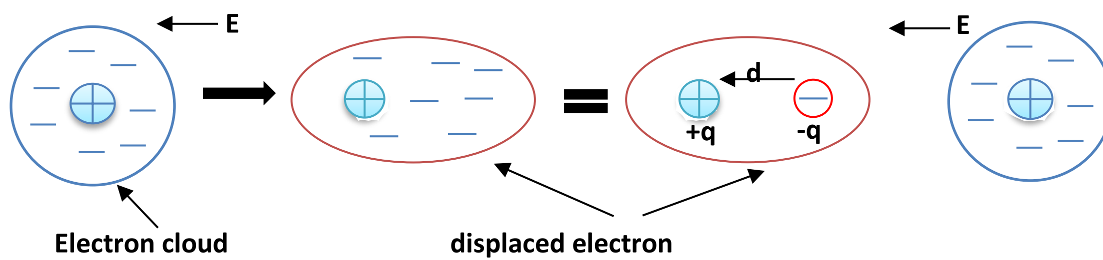


Figure 2.6. The Electric Dipole Moment when an external E field applied.

A- For isotropic linear materials $\boldsymbol{P} = \varepsilon_0 \, x_e \boldsymbol{E}$, and $\boldsymbol{M} = x_m \boldsymbol{H}$ (This relation is limited to ferromagnetic materials only). Thus, substituting this into eq (2.8)

$$\boldsymbol{D} = \varepsilon_0 \, \boldsymbol{E} + \varepsilon_0 \, x_e \boldsymbol{E}$$

$$\boldsymbol{H} = \frac{1}{\mu_0} \boldsymbol{B} - x_m \boldsymbol{H}$$

$$\boldsymbol{D} = \varepsilon_0 \, \boldsymbol{E} \, (1 + x_e),$$

$$\boldsymbol{H} = \frac{1}{\mu_0 \, (1 + x_m)} \boldsymbol{B}$$

Where x_e, x_μ are electric and magnetic susceptibility constants that indicate the degree of polarization and magnetization of a dielectric material, and it is related to its relative permittivity and permeability respectively by:

$$x_e = \varepsilon_r - 1$$

$$x_\mu = \mu_r - 1$$

$$\boldsymbol{D} = \varepsilon_0 \, \varepsilon_r \, \boldsymbol{E}$$

$$\boldsymbol{H} = \frac{1}{\mu_0 \, \mu_r} \boldsymbol{B}$$

$$\boldsymbol{D} = \varepsilon \boldsymbol{E} \qquad (2.10)$$

$$\boldsymbol{H} = \frac{1}{\mu} \boldsymbol{B} \qquad (2.11)$$

Where:

ε is the electric permittivity = $\varepsilon_0 \, \varepsilon_r$ (farads / meter)

ε_r relative electric permittivity (dimensionless scalar)

ε_0 free-space electric permittivity (8.854 x 10^{-12})

μ magnetic permeability (henrys / meter)

μ_r relative permeability (dimensionless scalar)

μ_0 free-space permeability (4π x 10^{-7} henrys / meter)

B – For dispersive materials:

$$\boldsymbol{D}(t) = \varepsilon(t) * \boldsymbol{E}(t) \quad (2.12)$$

$$\boldsymbol{H}(t) = \frac{1}{\mu(t)} * \boldsymbol{B}(t) \quad (2.13)$$

3- Anisotropic materials:

$$\boldsymbol{D}(t) = [\varepsilon]\, \boldsymbol{E}(t) \quad (2.14)$$

$$\boldsymbol{H}(t) = \frac{1}{[\mu(t)]} \boldsymbol{B}(t) \quad (2.15)$$

4- Nonlinear materials:

$$\mathbf{D}(t) = \varepsilon_0 [\, 1 + x_e^{(1)}\, \mathbf{E(t)} + x_e^{(2)}\, \mathbf{E^2(t)} + x_e^{(3)}\, \mathbf{E^3(t)} + \ldots] \quad (2.16)$$

In case of the isotropic, nondispersive materials:

Note that the electric and magnetic currents ***J*** and ***M*** are the sum of the independent impressed sources of ***E*** and ***H*** field energy, $\boldsymbol{J_i}$ and $\boldsymbol{M_i}$, and the conduction current densities ($\boldsymbol{J_c} = \sigma^e \boldsymbol{E}$) and the magnetic current density $\boldsymbol{M_c} = \sigma^m \boldsymbol{H}$

$$\boldsymbol{J} = \boldsymbol{J_c} + \boldsymbol{J_i} = \sigma^{\boldsymbol{e}}\, \boldsymbol{E} + \boldsymbol{J_i} \quad (2.17)$$

$$\boldsymbol{M} = \boldsymbol{M_c} + \boldsymbol{M_i} = \sigma^m\, \boldsymbol{H} + \boldsymbol{M_i} \quad (2.18)$$

Where σ^e is the electric conductivity (siemens/meter), and σ^m is the magnetic conductivity (ohms/meter). By using equations (2.17), and (2.18) we can rewrite Maxwell's equations (1.14), and (1.15) as:

$$\nabla \times \boldsymbol{H} = \frac{\partial \boldsymbol{D}}{\partial t} + \sigma^{\boldsymbol{e}}\, \boldsymbol{E} + \boldsymbol{J}_i \quad (2.19)$$

$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} - \sigma^m\, \boldsymbol{H} - M_i \quad (2.20)$$

2.4.1. Three-Dimensional Problems

For constitutive equations of linear, isotropic, and nondispersive materials, equations (2.14), and (2.15), and equations (2.19) and (2.20) become:

$$\nabla \times \boldsymbol{H} = [\varepsilon]\,\frac{\partial \boldsymbol{E}}{\partial t} + \sigma^{\boldsymbol{e}}\,\boldsymbol{E} + \boldsymbol{J}_i$$

$$\nabla \times \boldsymbol{E} = -[\mu]\,\frac{\partial \boldsymbol{H}}{\partial t} - \sigma^{m}\,\boldsymbol{H} - M_i$$

$$\varepsilon = \varepsilon_r\,\varepsilon_0,\ \mu = \mu_r\,\mu_0,\ c_0 = \sqrt{\frac{1}{\varepsilon_0\,\mu_0}} = 299792458\,m/s\ ,\quad n_0 = \sqrt{\frac{\mu_0}{\varepsilon_0}} = 376.73031346177\,\Omega,$$

Let $\mathbf{H} = \frac{\widetilde{\boldsymbol{H}}}{\mathrm{n}_0}$

$$\nabla \times \widetilde{\boldsymbol{H}} = n_0[\varepsilon]\,\frac{\partial \boldsymbol{E}}{\partial t} + n_0\sigma^{\boldsymbol{e}}\,\boldsymbol{E} + n_0\,\boldsymbol{J}_i = \sqrt{\frac{\mu_0}{\varepsilon_0}}\,[\varepsilon_r\,\varepsilon_0\,]\frac{\partial \boldsymbol{E}}{\partial t} + n_0\sigma^{\boldsymbol{e}}\,\boldsymbol{E} + n_0\,\boldsymbol{J}_i$$

$$= \sqrt{\varepsilon_0\,\mu_0}\ \ [\varepsilon_r\,]\frac{\partial \boldsymbol{E}}{\partial t} + n_0\sigma^{\boldsymbol{e}}\,\boldsymbol{E} + n_0\,\boldsymbol{J}_i$$

$$\nabla \times \widetilde{\boldsymbol{H}} = \frac{[\varepsilon_r]}{c_0}\,\frac{\partial \boldsymbol{E}}{\partial t} + n_0\sigma^{\boldsymbol{e}}\,\boldsymbol{E} + n_0\,\boldsymbol{J}_i \qquad (2.21)$$

$$\nabla \times \boldsymbol{E} = -\frac{[\mu]}{n_0}\,\frac{\partial \widetilde{\boldsymbol{H}}}{\partial t} - \frac{\sigma^m}{n_0}\,\widetilde{\boldsymbol{H}} - M_i\ = -[\mu_r]\,\frac{\mu_0}{\sqrt{\frac{\mu_0}{\varepsilon_0}}}\,\frac{\partial \widetilde{\boldsymbol{H}}}{\partial t} - \mathrm{n}_0\,\sigma^m\,\widetilde{\boldsymbol{H}} - M_i$$

$$\nabla \times \boldsymbol{E} = -\frac{[\mu_r]}{c_0}\,\frac{\partial \widetilde{\boldsymbol{H}}}{\partial t} - \mathrm{n}_0\,\sigma^m\,\widetilde{\boldsymbol{H}} - M_i \qquad (2.22)$$

Expanding equations (2.21) and (2.22) to get the scalar form of these two curls equations gives:

1- Expanding Equation (2.21) $\nabla \times \widetilde{\boldsymbol{H}} = \frac{[\varepsilon_r]}{c_0}\,\frac{\partial \boldsymbol{E}}{\partial t} + n_0(\sigma^{\boldsymbol{e}}\,\boldsymbol{E} + \boldsymbol{J}_i)$

Where $\nabla \times \widetilde{\boldsymbol{H}} = \begin{bmatrix} \hat{\mathbf{x}} & \hat{\mathbf{y}} & \hat{\mathbf{z}} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ \widetilde{H}_x & \widetilde{H}_y & \widetilde{H}_z \end{bmatrix}$, $[\varepsilon_r] = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{zz} \end{bmatrix}$, $[\sigma^e] = \begin{bmatrix} {\sigma^e}_{xx} & {\sigma^e}_{xy} & {\sigma^e}_{xz} \\ {\sigma^e}_{yx} & {\sigma^e}_{yy} & {\sigma^e}_{yz} \\ {\sigma^e}_{zx} & {\sigma^e}_{zy} & {\sigma^e}_{zz} \end{bmatrix}$,

$\boldsymbol{E} = \begin{bmatrix} E_x \\ E_y \\ E_z \end{bmatrix}$, $\boldsymbol{J}_i = \begin{bmatrix} J_{i_x} \\ J_{i_y} \\ J_{i_z} \end{bmatrix}$

$$\left(\frac{\partial \tilde{H}_z}{\partial y} - \frac{\partial \tilde{H}_y}{\partial z}\right)\hat{\mathrm{x}} + \left(\frac{\partial \tilde{H}_x}{\partial z} - \frac{\partial \tilde{H}_z}{\partial x}\right)\hat{\mathrm{y}} + \left(\frac{\partial \tilde{H}_y}{\partial x} - \frac{\partial \tilde{H}_x}{\partial y}\right)\hat{\mathrm{z}} = \frac{1}{c_0}\begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{zz} \end{bmatrix}\begin{bmatrix} \frac{\partial \boldsymbol{E_x}}{\partial t} \\ \frac{\partial \boldsymbol{E_y}}{\partial t} \\ \frac{\partial \boldsymbol{E_z}}{\partial t} \end{bmatrix} +$$

$$n_0 \begin{bmatrix} \sigma^{\boldsymbol{e}}{}_{xx} & \sigma^{\boldsymbol{e}}{}_{xy} & \sigma^{\boldsymbol{e}}{}_{xz} \\ \sigma^{\boldsymbol{e}}{}_{yx} & \sigma^{\boldsymbol{e}}{}_{yy} & \sigma^{\boldsymbol{e}}{}_{yz} \\ \sigma^{\boldsymbol{e}}{}_{zx} & \sigma^{\boldsymbol{e}}{}_{zy} & \sigma^{\boldsymbol{e}}{}_{zz} \end{bmatrix}\begin{bmatrix} E_x \\ E_y \\ E_z \end{bmatrix} + n_0 \begin{bmatrix} J_{i_x} \\ J_{i_y} \\ J_{i_z} \end{bmatrix}$$

$$\left(\frac{\partial \tilde{H}_z}{\partial y} - \frac{\partial \tilde{H}_y}{\partial z}\right) = \frac{1}{c_0}\left(\varepsilon_{xx}\left(\frac{\partial E_x}{\partial t}\right) + \varepsilon_{xy}\left(\frac{\partial E_y}{\partial t}\right) + \varepsilon_{xz}\left(\frac{\partial E_z}{\partial t}\right)\right) + \mathrm{n}_0\,(\,\sigma^{\boldsymbol{e}}_{xx}\,E_x + \sigma^{\boldsymbol{e}}_{xy}\,E_y + \sigma^{\boldsymbol{e}}_{xz}\,E_z + J_{ix})$$

$$\left(\frac{\partial \tilde{H}_x}{\partial z} - \frac{\partial \tilde{H}_z}{\partial x}\right) = \frac{1}{c_0}\left(\varepsilon_{yx}\left(\frac{\partial E_x}{\partial t}\right) + \varepsilon_{yy}\left(\frac{\partial E_y}{\partial t}\right) + \varepsilon_{yz}\left(\frac{\partial E_z}{\partial t}\right)\right) + \mathrm{n}_0\,(\,\sigma^{\boldsymbol{e}}_{yx}\,E_x + \sigma^{\boldsymbol{e}}_{yy}\,E_y + \sigma^{\boldsymbol{e}}_{yz}\,E_z + J_{iy})$$

$$\left(\frac{\partial \tilde{H}_y}{\partial x} - \frac{\partial \tilde{H}_x}{\partial y}\right) = \frac{1}{c_0}\left(\varepsilon_{zx}\left(\frac{\partial E_x}{\partial t}\right) + \varepsilon_{zy}\left(\frac{\partial E_y}{\partial t}\right) + \varepsilon_{zz}\left(\frac{\partial E_z}{\partial t}\right)\right) + \mathrm{n}_0\,(\,\sigma^{\boldsymbol{e}}_{zx}\,E_x + \sigma^{\boldsymbol{e}}_{zy}\,E_y + \sigma^{\boldsymbol{e}}_{zz}\,E_z + J_{iz})$$

Assuming only diagonal tensors

$$\left(\frac{\partial \tilde{H}_z}{\partial y} - \frac{\partial \tilde{H}_y}{\partial z}\right) = \frac{\varepsilon_{xx}}{c_0}\frac{\partial E_x}{\partial t} + \mathrm{n}_0\,(\,\sigma^{\boldsymbol{e}}_{xx}\,E_x + J_{ix})$$

$$\left(\frac{\partial \tilde{H}_x}{\partial z} - \frac{\partial \tilde{H}_z}{\partial x}\right) = \frac{\varepsilon_{yy}}{c_0}\frac{\partial E_y}{\partial t} + \mathrm{n}_0\,(\,\sigma^{\boldsymbol{e}}_{yy}\,E_y + J_{iy})$$

$$\left(\frac{\partial \tilde{H}_y}{\partial x} - \frac{\partial \tilde{H}_x}{\partial y}\right) = \frac{\varepsilon_{zz}}{c_0}\frac{\partial E_z}{\partial t} + \mathrm{n}_0\,(\sigma^{\boldsymbol{e}}_{zz}\,E_z + J_{iz})$$

$$\frac{\partial E_x}{\partial t} = \frac{c_0}{\varepsilon_{xx}}\left(\frac{\partial \tilde{H}_z}{\partial y} - \frac{\partial \tilde{H}_y}{\partial z} - \mathrm{n}_0\,(\,\sigma^{\boldsymbol{e}}_{xx}\,E_x + J_{ix})\right), \tag{2.23}$$

$$\frac{\partial E_y}{\partial t} = \frac{c_0}{\varepsilon_{yy}}\left(\frac{\partial \tilde{H}_x}{\partial z} - \frac{\partial \tilde{H}_z}{\partial x} - \mathrm{n}_0\,(\,\sigma^{\boldsymbol{e}}_{yy}\,E_y + J_{iy})\right), \tag{2.24}$$

$$\frac{\partial E_z}{\partial t} = \frac{c_0}{\varepsilon_{zz}}\,(\frac{\partial \tilde{H}_y}{\partial x} - \frac{\partial \tilde{H}_x}{\partial y} - \mathrm{n}_0\,(\sigma^{\boldsymbol{e}}_{zz}\,E_z + J_{iz}) \tag{2.25}$$

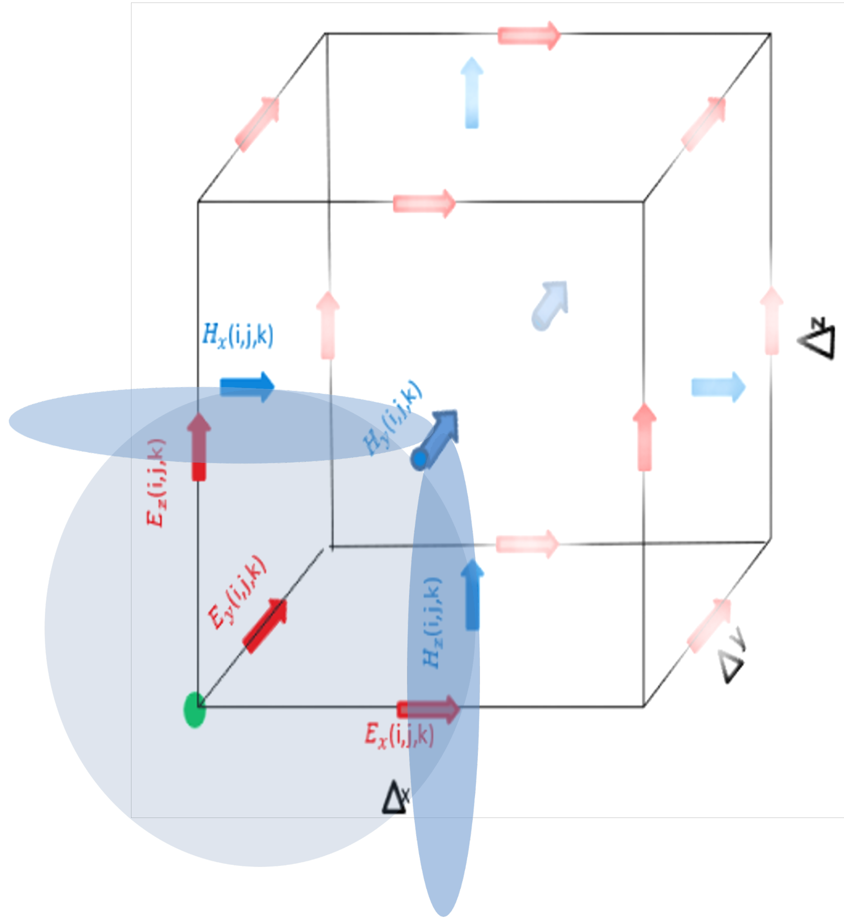


Figure 2.7. Magnetic field components on Yee's algorithm.

We will express these equations in terms of the finite central differences' formula.

$$\frac{E_x^{n+1}(i,j,k)-E_x^{n}(i,j,k)}{\Delta t} = \frac{c_0}{\varepsilon_{xx}}\left(\frac{H_z^{n+\frac{1}{2}}(i,j,k)-H_z^{n+\frac{1}{2}}(i,j-1,k)}{\Delta y} - \frac{H_y^{n+\frac{1}{2}}(i,j,k)-H_y^{n+\frac{1}{2}}(i,j,k-1)}{\Delta z} -\right.$$

$$\left. \mathrm{n}_0\, \sigma^e{}_x(i,j,k) E_x^{n+\frac{1}{2}}(i,j,k) + \mathrm{n}_0\, J_{ix}^{n+\frac{1}{2}}(i,j,k)\right) \qquad (2.26)$$

Now we need to rewrite the right-hand term in eq (2.26), $E_x^{n+\frac{1}{2}}(i,j,k)$ in such a way that is defined at integer time steps (not half steps) by taking the average of E_x^{n+1} and E_x^{n}

$$E_x^{n+\frac{1}{2}}(i,j,k) = \frac{E_x^{n+1}(i,j,k)+E_x^{n}(i,j,k)}{2} \qquad (2.27)$$

By using (2.27) in (2.26)

$$\frac{E_x^{n+1}(i,j,k)-E_x^{n}(i,j,k)}{\Delta t} + \frac{c_0\, \mathrm{n}_0\, \sigma^e{}_x(i,j,k)\, (E_x^{n+1}(i,j,k)+E_x^{n}(i,j,k)}{2\varepsilon_{xx}} = \frac{c_0}{\varepsilon_{xx}}(\frac{H_z^{n+\frac{1}{2}}(i,j,k)-H_z^{n+\frac{1}{2}}(i,j-1,k)}{\Delta y} - \frac{H_y^{n+\frac{1}{2}}(i,j,k)-H_y^{n+\frac{1}{2}}(i,j,k-1)}{\Delta z} - \mathrm{n}_0\, J_{ix}^{n+\frac{1}{2}}(i,j,k)))$$

$$\frac{2\varepsilon_{xx}\, E_x^{n+1}(i,j,k)- 2\varepsilon_{xx}\, E_x^{n}(i,j,k)+\Delta t\ \mathrm{n}_0\, \sigma^e{}_x(i,j,k)\, E_x^{n+1}(i,j,k)+\Delta t\, c_0\, \mathrm{n}_0\, \sigma^e{}_x(i,j,k)\, E_x^{n}(i,j,k)}{2\, \Delta t\, \varepsilon_{xx}} = \frac{c_0}{\varepsilon_{xx}}(\frac{H_z^{n+\frac{1}{2}}(i,j,k)-H_z^{n+\frac{1}{2}}(i,j-1,k)}{\Delta y} - \frac{H_y^{n+\frac{1}{2}}(i,j,k)-H_y^{n+\frac{1}{2}}(i,j,k-1)}{\Delta z} - \mathrm{n}_0\, J_{ix}^{n+\frac{1}{2}}(i,j,k)))$$

$$\frac{E_x^{n+1}(i,j,k)\,(2\varepsilon_{xx}+ \Delta t\, \mathrm{n}_0\, \sigma^e{}_x(i,j,k))- E_x^{n}(i,j,k)\,(\,2\varepsilon_{xx}-\Delta t\, c_0 \mathrm{n}_0\, \sigma^e{}_x(i,j,k)}{2\, \Delta t\, \varepsilon_{xx}} = \frac{c_0}{\varepsilon_{xx}}(\frac{H_z^{n+\frac{1}{2}}(i,j,k)-H_z^{n+\frac{1}{2}}(i,j-1,k)}{\Delta y} - \frac{H_y^{n+\frac{1}{2}}(i,j,k)-H_y^{n+\frac{1}{2}}(i,j,k-1)}{\Delta z} - \mathrm{n}_0\, J_{ix}^{n+\frac{1}{2}}(i,j,k))).$$

$$E_x^{n+1}(i,j,k) = E_x^{n}(i,j,k)\, \frac{2\varepsilon_{xx}- \Delta t\, c_0 \mathrm{n}_0\, \sigma^e{}_x(i,j,k)}{2\varepsilon_{xx}+ \Delta t\, \mathrm{n}_0\, \sigma^e{}_x(i,j,k)} + \frac{2c_0\Delta t}{\Delta y\,(\,2\varepsilon_{xx}+ \Delta t\, c_0 \mathrm{n}_0\, \sigma^e{}_x(i,j,k)}\,(H_z^{n+\frac{1}{2}}(i,j,k) - H_z^{n+\frac{1}{2}}(i,j-1,k)) - \frac{2c_0\Delta t}{\Delta z(\,2\varepsilon_{xx}+ \Delta t\, c_0 \mathrm{n}_0\, \sigma^e{}_x(i,j,k)}\,(H_y^{n+\frac{1}{2}}(i,j,k) - H_y^{n+\frac{1}{2}}(i,j,k-1)) - \frac{2c_0\Delta t n_0}{(\,2\varepsilon_{xx}+ \Delta t\, c_0 \mathrm{n}_0\, \sigma^e{}_x(i,j,k)}\, J_{ix}^{n+\frac{1}{2}}(i,j,k)))$$

$$E_x^{n+1}(i,j,k) = C_{exe}(i,j,k) \times E_x^{n}(i,j,k) + C_{exhz}(i,j,k) \times \left(H_z^{n+\frac{1}{2}}(i,j,k) - H_z^{n+\frac{1}{2}}(i,j-1,k)\right) - C_{exhy}(i,j,k) \times (H_y^{n+\frac{1}{2}}(i,j,k) - H_y^{n+\frac{1}{2}}(i,j,k-1)) - C_{exj}(i,j,k) \times J_{ix}^{n+\frac{1}{2}}(i,j,k) \quad (2.28a)$$

Where:

$$C_{exe}(i,j,k) = \frac{2\varepsilon_{xx}- \Delta t\, c_0 \mathrm{n}_0\, \sigma^e{}_x(i,j,k)}{2\varepsilon_{xx}+ \Delta t\, \mathrm{n}_0\, \sigma^e{}_x(i,j,k)}$$

$$C_{exhz}(i,j,k) = \frac{2c_0\Delta t}{\Delta y\,(\,2\varepsilon_{xx}+ \Delta t\, c_0 \mathrm{n}_0\, \sigma^e{}_x(i,j,k)}$$

$$C_{exhy}(i,j,k) = \frac{2c_0\Delta t}{\Delta z(\,2\varepsilon_{xx} + \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_x(i,j,k)}$$

$$C_{exj}(i,j,k) = \frac{2c_0\Delta t n_0}{2\varepsilon_{xx} +\; \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_x(i,j,k)}$$

Using the same process for the rest of the components we get:

$$E_y^{n+1}(i,j,k) = \;C_{eye}(i,j,k)\;\times\; E_y^{n}(i,j,k) +\; C_{eyhx}(i,j,k) \times \left(H_x^{n+\frac{1}{2}}(i,j,k) - H_x^{n+\frac{1}{2}}(i,j,k-1)\right)$$

$$-\,C_{eyhz}(i,j,k) \times (H_z^{n+\frac{1}{2}}(i,j,k) - H_z^{n+\frac{1}{2}}(i-1,j,k)) -\; C_{eyj}(i,j,k)\;\times\; J_{iy}^{n+\frac{1}{2}}(i,j,k) \qquad \textit{(2.28b)}$$

Where:

$$C_{eye}(i,j,k) = \frac{2\varepsilon_{yy} - \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_y(i,j,k)}{2\varepsilon_{yy} + \Delta t\; \mathrm{n}_0\; \sigma^e{}_y(i,j,k)}$$

$$C_{eyhx}(i,j,k) = \frac{2c_0\Delta t}{\Delta z\;(\,2\varepsilon_{yy} + \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_y(i,j,k)}$$

$$C_{eyhz}(i,j,k) = \frac{2c_0\Delta t}{\Delta x(\,2\varepsilon_{yy} + \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_y(i,j,k)}$$

$$C_{eyj}(i,j,k) = \frac{2c_0\Delta t n_0}{2\varepsilon_{yy} +\; \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_y(i,j,k)}$$

$$E_z^{n+1}(i,j,k) = \;C_{eze}(i,j,k)\;\times\; E_z^{n}(i,j,k) \;+ C_{ezhy}(i,j,k) \times \left(H_y^{n+\frac{1}{2}}(i,j,k) - H_y^{n+\frac{1}{2}}(i-1,j,k)\right)$$

$$-\,C_{ezhx}(i,j,k) \times (H_x^{n+\frac{1}{2}}(i,j,k) - H_x^{n+\frac{1}{2}}(i,j-1,k)) -\; C_{ezj}(i,j,k)\;\times\; J_{iz}^{n+\frac{1}{2}}(i,j,k) \qquad \textit{(2.28c)}$$

Where:

$$C_{eze}(i,j,k) = \frac{2\varepsilon_{zz} - \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_z(i,j,k)}{2\varepsilon_{zz} + \Delta t\; \mathrm{n}_0\; \sigma^e{}_z(i,j,k)}$$

$$C_{ezhy}(i,j,k) = \frac{2c_0\Delta t}{\Delta x\;(\,2\varepsilon_{zz} + \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_z(i,j,k)}$$

$$C_{exhy}(i,j,k) = \frac{2c_0\Delta t}{\Delta y(\,2\varepsilon_{zz} + \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_z(i,j,k)}$$

$$C_{exj}(i,j,k) = \frac{2c_0\Delta t n_0}{2\varepsilon_{zz} +\; \Delta t\; c_0 \mathrm{n}_0\; \sigma^e{}_z(i,j,k)}$$

2- Expanding Equation (2.22) $\nabla \times \boldsymbol{E} = -\frac{[\mu_r]}{c_0}\;\frac{\partial \widetilde{\boldsymbol{H}}}{\partial t} - \;\mathrm{n}_0\; \sigma^m\; \widetilde{\boldsymbol{H}} - \;M_i$

$$\nabla \times \boldsymbol{E} = \begin{bmatrix} \hat{\mathbf{x}} & \hat{\mathbf{y}} & \hat{\mathbf{z}} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ E_x & E_y & E_z \end{bmatrix}, \ [\mu_r] = \begin{bmatrix} \mu_{xx} & \mu_{xy} & \mu_{xz} \\ \mu_{yx} & \mu_{yy} & \mu_{yz} \\ \mu_{zx} & \mu_{zy} & \mu_{zz} \end{bmatrix}, \ \widetilde{\boldsymbol{H}} = \begin{bmatrix} \widetilde{H}_x \\ \widetilde{H}_y \\ \widetilde{H}_z \end{bmatrix}$$

$$\left(\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z}\right)\hat{\mathrm{x}} + \left(\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x}\right)\hat{\mathrm{y}} + \left(\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y}\right)\hat{\mathrm{z}} = \frac{-1}{c_0}\begin{bmatrix} \mu_{xx} & \mu_{xy} & \mu_{xz} \\ \mu_{yx} & \mu_{yy} & \mu_{yz} \\ \mu_{zx} & \mu_{zy} & \mu_{zz} \end{bmatrix}\begin{bmatrix} \frac{\partial \widetilde{H}_x}{\partial t} \\ \frac{\partial \widetilde{H}_y}{\partial t} \\ \frac{\partial \widetilde{H}_z}{\partial t} \end{bmatrix} - \mathrm{n}_0\, \sigma^m\, \widetilde{\boldsymbol{H}} - M_i$$

$$\left(\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z}\right) = \frac{-1}{c_0}\left(\mu_{xx}\left(\frac{\partial \widetilde{H}_x}{\partial t}\right) + \mu_{xy}\left(\frac{\partial \widetilde{H}_y}{\partial t}\right) + \mu_{xz}\left(\frac{\partial \widetilde{H}_z}{\partial t}\right)\right) - \mathrm{n}_0\, \sigma_x^m\, \widetilde{H}_x - M_{ix}$$

$$\left(\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x}\right) = \frac{-1}{c_0}\left(\mu_{yx}\left(\frac{\partial \widetilde{H}_x}{\partial t}\right) + \mu_{yy}\left(\frac{\partial \widetilde{H}_y}{\partial t}\right) + \mu_{yz}\left(\frac{\partial \widetilde{H}_z}{\partial t}\right)\right) - \mathrm{n}_0\, \sigma_y^m\, \widetilde{H}_y - M_{iy}$$

$$\left(\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y}\right) = \frac{-1}{c_0}\left(\mu_{zx}\left(\frac{\partial \widetilde{H}_x}{\partial t}\right) + \mu_{zy}\left(\frac{\partial \widetilde{H}_y}{\partial t}\right) + \mu_{zz}\left(\frac{\partial \widetilde{H}_z}{\partial t}\right)\right) - \mathrm{n}_0\, \sigma_z^m\, \widetilde{H}_z - M_{iz}$$

Assuming only diagonal tensors

$$\left(\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z}\right) = -\frac{\mu_{xx}}{c_0}\frac{\partial \widetilde{H}_x}{\partial t} - \mathrm{n}_0\, \sigma_x^m\, \widetilde{H}_x - M_{ix}$$

$$\left(\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x}\right) = -\frac{\mu_{yy}}{c_0}\frac{\partial \widetilde{H}_y}{\partial t} - \mathrm{n}_0\, \sigma_y^m\, \widetilde{H}_y - M_{iy}$$

$$\left(\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y}\right) = -\frac{\mu_{zz}}{c_0}\frac{\partial \widetilde{H}_z}{\partial t} - \mathrm{n}_0\, \sigma_z^m\, \widetilde{H}_z - M_{iz}$$

$$\frac{\partial \widetilde{H}_x}{\partial t} = \frac{c_0}{\mu_{xx}}\left(\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} - \mathrm{n}_0\, \sigma_x^m\, \widetilde{H}_x - M_{ix}\right) \quad (2.29)$$

$$\frac{\partial \widetilde{H}_y}{\partial t} = \frac{c_0}{\mu_{yy}}\left(\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z} - \mathrm{n}_0\, \sigma_y^m\, \widetilde{H}_y - M_{iy}\right) \quad (2.30)$$

$$\frac{\partial \widetilde{H}_z}{\partial t} = \frac{c_0}{\mu_{zz}}\left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} - \mathrm{n}_0\, \sigma_z^m\, \widetilde{H}_z - M_{iz}\right) \quad (2.31)$$

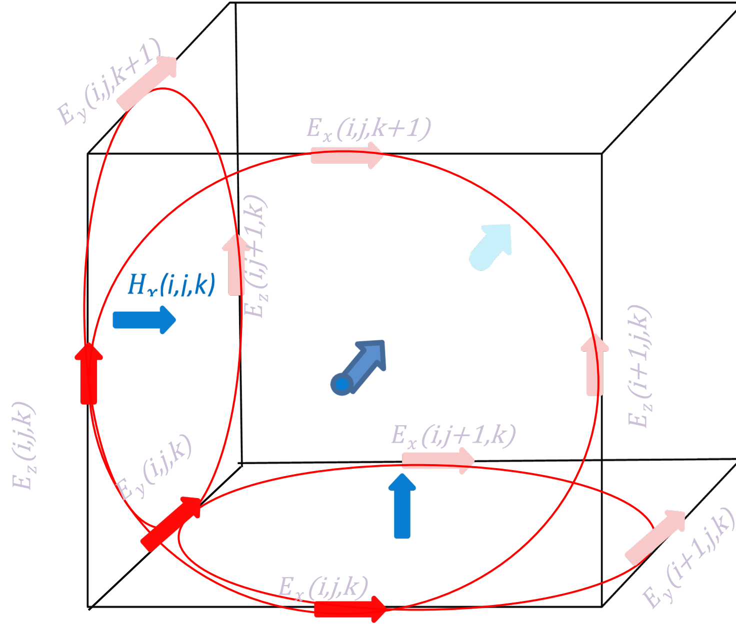


Figure 2.8. Arrangement of field's components on a (3-D) Yee's cell indexed at (*i*, *j*, *k*)

The finite central differences form of the equations will be:

$$H_x^{n+1/2}(i,j,k) = C_{hxh}(i,j,k) \times H_x^{n-1/2}(i,j,k) + C_{hxey}(i,j,k) \times \left(E_y^n(i,j,k+1) - E_y^n(i,j,k)\right)$$
$$- C_{hxez}(i,j,k) \times (E_z^n(i,j+1,k) - E_z^n(i,j,k)) - C_{hxm}(i,j,k) \times M_{ix}^n(i,j,k) \qquad (2.32a)$$

Where:

$$C_{hxh}(i,j,k) = \frac{2\mu_{xx}(i,j,k) - \Delta t\, \sigma^m{}_x(i,j,k)}{2\mu_x(i,j,k) + \Delta t\, \sigma^m{}_x(i,j,k)}$$

$$C_{hxey}(i,j,k) = \frac{2c_0\Delta t}{(2\mu_x(i,j,k) + \Delta t\, \sigma^m{}_x(i,j,k))\Delta z}$$

$$C_{hxez}(i,j,k) = \frac{2c_0\Delta t}{(2\mu_x(i,j,k) + \Delta t\, \sigma^m{}_x(i,j,k))\Delta y}$$

$$C_{hxm}(i,j,k) = \frac{2c_0\Delta t n_0}{2\mu_x(i,j,k) + \Delta t\, \sigma^m{}_x(i,j,k)}$$

$$H_y^{n+\frac{1}{2}}(i,j,k) = C_{hyh}(i,j,k) \times H_y^{n-\frac{1}{2}}(i,j,k) + C_{hyez}(i,j,k) \times \left(E_z^n(i+1,j,k) - E_z^n(i,j,k)\right)$$
$$- C_{hyex}(i,j,k) \times (E_x^n(i,j,k+1) - E_x^n(i,j,k)) - C_{hym}(i,j,k) \times M_{iy}^n(i,j,k) \qquad (2.32b)$$

Where:

$$C_{hyh}(i,j,k) = \frac{2\mu_y(i,j,k) - \Delta t\, \sigma^m{}_y(i,j,k)}{2\mu_y(i,j,k) + \Delta t\, \sigma^m{}_y(i,j,k)} \;,$$

$$C_{hyez}(i,j,k) = \frac{2c_0\Delta t}{(2\mu_y(i,j,k) + \Delta t\, \sigma^m{}_y(i,j,k))\Delta x},$$

$$C_{hyex}(i,j,k) = \frac{2c_0\Delta t}{(2\mu_y(i,j,k) + \Delta t\, \sigma^m{}_y(i,j,k))\Delta z},$$

$$C_{hym}(i,j,k) = \frac{2c_0\Delta t n_0}{2\mu_y(i,j,k) + \Delta t\, \sigma^m{}_y(i,j,k)} .$$

$$H_z^{n+\frac{1}{2}}(i,j,k) = C_{hzh}(i,j,k) \times H_z^{n-\frac{1}{2}}(i,j,k) + C_{hzex}(i,j,k) \times \left(E_x^n(i,j+1,k) - E_x^n(i,j,k)\right)$$
$$- C_{hzey}(i,j,k) \times (E_y^n(i+1,j,k) - E_y^n(i,j,k)) - C_{hzm}(i,j,k) \times M_{iz}^n(i,j,k) \qquad (2.32c)$$

Where:

$$C_{hzh}(i,j,k) = \frac{2\mu_z(i,j,k) - \Delta t\, \sigma^m{}_z(i,j,k)}{2\mu_z(i,j,k) + \Delta t\, \sigma^m{}_z(i,j,k)},$$

$$C_{hzex}(i,j,k) = \frac{2c_0\Delta t}{(2\mu_z(i,j,k) + \Delta t\, \sigma^m{}_z(i,j,k))\Delta y},$$

$$C_{hzey}(i,j,k) = \frac{2c_0\Delta t}{(2\mu_z(i,j,k) + \Delta t\, \sigma^m{}_z(i,j,k))\Delta x},$$

$$C_{hzm}(i,j,k) = \frac{2c_0\Delta t n_0}{2\mu_z(i,j,k) + \Delta t\, \sigma^m{}_z(i,j,k)} .$$

For many of FDTD problems the electric and magnetic currents do not exist (**J**, and **M** = 0). In this case Equations (2.28) and (2.32) can be simplified to:

$$E_x^{n+1}(i,j,k) = E_x^n(i,j,k) + \frac{c_0\Delta t}{\varepsilon_{xx}}\left(\frac{H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i,j-1,k)}{\Delta y} - \frac{H_y^{n+1/2}(i,j,k) - H_y^{n+1/2}(i,j,k-1)}{\Delta z}\right) \qquad (2.33a)$$

$$E_y^{n+1}(i,j,k) = E_y^n(i,j,k) + \frac{c_0\Delta t}{\varepsilon_{yy}}\left(\frac{H_x^{n+1/2}(i,j,k) - H_x^{n+1/2}(i,j,k-1)}{\Delta z} - \frac{H_z^{n+1/2}(i,j,k) - H_z^{n+1/2}(i-1,j,k)}{\Delta x}\right) \qquad (2.33b)$$

$$E_z^{n+1}(i,j,k) = E_z^n(i,j,k) + \frac{c_0\Delta t}{\varepsilon_{zz}}\left(\frac{H_y^{n+1/2}(i,j,k) - H_y^{n+1/2}(i-1,j,k)}{\Delta x} - \frac{H_x^{n+1/2}(i,j,k) - H_x^{n+1/2}(i,j-1,k)}{\Delta y}\right) \qquad (2.33c)$$

$$H_x^{n+1/2}(i,j,k) = H_x^{n-1/2}(i,j,k) + \frac{c_0\Delta t}{\mu_{xx}}\left(\frac{E_y^n(i,j,k+1) - E_y^n(i,j,k)}{\Delta z} - \frac{E_z^n(i,j+1,k) - E_z^n(i,j,k)}{\Delta y}\right) \qquad (2.34a)$$

$$H_y^{n+1/2}(i,j,k) = H_y^{n-1/2}(i,j,k) + \frac{c_0\Delta t}{\mu_{yy}}\left(\frac{E_z^n(i+1,j,k) - E_z^n(i,j,k)}{\Delta x} - \frac{E_x^n(i,j,k+1) - E_x^n(i,j,k)}{\Delta z}\right) \qquad (2.34b)$$

$$H_z^{n+1/2}(i,j,k) = H_z^{n-1/2}(i,j,k) + \frac{c_0\Delta t}{\mu_{zz}}\left(\frac{E_x^n(i,j+1,k) - E_x^n(i,j,k)}{\Delta y} - \frac{E_y^n(i+1,j,k) - E_y^n(i,j,k)}{\Delta x}\right) \qquad (2.34c)$$

2.4.2. Two-Dimensional Problems

Sometimes, the electromagnetic problem we need to simulate can be described by using only two dimensions so the material and fields are uniform in one direction and the derivatives in that direction will be zero. Let $\frac{\partial}{\partial z} = 0$, then Maxwell's equations will be:

$$E_x^{n+1}(i,j) = E_x^n(i,j) + \frac{c_0\Delta t}{\varepsilon_{xx}}\left(\frac{H_z^{n+1/2}(i,j)-H_z^{n+1/2}(i,j-1)}{\Delta y}\right) \quad (2.35a)$$

$$E_y^{n+1}(i,j) = E_y^n(i,j) - \frac{H_z^{n+1/2}(i,j)-H_z^{n+1/2}(i-1,j)}{\Delta x} \quad (2.35b)$$

$$H_z^{n+1/2}(i,j) = H_z^{n-1/2}(i,j) + \frac{c_0\Delta t}{\mu_{zz}}\left(\frac{E_x^n(i,j+1)-E_x^n(i,j)}{\Delta y} - \frac{E_y^n(i+1,j)-E_y^n(i,j)}{\Delta x}\right) \quad (2.35c)$$

$$E_z^{n+1}(i,j) = E_z^n(i,j) + \frac{c_0\Delta t}{\varepsilon_{zz}}\left(\frac{H_y^{n+1/2}(i,j)-H_y^{n+1/2}(i-1,j)}{\Delta x} - \frac{H_x^{n+1/2}(i,j)-H_x^{n+1/2}(i,j-1)}{\Delta y}\right) \quad (2.36a)$$

$$H_x^{n+1/2}(i,j) = H_x^{n-1/2}(i,j) - \frac{c_0\Delta t}{\mu_{xx}}\frac{E_z^n(i,j+1)-E_z^n(i,j)}{\Delta y} \quad (2.36b)$$

$$H_y^{n+1/2}(i,j) = H_y^{n-1/2}(i,j) + \frac{c_0\Delta t}{\mu_{yy}}\frac{E_z^n(i+1,j)-E_z^n(i,j)}{\Delta x} \quad (2.36c)$$

Notice that Equations (2.35a), (2.35b), and (2.35c) and Equations (2.36a), (2.36b), and (2.36c) have decoupled into two sets (modes): 1- E_z- mode and 2- H_z-mode see Figures (2.9.) – (2.13.)

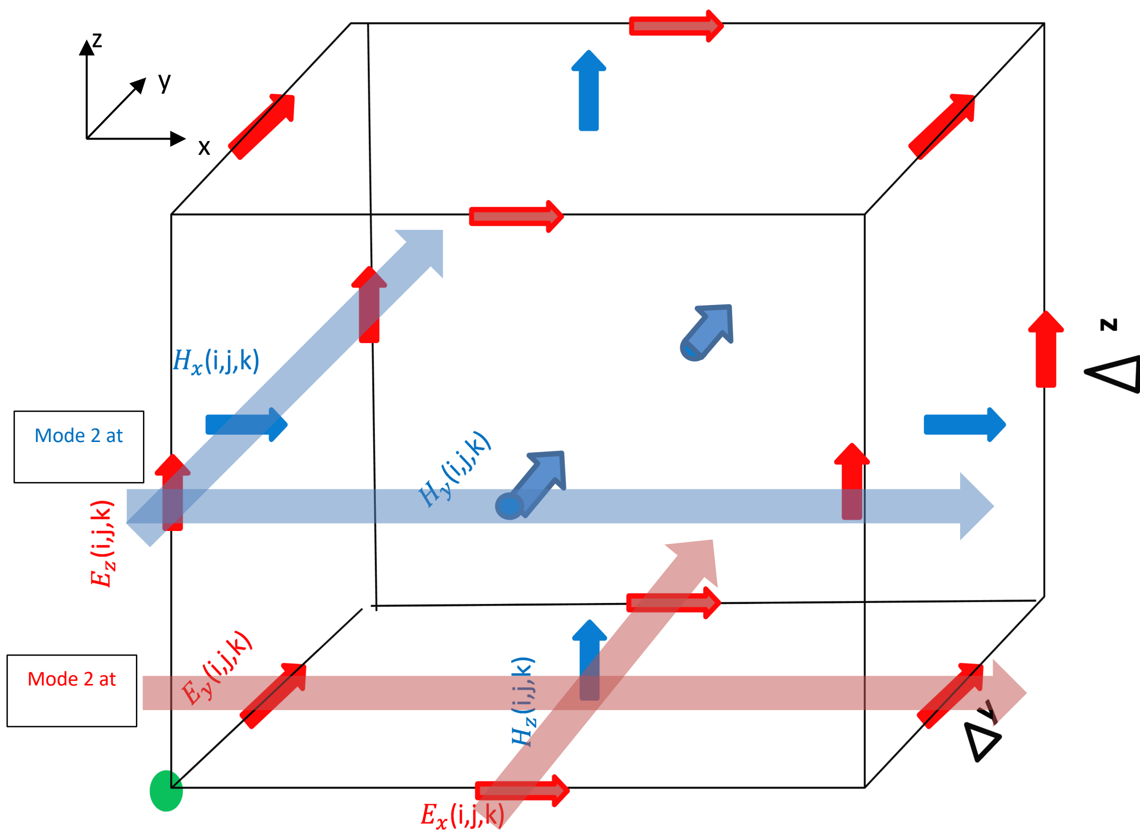


Figure 2.9. Unit cell Indexed as (*i*, *j*, *k*) Showing the Two Modes of Wave Propagation

Mode 1: when z =0

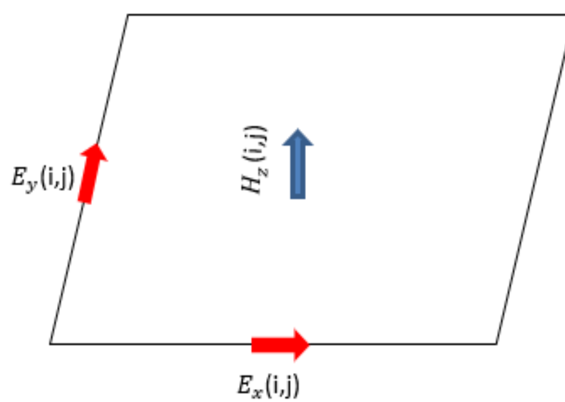


Figure 2.10. $\boldsymbol{H_z}$-Mode Sliced figure (2.9.) at z=0

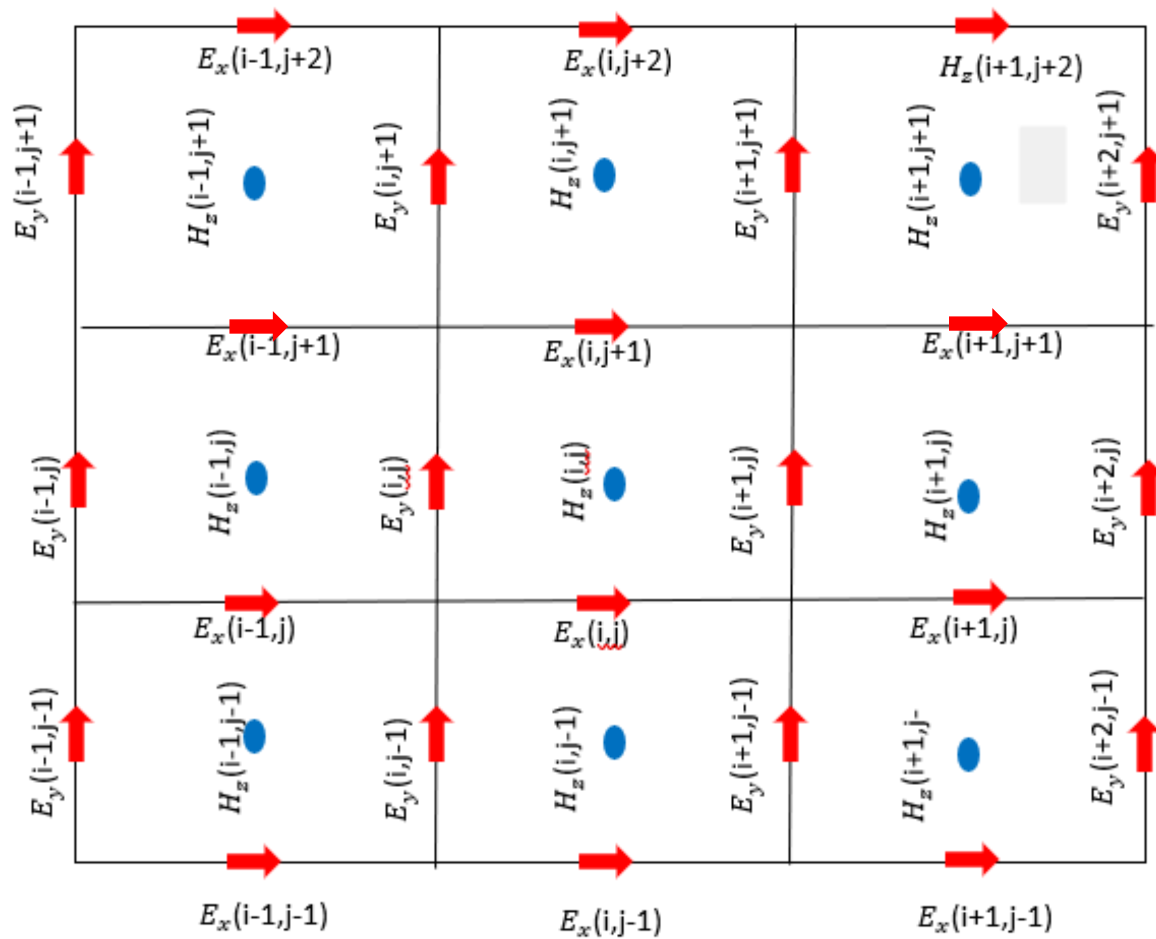


Figure 2.11. 2D-FDTD $\boldsymbol{H_z}$- Mode

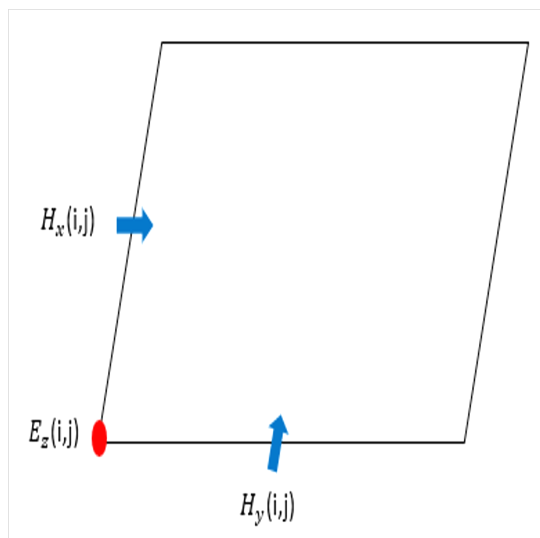


Figure 2.12. $\boldsymbol{E_z}$- Mode Sliced figure (2.9.) at z=0.5

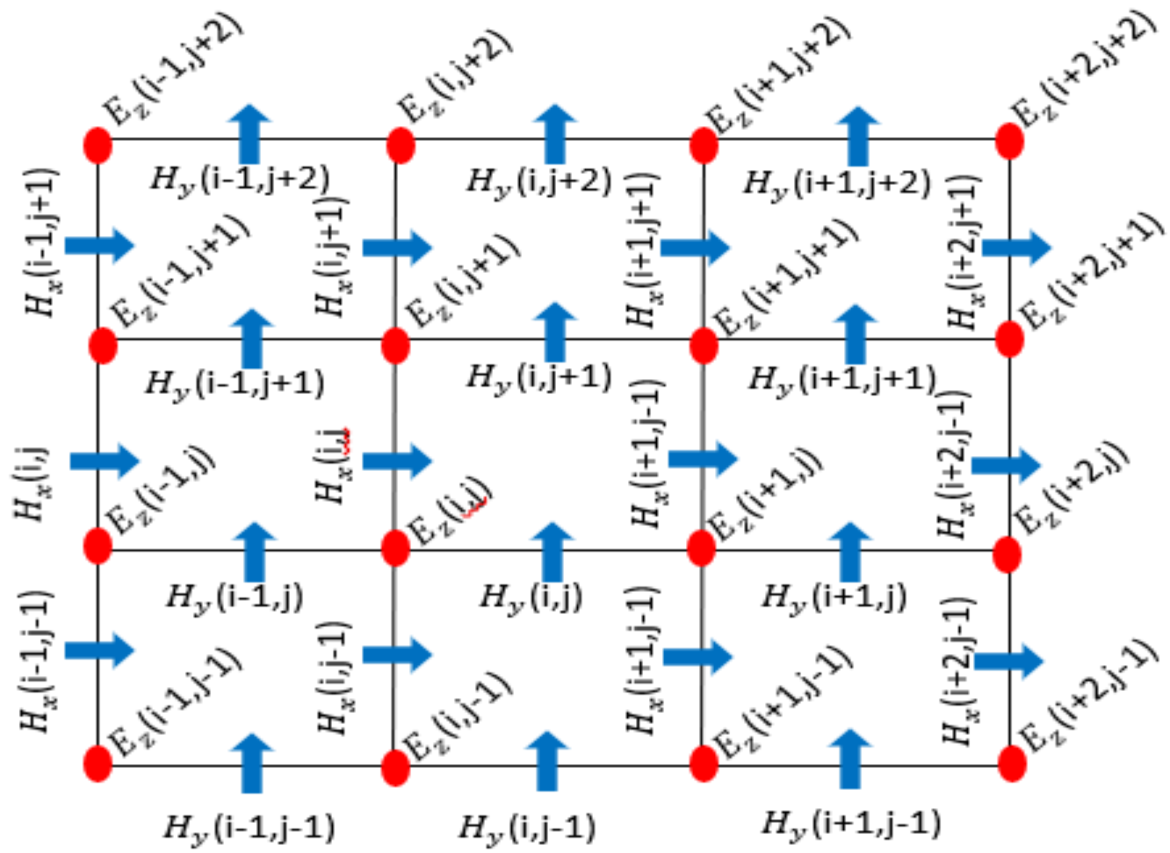


Figure 2.13. 2D-FDTD $\boldsymbol{E_z}$-Mode

2.4.3. One-Dimensional Problems

For one-dimensional problems, $\frac{\partial}{\partial z}, \frac{\partial}{\partial y} = 0$, and we have two sets (modes): E_z/H_y and E_y/H_z.

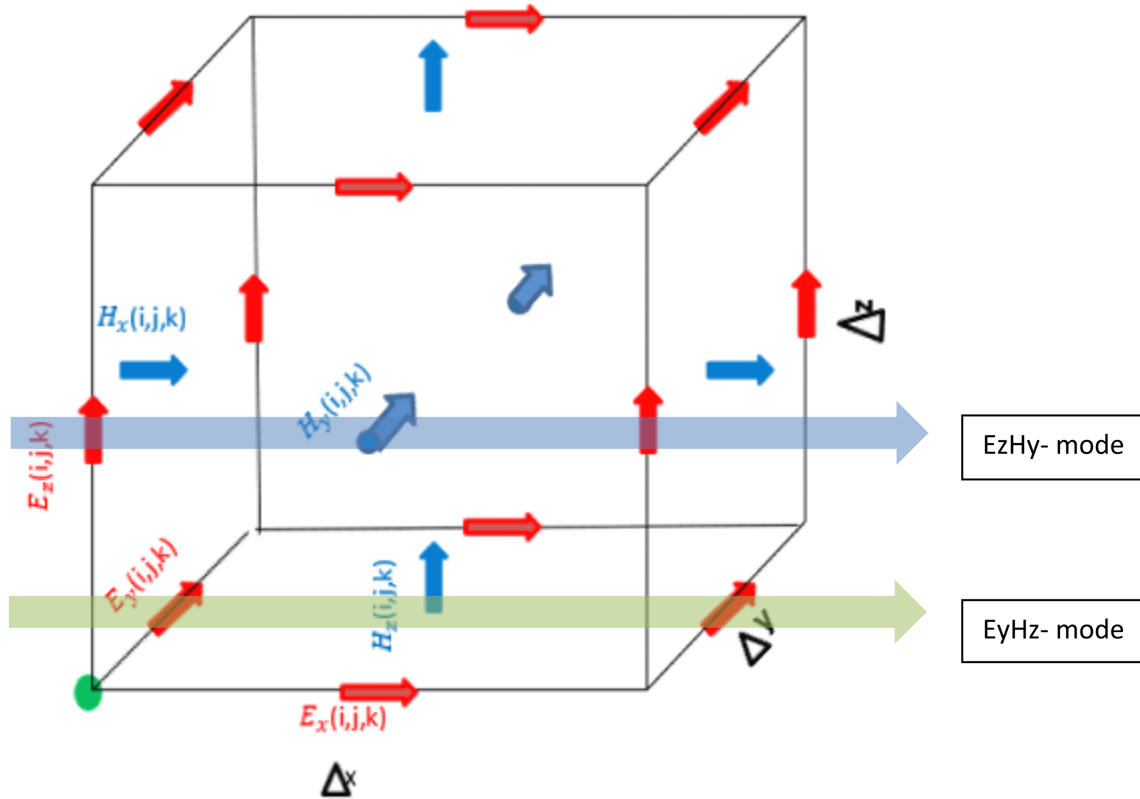


Figure 2.14. Unit cell Indexed as (i, j, k) Showing the Two Modes of Propagation

Mode 1:

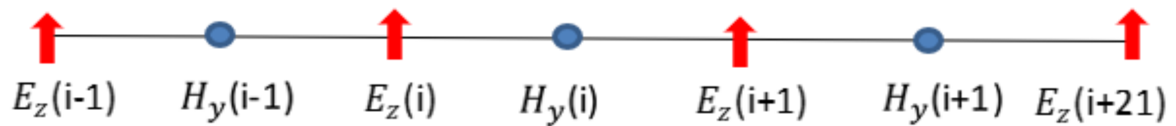


Figure 2.15. One-Dimensional FDTD-Mode 1

$$E_z^{n+1}(i) = E_z^n(i) + \frac{c_0 \Delta t}{\varepsilon_{zz}} \frac{H_y^{n+1/2}(i) - H_y^{n+1/2}(i-1)}{\Delta x}$$

$$H_y^{n+1/2}(i) = H_y^{n-1/2}(i) + \frac{c_0 \Delta t}{\mu_{yy}} \frac{E_z^n(i+1) - E_z^n(i)}{\Delta x}$$

Mode 2

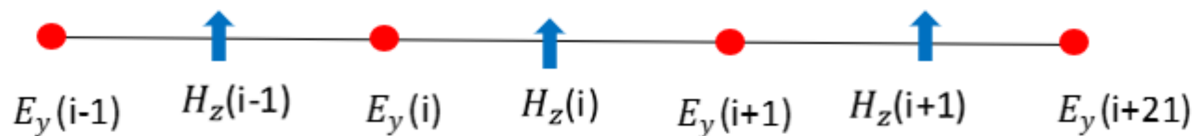


Figure 2.16. One-Dimensional FDTD-Mode 2

$$E_y^{n+1}(i) = E_y^n(i) - \frac{H_z^{n+1/2}(i) - H_z^{n+1/2}(i-1)}{\Delta x}$$

$$H_z^{n+1/2}(i) = H_z^{n-1/2}(i) + \frac{c_0 \Delta t}{\mu_{zz}} \frac{E_y^n(i+1) - E_y^n(i)}{\Delta x}$$

2.5. Conclusion

Finite-Difference Time-Domain (FDTD) method was used to find the solutions of Maxwell's equations by approximating the time and space partial derivatives with finite central differences. To apply this method, the components of electrical and magnetic fields need to be staggered relative to each other in time and space. Either the E or the H fields may stagger on half time and space step offset from each other. The complexity of the formulation of the FDTD equations is because the components of the fields are physically located in various locations in time and space, and care needs to be taken when approximating them. Additionally, these components may reside in different materials even when they are in the same Yee unit cell. The simplified process to drive the FDTD equations (update equations) is shown in figure (2.17.) for one, two, and three dimensions:

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\mathbf{B} = [\mu] * \mathbf{H} * \frac{\eta_0}{\eta_0}$$

$$\mathbf{H} = \frac{\tilde{\mathbf{H}}}{\eta_0}, \quad n_0 = \sqrt{\frac{\mu_0}{\varepsilon_0}}$$

$$[\mu] = \mu_0 [\mu_r]$$

$$c_0 = \frac{1}{\sqrt{\varepsilon_0 \mu_0}}$$

$$\mathbf{B} = \mu_0 [\mu_r] * \frac{\tilde{\mathbf{H}}}{\eta_0}$$

$$-\frac{\partial \mathbf{B}}{\partial t} = \frac{-[\mu_r]}{c_0} \frac{\partial \tilde{\mathbf{H}}}{\partial t}$$

$$\mathbf{D} = [\varepsilon] * \mathbf{E}$$

$$\frac{\partial \mathbf{D}}{\partial t} = \frac{[\varepsilon_r]}{c_0} \frac{\partial \mathbf{E}}{\partial t}$$

$$\left(\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z}\right) = \frac{-1}{c_0} \left(\mu_{xx} \left(\frac{\partial \tilde{H}_x}{\partial t}\right) + \mu_{xy} \cancel{\left(\frac{\partial \tilde{H}_y}{\partial t}\right)} + \mu_{xz} \cancel{\left(\frac{\partial \tilde{H}_z}{\partial t}\right)} \right)$$

$$\left(\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x}\right) = \frac{-1}{c_0} \left(\mu_{yx} \cancel{\left(\frac{\partial \tilde{H}_x}{\partial t}\right)} + \mu_{yy} \left(\frac{\partial \tilde{H}_y}{\partial t}\right) + \mu_{yz} \cancel{\left(\frac{\partial \tilde{H}_z}{\partial t}\right)} \right)$$

$$\left(\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y}\right) = \frac{-1}{c_0} \left(\mu_{zx} \cancel{\left(\frac{\partial \tilde{H}_x}{\partial t}\right)} + \mu_{zy} \cancel{\left(\frac{\partial \tilde{H}_y}{\partial t}\right)} + \mu_{zz} \left(\frac{\partial \tilde{H}_z}{\partial t}\right) \right)$$

$$\frac{\partial \tilde{H}_z}{\partial y} - \frac{\partial \tilde{H}_y}{\partial z} = \frac{-1}{c_0} \left(\varepsilon_{xx} \left(\frac{\partial E_x}{\partial t}\right) + \varepsilon_{xy} \cancel{\left(\frac{\partial E_y}{\partial t}\right)} + \varepsilon_{xz} \cancel{\left(\frac{\partial E_z}{\partial t}\right)} \right)$$

$$\frac{\partial \tilde{H}_x}{\partial z} - \frac{\partial \tilde{H}_z}{\partial x} = \frac{-1}{c_0} \left(\varepsilon_{yx} \cancel{\left(\frac{\partial E_x}{\partial t}\right)} + \varepsilon_{yy} \left(\frac{\partial E_y}{\partial t}\right) + \varepsilon_{yz} \cancel{\left(\frac{\partial E_z}{\partial t}\right)} \right)$$

$$\frac{\partial \tilde{H}_y}{\partial x} - \frac{\partial \tilde{H}_x}{\partial y} = \frac{-1}{c_0} \left(\varepsilon_{zx} \cancel{\left(\frac{\partial E_x}{\partial t}\right)} + \varepsilon_{zy} \cancel{\left(\frac{\partial E_y}{\partial t}\right)} + \varepsilon_{zz} \left(\frac{\partial E_z}{\partial t}\right) \right)$$

$$\left(\frac{\partial E_z}{\partial y}-\frac{\partial E_y}{\partial z}\right)=\frac{-1}{c_0}\left(\mu_{xx}\left(\frac{\partial \widetilde{H}_x}{\partial t}\right)\right) \qquad E_x = E_x + \frac{c_0 \Delta t}{\varepsilon_{xx}}\left(\frac{H_z(j)-H_z(j-1)}{\Delta y}-\frac{H_y(k)-H_y(k-1)}{\Delta z}\right)$$

$$\left(\frac{\partial E_x}{\partial z}-\frac{\partial E_z}{\partial x}\right)=\frac{-1}{c_0}\left(\mu_{yy}\left(\frac{\partial \widetilde{H}_y}{\partial t}\right)\right) \qquad E_x = E_x + \frac{c_0 \Delta t}{\varepsilon_{yy}}\left(\frac{H_x(k)-H_x(k-1)}{\Delta z}-\frac{H_z(i)-H_z(i-1)}{\Delta x}\right)$$

$$\left(\frac{\partial E_y}{\partial x}-\frac{\partial E_x}{\partial y}\right)=\frac{-1}{c_0}\left(\mu_{zz}\left(\frac{\partial \widetilde{H}_z}{\partial t}\right)\right) \qquad E_x = E_x + \frac{c_0 \Delta t}{\varepsilon_{zz}}\left(\frac{H_y(i)-H_y(i-1)}{\Delta x}-\frac{H_x(j)-H_x(j-1)}{\Delta y}\right)$$

$$\frac{\partial \widetilde{H}_z}{\partial y}-\frac{\partial \widetilde{H}_y}{\partial z}=\frac{-\varepsilon_{xx}}{c_0}\frac{\partial E_x}{\partial t} \qquad H_x = H_x + \frac{c_0 \Delta t}{\mu_{xx}}\left(\frac{E_y(k+1)-E_y(k)}{\Delta z}-\frac{E_z(j+1)-E_z(j)}{\Delta y}\right)$$

$$\frac{\partial \widetilde{H}_x}{\partial z}-\frac{\partial \widetilde{H}_z}{\partial x}=\frac{-\varepsilon_{yy}}{c_0}\frac{\partial E_y}{\partial t} \qquad H_y = H_y + \frac{c_0 \Delta t}{\mu_{yy}}\left(\frac{E_z(i+1)-E_z(i)}{\Delta x}-\frac{E_x(k+1)-E_x(k)}{\Delta z}\right)$$

$$\frac{\partial \widetilde{H}_y}{\partial x}-\frac{\partial \widetilde{H}_x}{\partial y}=\frac{-\varepsilon_{zz}}{c_0}\frac{\partial E_z}{\partial t} \qquad H_z = H_z + \frac{c_0 \Delta t}{\mu_{zz}}\left(\frac{E_x(j+1)-E_x(j)}{\Delta y}-\frac{E_y(i+1)-E_y(i)}{\Delta x}\right)$$

Figure 2.17. Summary of the Formulation of the FDTD equations.

For one or two-dimensional cases, when the derivative is not changed in one axis the derivatives in figure (2.17.) is set to zero.

CHAPTER 3

3. HARDWARE ACCELERATION OF FDTD METHOD – CUDA

3.1. Introduction

The FDTD algorithm can be easily divided into independent tasks that can be executed in parallel. For example, the update equation of the components E_x is not depending on the values of the E_y *and* E_z. Similarly, to update the E_y *and* E_z we only need the magnitude of the ***H*** field (H_x, H_y, *and* H_z) and previous values of themselves. Thus, the update equations of E_x, E_y, *and* E_z can be executed simultaneously in parallel without any contention. Similarly, the components H_x, H_y, *and Hz of* the magnetic field can be computed independently of one another, see Figures (3.1.) and (3.2.). That is one reason the FDTD method is preferred by the computational electromagnetic developer community.

The FDTD problem can be divided into small programs that are independent and distributed into blocks of threads that can be executed concurrently in parallel.

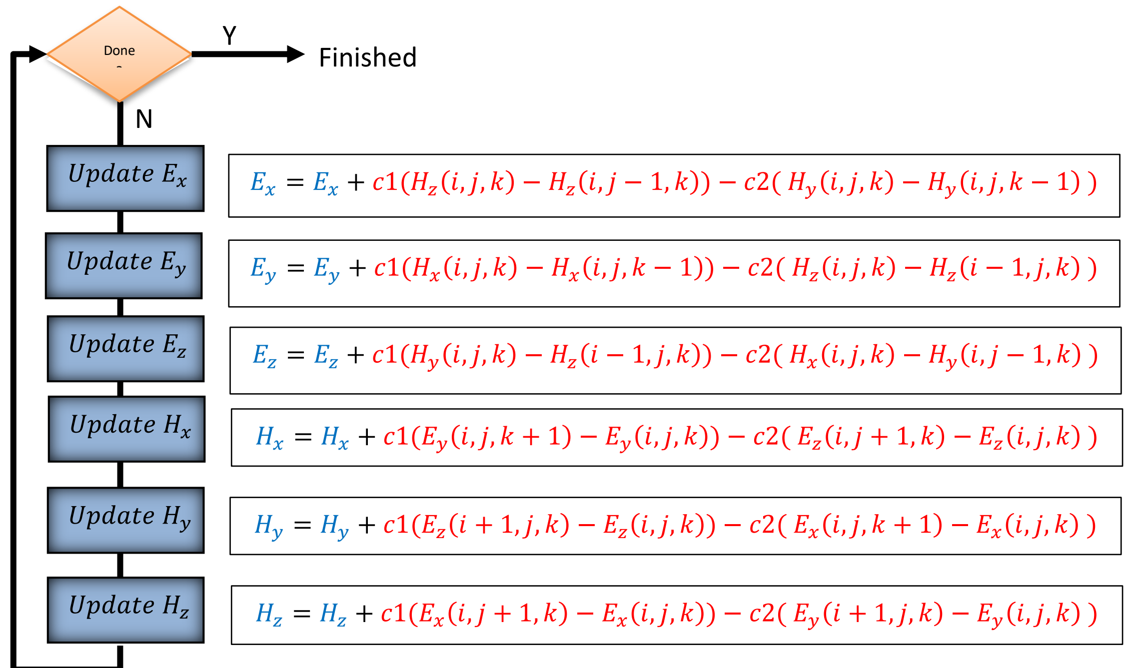


Figure 3.1. FDTD Equations in serial

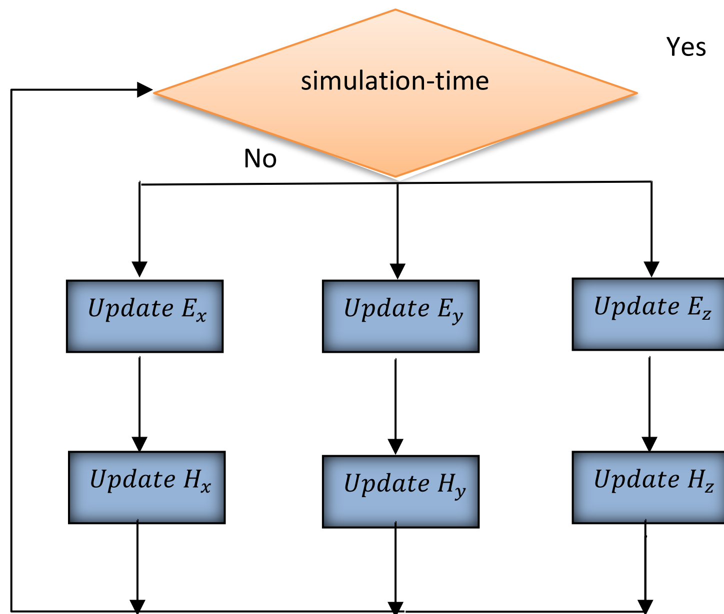


Figure 3.2. FDTD Updated Equations in Parallel

3.2. GPU Programming using CUDA

The graphics processing unit (GPU) was initially designed in 1999 by NVIDIA as a specialized processor to accelerate graphics rendering. However, with massively parallel matrix-processing capabilities, it was later realized that GPU's floating-point performance and memory bandwidth were much higher in performance in data parallel applications than the CPU.

Although a GPU's single thread executing time is often much slower than that of a CPU, GPUs can often achieve much higher performance when executing parallel applications due to the way that the GPU is designed with more transistors for data processing and less data caching and flow control [10], see Figure (3.3.). For FDTD applications, the programmer can collapse "for loop" constructs into thousands of independent threads that the GPUs can process simultaneously,

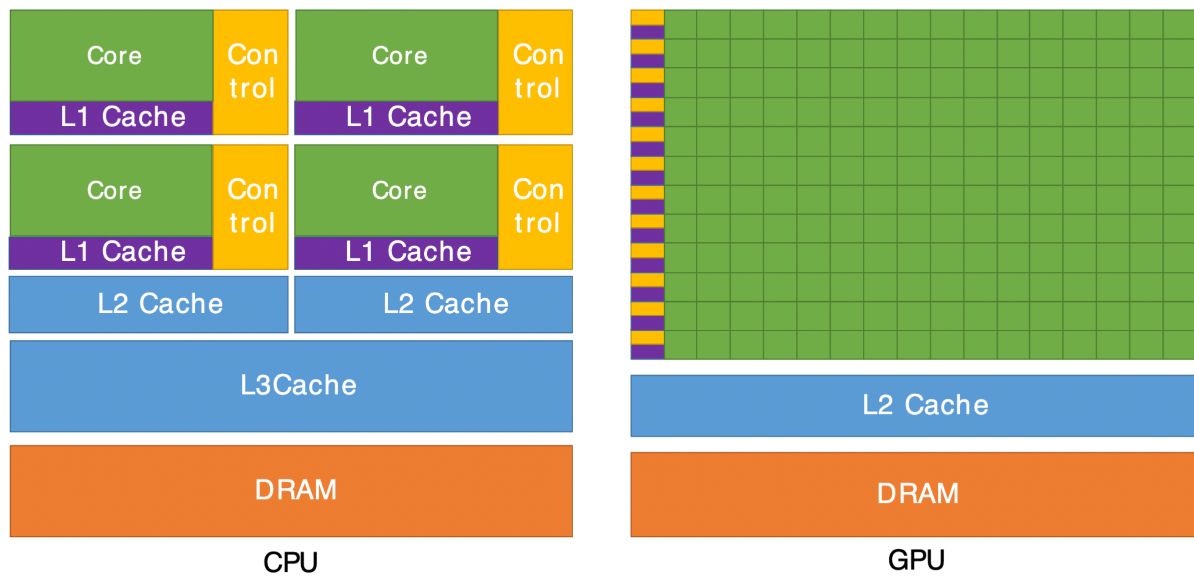


Figure 3.3. Distributions of chip resources for CPU versus a GPU [10]

3.2.1. Compute Unified Device Architecture

GPUs are becoming more powerful with an increase in the number of processors and memory associated with each processor. To take the advantage of these increased abilities, especially in computing applications, it is important to have software that can use all the GPU's processors in an efficient way as compared to the CPU. Additionally, this new software should be easy enough for programmers that are familiar with high level languages, like C, C++, Fortran, etc.

NVIDIA, In November 2006, produced a general-purpose parallel computing platform and programming model called Compute Unified Device Architecture (CUDA). This supports both high-level languages such as C and directives-based languages like FORTRAN [9].

The CPU program launches the kernel grid along with execution configurations to run in parallel. The blocks of the grid are divided and distributed to be executed on multithreaded streaming multiprocessors (SMs) of the NVIDIA GPU. In the case of NVIDIA Quadro P400 (256 core), two SMs each consist of 128 cores for arithmetic operations and 4 warp schedulers. SM with a single instruction multiple threads (SIMT) architecture will receive one or more blocks, group the block's

threads into *warps*[1] (One warp has 32 parallel threads) and assign warps equally among four warp schedulers. When the instruction issue time comes, the scheduler assigns the warp to be executed [10].

3.2.2. Programming Model

3.2.2.1 CUDA Kernel

In C, functions when called are run sequentially, one by one on the CPU. In CUDA Kernels are executed in parallel on different GPU threads [*14*]. CUDA GPUs (in CUDA the GPU is called the device while the CPU is called the host) run kernels using blocks of threads that are multiple of 32 in size. The user can define the kernel by using the __global__ declaration specifier. The programmer also can tell the CUDA runtime how many (N) parallel threads are needed to use for the launch on the CUDA device using execution configuration like the example below:

Let the kernel's function be declared as:

__global__ void ADD (float* A[N], float* B[N], float* C[N])

To call the kernel we use:

Kernel's name (in this case, ADD) the triple angle bracket syntax<<<….>>> (parameter list): i.e.,

ADD<<<Dg, Db, Ns, S>>> (A, B, C)

Where *Dg* has the type of an integer vector (3d) based on unit3 used to specify grid dimensions gridDim.x * gridDim.y * gridDim.z equals the number of blocks being launched.

Db has the type of an integer vector (3d) based on unit3 used to specify block dimensions blockDim.x * blockDim.y * blockDim.z equals the number of threads per block.

Ns is an optional argument which defaults to 0[14].

---

[1] number of warps = $ceil\ (\frac{number\ of\ threads\ per\ block}{warp\ size = 32}, 1)$[10].

S is of type cudaStream_t and is used to specify the associated stream. This is also optional with a default of 0.

In addition to the built-in variables mentioned above (gridDim, and blockDim), CUDA also has two more built-in variables for the block and thread indices: blockIdx (block index within the grid) and threadIdx (thread index within the block).

The number of thread blocks that the kernel requires to execute all N elements, can be determined by dividing N elements by the size of threads per block and rounding up to the nearest one:

$$\frac{(N+block\ size-1)}{block\ size} \qquad (3.1)$$

3.2.2.2 Thread Hierarchy. In CUDA, every thread has its own unique ID which is related to the index of the thread as shown below.

Thread ID (x, y, z) = (x + y D_X + z D_X Dy) --- for three-dimensional block size (D_x , D_y , D_z).

Thread ID (x, y) = (x + y Dx) --- for two-dimensional block size (D_x , D_y).

Thread ID (x) = threadIdx.x --- for one-dimensional block size (D_x).

By knowing that each thread block has its own memory resources that reside on the processor core shared by all threads in the block, it follows that the number of the threads per block is limited depending on the resources available in the architecture of the GPU.

3.2.2.3 Memory Hierarchy.

There are multiple memory spaces available for a thread during execution as shown below in Figure (3.4.)

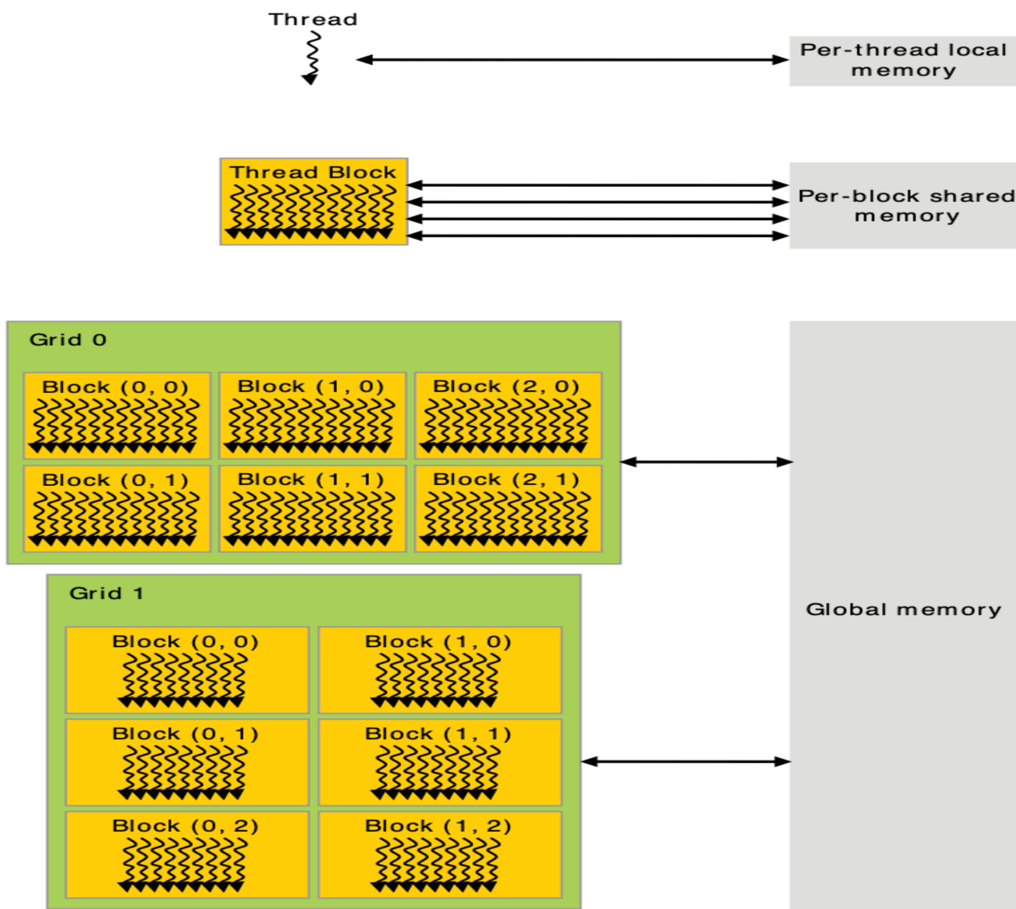


Figure 3.4. Available memories for thread execution period [10]

One should notice that CPU and GPU have separate memories. Thus, data needs to be copied from the host to the device to be processed, and after the computation is done, the results need to be copied back to the CPU. This is one downside of the NVIDIA CUDA (prior to the release of CUDA 6), as such copies are expensive in terms of time and power.

The CUDA environment used in this work is summarized below in Table 3.1.

Name / Brand / Architecture

Manufacturer:	NVIDIA
Model:	Quadro P400
Reference card:	Yes
Target market segment:	Desktop Workstation
Die name:	GP107
Architecture:	Pascal
Fabrication process:	14 nm
Transistors:	3.3 billion
Bus interface:	PCI-E 3.0 x 16
Launch date:	February 2017

Frequency

Base clock:	1070 MHz
Boost clock:	1170 MHz

Memory specifications

Memory size:	2 GB
Memory type:	GDDR5
Memory clock:	1752 MHz
Memory clock (effective):	7008 MHz
Memory interface width:	64-bit
Memory bandwidth:	56.06 GB/s

Cores / Texture

Table 3.1 Continued.

CUDA:	6.1
CUDA cores:	256
ROPs:	16
Texture units:	16
Electric characteristics	
Maximum power draw:	30 W
Video features	
Maximum digital resolution:	7680 x 4320 @60 Hz
Maximum DP resolution:	7680 x 4320 @60 Hz
Maximum HDMI resolution:	4096 x 2160 @ 60 Hz
Performance	
Pixel fill rate:	18.72 Gigapixels/s
Texture fill rate:	18.72 Gigatexels/s
Single precision compute power:	599.04 GFLOPS
Open CL support:	1.2
OpenGL support:	4.5
DirectX support:	12.0
Shader model:	5.0

Table 3.1. Specifications of CUDA Quadro P400

3.3. Hardware Testing Platform

In this thesis, three platforms have been used to run and test the performance of one-dimensional and two-dimensional FDTD problems. In the case of the one-dimensional problem, propagation of electromagnetic waves through a dielectric device with relative permeability of 1 and relative permittivity of 12 and thickness of 3 cm were used.

- Platform one: Haswell-EP w/ NVIDIA Quadro P400 (for full specifications see table (4.1.).
- HP Pavilion Notebook, Intel core i5—6200U CPU @ 2.30GHz (4 CPUs), ~2.4GHz, 8192MB RAM, Intel HD Graphics 520
- OpenCL platform information:
  - Platform 0:
    - Platform name: NVIDIA CUDA
    - Platform version: OpenCL 1.2 CUDA 11.2.162
    - Platform profile: FULL_PROFILE
  - Platform 1:
    - Platform name: Intel CPU Runtime for OpenCL™ Applications
    - Platform version: OpenCL 2.1 LINUX
    - Platform profile: FULL_PROFILE
  - Platform 2:
    - Platform name: Portable Computing Language
    - Platform version: OpenCL 1.2 pocl 1.6, None+Asserts, LIVM 9.0.1, RELOC, SLEEF, DISTRO, POCL_DEBUG
    - Platform profile: FULL_PROFILE

3.4. CUDA Implementation of the One-, and Two-dimensional Problems

3.4.1. Initializing

In general, the FDTD code needs to define specific constants that appear in Maxwell's equations (and specify the units used), define the materials of the device we need to simulate, initialize the fields arrays, and values like computed time step dt, and space step, …etc.

The steps to initialize any FDTD code are listed below. In addition to an example of the flowchart of initializing and defining any two-dimensional FDTD problem in MATLAB. See figure (3.5.).

General steps to initialize and define any FDTD code:

- Define units and constants
- Define the device and initialize its materials
- Calculate space steps $\Delta x, \Delta y, and\ \Delta z$
- Compute grid size (i.e., how many grid cells (Nx, Ny, and Nz) are needed in x, y, and z-directions, respectively).
- Compute the time step (Δt)
- Compute number of time steps needed for simulation
- Compute the source functions
- Initialize the Fourier transform for the frequency response
- Initialize EM fields
- Calculate update coefficients

The following flow chart in figure (3.5) showing an example of the general steps of initializing 2D-FDTD program in MATLAB platform.

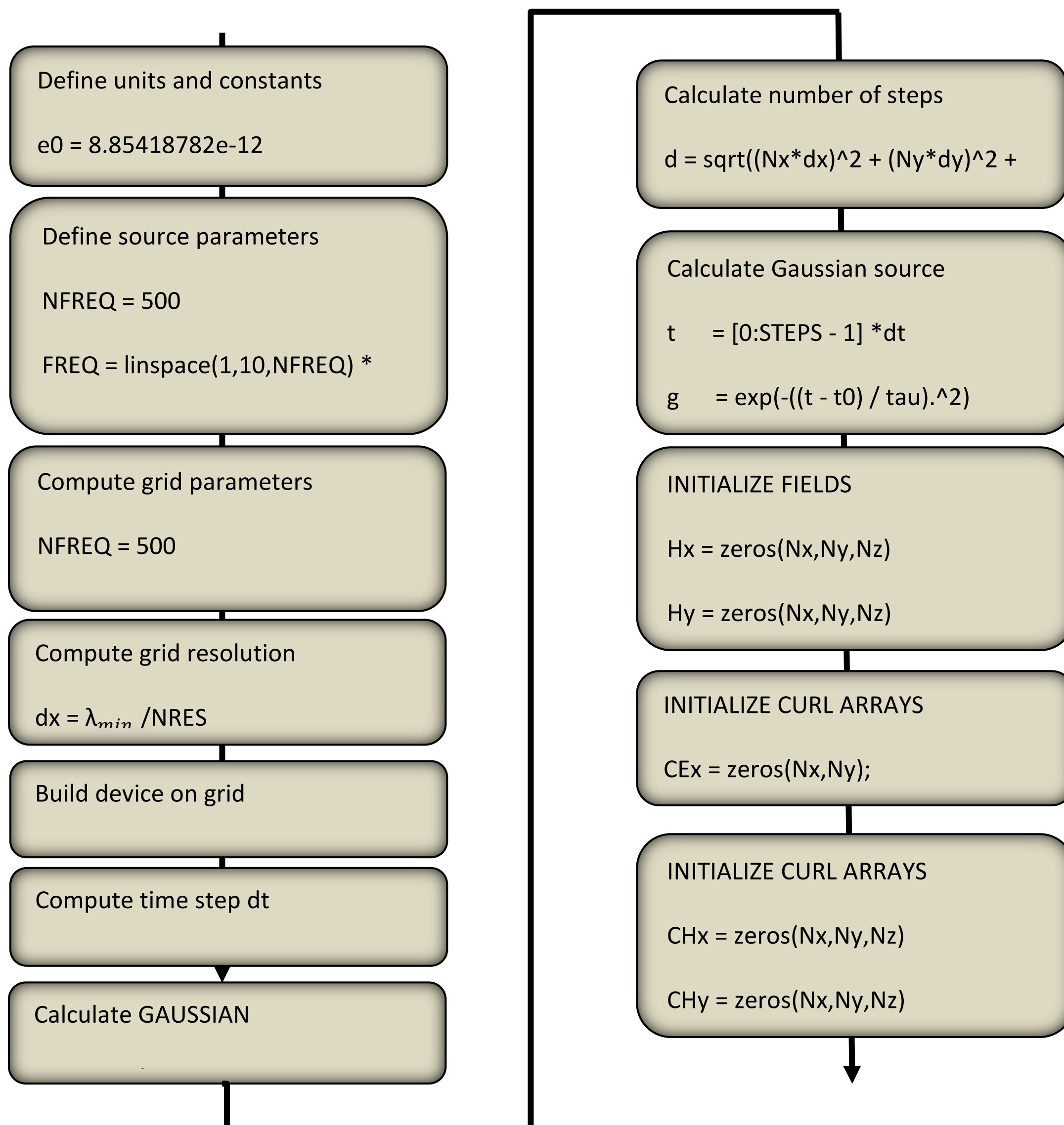


Figure 3.5. An example of initializing and defining Two- Dimensional FDTD problem in MATLAB.

3.4.1 Main Time Loop

After completing the steps above, we can now enter the main updating loop. This loop will be running repeatedly over the time needed for the simulation, usually thousands of steps. The number of steps depends on the device features and materials simulated.

For one-, and two-dimensional FDTD, the updating loop algorithms will be like that shown in figures (3.6.), and (3.7.) respectively.

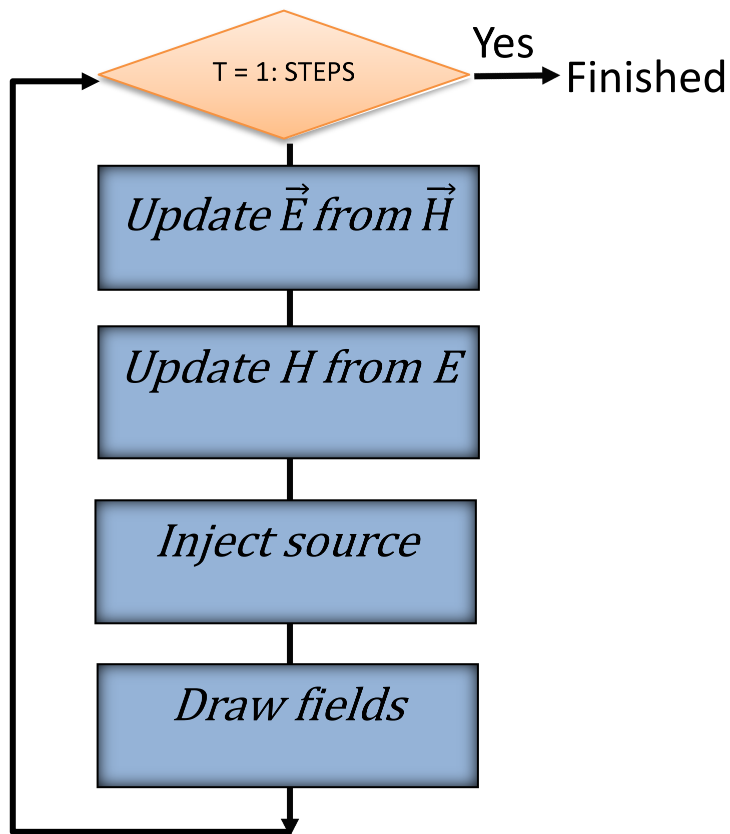


Figure 3.6. One-dimensional FDTD algorithm.

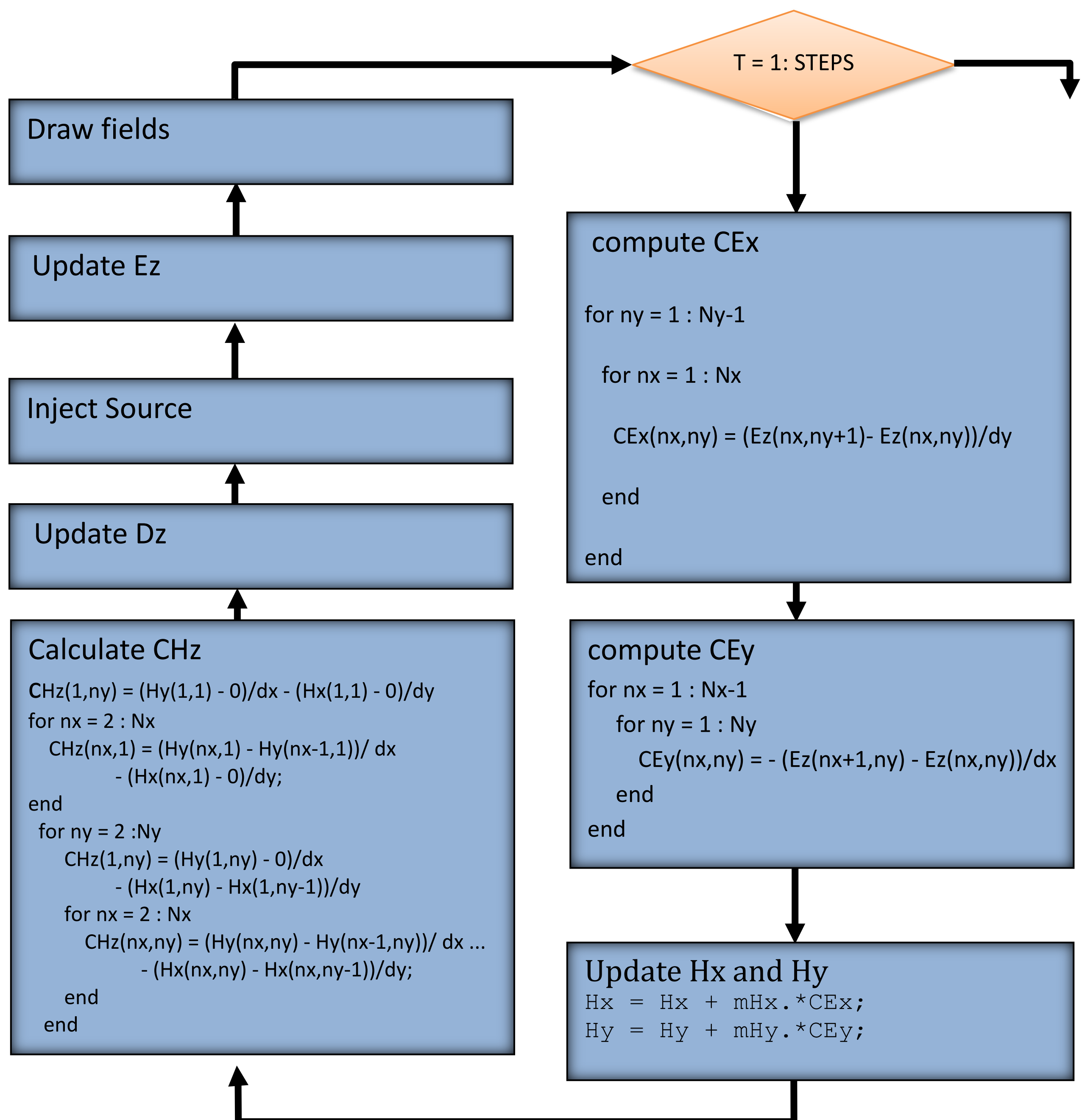


Figure 3.7. Two-dimensional FDTD algorithm (CPU).

3.4.2. Programming GPGPU CUDA

In this thesis we will take the 1D-FDTD c code and parallelize it using CUDA. First, we need to add the header #include <cuda.h> for CUDA libraries and #include <papi.h> for time measurements. Second, we will follow the steps below to program a GPGPU:

Step one: Write a kernel to update the electrical and magnetic fields. The kernels will be run using Nvidia CUDA in parallel. The following code sample shows how to write a CUDA kernel for updating the Hx field.

```
__global__ void cuda_update_magnetic (int n, double dz,double *Ey, double *Hx, double
                                                        *mHx, int src, double Esrc) {
  int tid  = blockIdx.x*blockDim.x + threadIdx.x;
  double e1=0.,e2=0.,e3=0.;
  if (tid<n-1) {
       Hx[tid] = Hx[tid] + mHx[tid] * (Ey[tid + 1] - Ey[tid])/dz;
  }
  if(tid==(n-1))  {
  e3 =e2,e2=e1,e1=Ey[n-1];
  Hx[src-1]  = Hx[src-1] - (mHx[src-1]/dz) * Esrc;
  Hx[tid]   = Hx[tid]  + mHx[tid] * (e3 - Ey[tid])/dz;
 }}
```

The CUDA runtime uses the two parameters from the execution configuration: the number of blocks per grid and the block size to determine how many parallel threads to use for the launch on the GPU. To get the indices of the running threads (which should equal to the number of Yee cells) we use

`int tid  = blockIdx.x*blockDim.x + threadIdx.x.`

Also, the if-condition (tidy<n-1) in the above code is necessary to make sure to set a limit for the "tid" to not to go out of the range of the processing N elements.

Step two: Allocate memory on the GPGPU for Ey, Hx, mEy, and mHx and make a copy of field's arrays from host to device using:

```
cudaMalloc((void**)&dev_Hx , Nz*sizeof(double));

cudaMemcpy(dev_Hx ,Hx ,Nz*sizeof(double),cudaMemcpyHostToDevice);
```

Step three: run the program by calling the kernel

```
cuda_update_magnetic<<<numBlocks,blockSize>>>
                    (Nz,dz,dev_Ey,dev_Hx,dev_mHx,src,Esrc[T]);
```

Step four: Copy the data values back to the host

```
cudaMemcpy(Ey,dev_Ey,Nz*sizeof(double),cudaMemcpyDeviceToHost);
```

3.5. Verification

One important step during parallelizing the FDTD code is verification which was implemented in this thesis. The simple sequential code must be modified significantly before it can run in a parallel platform. The error during parallelizing could arise and the simulation could become unstable and may give exploding values. For that, the same code has been executed twice or more in MATLAB, C, and CUDA and the results have been compared for accuracy (see table (3.2.) below). When the results of MATLAB, CUDA, and C were confirmed to be matching, we set the results of that code as a reference result for any future modifications.

Consider the problem in Figure (3.8.) which was first coded using ***MATLAB***, and then **C** languages see Figure (3.9.) and (3.10) respectively. The Hx field components have been recorded in both approaches (MATLAB and C).

```
e0 = 8.85418782e-12;

u0 = 1.25663706e-6;

c0 = 299792458 ;

er1 = 12.0;        %relative permitivity device

er2 = 1;           %relative permitivity air

ur1 = 1;

ur2 = 1;

L   = 0.03         % thickness of the slab 0.03 m

dz = 0.0021;

Nz = 217

xa = [0:Nz-1]*dz;

ERyy =  ones(1,Nz);

URxx =  ones(1,Nz);

ERyy(103:116) = 12;

URxx(103:116) = 1;

dt  = 3.5739e-12;
```

Figure 3.8. An example of FDTD problem

The **MATLAB** code above transferred to **C** and verified that the results are accurate with zero error to the machine precision. Then the code was modified to run in a CUDA parallel environment and to compare the timing results with those previously obtained. The MATLAB and C code of the main loop of one- dimensional FDTD problem is as shown below in Figure (3.9) and (3.10):

```
for T = 1 :  STEPS
   % Update H from E
   for nz = 1 : (Nz-1)
       Hx(nz) = Hx(nz) + mHx(nz)*(Ey(nz+1) - Ey(nz))/dz;
   end
   e3=e2;e2 = e1;e1 = Ey(Nz);
   % Handle H source
   Hx(src-1)= Hx(src-1)-(mHx(src-1)/dz) * Esrc(T);
   Hx(Nz) = Hx(Nz) + mHx(Nz) * (e3 - Ey(Nz))/dz;     %(Perfect
Absorbing)
```

Figure 3.9. An example of FDTD code in MATLAB

```
/* start Main loop */
for (T = 0; T < STEPS; T++) {
/* Update Magnetic field H from E*/
for (nz = 0; nz < Nz - 1; nz++)
Hx[nz] = Hx[nz] + mHx[nz] * (Ey[nz + 1] - Ey[nz])/dz;
e3 =e2,e2=e1,e1=Ey[Nz-1];
Hx[src-1]  = Hx[src-1] - (mHx[src-1]/dz) * Esrc[T];
Hx[Nz-1]   = Hx[Nz-1]  + mHx[Nz-1] * (e3 - Ey[Nz-1])/dz;
h3 = h2, h2 = h1, h1 = Hx[0];
/* Update Electric field E from H */
Ey[0] = Ey[0] + mEy[0] * (Hx[0] - h3)/dz;
for (nz = 1; nz<Nz; nz++)
Ey[nz] = Ey[nz] + mEy[nz] * (Hx[nz] - Hx[nz - 1])/dz ;
/* Handle E source */
Ey[src] = Ey[src] - ((mEy[src]/dz) * Hsrc[T]);
} /* end of Main Loop */
```

Figure 3.10. An example of the 1D-FDTD sequential code in C

After running the code in Figure (3.9.) and (3.10.), the Hx field values recorded and used to compute the error. The Table (3.2.) showing the values of Hx field at time ($1.429e^{-9}$ sec) when the code executed on different platforms ***MATLAB***, ***C***, and ***CUDA***.

	MATLAB	CUDA	C	Error
$H_x(0)$	-0.44448	-0.44448	-0.44448	0.00000
$H_x(1)$	-0.46842	-0.46842	-0.46842	0.00000
$H_x(2)$	-0.49278	-0.49279	-0.49279	0.00000
$H_x(3)$	-0.51751	-0.51751	-0.51751	0.00000
$H_x(4)$	-0.54252	-0.54252	-0.54252	0.00000
$H_x(5)$	-0.56774	-0.56774	-0.56774	0.00000
$H_x(6)$	-0.59310	-0.59310	-0.59310	0.00000
$H_x(7)$	-0.61850	-0.61850	-0.61850	0.00000
$H_x(8)$	-0.64386	-0.64386	-0.64386	0.00000
$H_x(9)$	-0.66908	-0.66908	-0.66908	0.00000
$H_x(10)$	-0.69407	-0.69408	-0.69407	0.00000
$H_x(11)$	-0.71874	-0.71874	-0.71874	0.00000
$H_x(12)$	-0.74298	-0.74298	-0.74298	0.00000
$H_x(13)$	-0.76668	-0.76668	-0.76668	0.00000
$H_x(14)$	-0.78976	-0.78976	-0.78976	0.00000
$H_x(15)$	-0.81211	-0.81211	-0.81211	0.00000
$H_x(16)$	-0.83362	-0.83362	-0.83362	0.00000
$H_x(17)$	-0.85421	-0.85421	-0.85421	0.00000
$H_x(18)$	-0.87376	-0.87377	-0.87376	0.00000
$H_x(19)$	-0.89220	-0.89220	-0.89220	0.00000
$H_x(20)$	-0.90944	-0.90944	-0.90944	0.00000
$H_x(21)$	-0.92537	-0.92538	-0.92537	0.00000

Table 3.2 Continued.

-0.93994	-0.93994	-0.93994	0.00000
-0.95307	-0.95307	-0.95307	0.00000
-0.96468	-0.96468	-0.96468	0.00000
-0.97472	-0.97472	-0.97472	0.00000
-0.98314	-0.98314	-0.98314	0.00000
-0.98990	-0.98990	-0.98990	0.00000
-0.99495	-0.99495	-0.99495	0.00000
-0.99828	-0.99828	-0.99828	0.00000
-0.99986	-0.99986	-0.99986	0.00000
-0.99969	-0.99969	-0.99969	0.00000
-0.99776	-0.99776	-0.99776	0.00000
-0.99410	-0.99410	-0.99410	0.00000
-0.98871	-0.98871	-0.98871	0.00000
-0.98163	-0.98163	-0.98163	0.00000
-0.97289	-0.97289	-0.97289	0.00000
-0.96254	-0.96254	-0.96254	0.00000
-0.95063	-0.95063	-0.95063	0.00000
-0.93722	-0.93722	-0.93722	0.00000
-0.92238	-0.92238	-0.92238	0.00000
-0.90618	-0.90618	-0.90618	0.00000
-0.88871	-0.88871	-0.88871	0.00000
-0.87005	-0.87005	-0.87005	0.00000

Table 3.2 Continued.

-0.85029	-0.85029	-0.85029	0.00000
-0.82952	-0.82952	-0.82952	0.00000
-0.80784	-0.80784	-0.80784	0.00000
-0.78535	-0.78535	-0.78535	0.00000
-0.76215	-0.76215	-0.76215	0.00000
-0.73834	-0.73834	-0.73834	0.00000
-0.71402	-0.71402	-0.71402	0.00000
-0.68929	-0.68929	-0.68929	0.00000
-0.66425	-0.66425	-0.66425	0.00000
-0.63900	-0.63900	-0.63900	0.00000
-0.61363	-0.61363	-0.61363	0.00000
-0.58824	-0.58824	-0.58824	0.00000
-0.56292	-0.56292	-0.56292	0.00000
-0.53774	-0.53774	-0.53774	0.00000
-0.51278	-0.51278	-0.51278	0.00000
-0.48813	-0.48813	-0.48813	0.00000
-0.46386	-0.46386	-0.46386	0.00000
-0.44002	-0.44002	-0.44002	0.00000
-0.41667	-0.41667	-0.41667	0.00000
-0.39388	-0.39388	-0.39388	0.00000
-0.37168	-0.37168	-0.37168	0.00000
-0.35012	-0.35012	-0.35012	0.00000

Table 3.2 Continued.

	-0.32923	-0.32924	-0.32924	0.00000
	-0.30906	-0.30906	-0.30906	0.00000
	-0.28961	-0.28961	-0.28961	0.00000
	-0.27092	-0.27092	-0.27092	0.00000
	-0.25300	-0.25300	-0.25300	0.00000
$H_x(211)$	-0.23585	-0.23585	-0.23585	0.00000
$H_x(212)$	-0.21949	-0.21949	-0.21949	0.00000
$H_x(213)$	-0.20391	-0.20391	-0.20391	0.00000
$H_x(214)$	-0.18911	-0.18911	-0.18911	0.00000
$H_x(215)$	-0.17510	-0.17510	-0.17510	0.00000
$H_x(216)$	-0.16184	-0.16185	-0.16185	0.00000

Table 3.2. An example of Hx values when the 1D-FDTD code executed in MATLAB, C, and CUDA C

3.6. Speedup, Parallel Efficiency, and Strong and Weak Scaling

Speedup is the improvement in time to run [16], and is typically defined as:

$$\boldsymbol{S} = \frac{\boldsymbol{t_{old}}}{\boldsymbol{t_{new}}} \qquad (3.2)$$

The parallel efficiency is the speedup divided by the number of processors[16]:

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p} \qquad (3.3)$$

Here *p* is the number of processors (threads) and *Ts* is the serial execution time, *Tp* is the parallel execution time.

Two types of scaling will be used to analyze the performance of parallelization strategy: Strong and Weak scaling.

Strong scaling describes the speedup we can get when we add more processors with a fixed problem size. Amdahl's law for strong scaling is shown below in equation (3.4):

$$S = \frac{\mathbf{1}}{(\mathbf{1}-\boldsymbol{P})+ \frac{\boldsymbol{P}}{\boldsymbol{N}}} \qquad (3.4)$$

S is the maximum speed up, *N* is the number of processors, *P* is the parallel fraction of the code (if the total time needed for running sequential program is t, then P is the fraction of the time t that can be parallelized).

Weak scaling describes the speedup we can get when expanding the problem size with a fixed number of processors as shown in Equation 3.5 below:

$$S = N + (1 - P)\,(1 - N) \qquad (3.5)$$

3.7. Timing

The time of the computation including launching kernels, copying to and from the device, and the total time of running has been recorded through the CPU- PAPI and through the GPU-CUDA events. By measuring the timing two diverse ways, higher accuracy can be assured.

The code of the CPU PAPI timer is as shown below:

```
PAPI_library_init(PAPI_VER_CURRENT);

Start_time = PAPI_get_real_usec();

…..

Stop_time = PAPI_get_real_usec();
```

While the code of the GPU CUDA events timer will be as below:

```
cudaEvent_t start, stop;

float time;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord( start, 0 );

kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y, NUM_REPS);

cudaEventRecord( stop, 0 );

cudaEventSynchronize( stop );

cudaEventElapsedTime( &time, start, stop );

cudaEventDestroy( start );

cudaEventDestroy( stop );
```

3.8. Performance of the FDTD Parallel Code

For measuring the performance of parallelized FDTD code, we used two methods. The first method is when the serial FDTD code runs on GPU first with one thread. Then it runs again with many threads, which is what we call **GPU-GPU** performance. The second method is when the code is first executed on the CPU then parallelized and executed on GPU with many threads. We call this method **CPU-GPU** performance.

3.8.1 GPU-GPU Performance

The C code was implemented in the CUDA language. The following sections will explore using one or more threads.

3.8.1.1. Grid Size 217 cells. In section 3.5, we ended with the C code that is verified and prepared for the parallel process. But the time needed to run the code in CPU would be differ than if it were executed in GPUs (GPU memory allocation time, copy data to and back from GPU, GPU free memory,

etc.) and we would need the elapsed time of sequential code to compute the speedup and parallel efficiency. For that, the same C code in Figure (3.10.) would need to be modified first to be run in CUDA sequentially using one thread, the result would need to be verified and recorded for parallel comparison. The following Figure (3.11.) illustrates how to call the kernel with the one thread:

```
for (T = 0; T < 400; T++) {

//%%%%%%%%%*CUDA call kernel to update Hx field*%%%%%%%%%//
cuda_update_magnetic<<<1,1>>>(Nz,dz,dev_Ey,dev_Hx,dev_mHx,src,Esrc[T]);

cuda_update_electric<<<1,1>>>(Nz,dz,dev_Ey,dev_Hx,dev_mEy,src,Hsrc[T]);

} /* end of Main Loop */
```

Figure 3.11. CUDA function to call the kernel of the same code of figure (3.10.)

The CUDA Toolkit comes with a tool called the GPU profiler. This tool can be used to find the elapsed time that the kernel takes during running the program.

Entering the command nvprof ./cuda file name on the command line gives the following:

```
PAPI_updateTime      : 136265 micro sec
cudaEvent_updateTime : 136234.562500 micro sec
Copy_HostDeviceTime  : 65 micro sec
Copy_DeviceHostTime  : 35 micro sec
Total time           : 614 msec
==18176== Profiling application: ./1d_oneThread_cuda
==18176== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.07%  67.967ms       400  169.92us  141.12us  249.72us  cuda_update_magnetic(int, double, double*, double*, double*
 int, double)
                   49.93%  67.783ms       400  169.46us  140.41us  249.44us  cuda_update_electric(int, double, double*, double*, double*
 int, double)
                    0.00%  4.3840us         4  1.0960us     864ns  1.7280us  [CUDA memcpy HtoD]
                    0.00%  2.2720us         2  1.1360us  1.0560us  1.2160us  [CUDA memcpy DtoH]
```

- Notice that both PAPI and CUDA-Event timers give the same field updated time (136 ms).

Next, we parallelize the code by adding more threads for computation. The kernel code needs to be modified by collapsing the for loop and getting the right threads indices. Additionally, the execution configuration needs to be modified as follows:

```
__global__
void cuda_update_magnetic (int n, double dz,double *Ey, double *Hx, double *mHx, int src, double Esrc) {
   int tid  = blockIdx.x*blockDim.x + threadIdx.x;
   double e1=0.,e2=0.,e3=0.;

   if (tid<n-1) {
         Hx[tid] = Hx[tid] + mHx[tid] * (Ey[tid + 1] - Ey[tid])/dz;
   }

    /* Handle H source */
   if(tid==(n-1))
   {
   e3 =e2,e2=e1,e1=Ey[n-1];
   Hx[src-1]  = Hx[src-1] - (mHx[src-1]/dz) * Esrc;
   Hx[tid]   = Hx[tid]  + mHx[tid] * (e3 - Ey[tid])/dz;
  }
}
```

```
for (T = 0; T < STEPS; T++) {
//%%%%%%%%%*CUDA call kernel to update Hx field*%%%%%%%%%//
cuda_update_magnetic<<<numBlocks,blockSize>>>(Nz,dz,dev_Ey,dev_Hx,dev_mHx,src,Esrc[T]);
cuda_update_electric<<<numBlocks,blockSize>>>(Nz,dz,dev_Ey,dev_Hx,dev_mEy,src,Hsrc[T]);
} /* end of Main Loop */
```

Where number of threads per block = 1, and number of blocks using equation (3.1)=

217+1-1/1 = 217

Thus, we added 217 threads in this case and the field updated time reduced to about (136 -3.8 ms), but the total elapsed time increased from 614 msec to 667 msec, as shown below:

```
-7.7705e-27
-2.99295e-27PAPI_updateTime    : 3841 micro sec
cudaEvent_updateTime : 3814.016113 micro sec
Copy_HostDeviceTime  : 57 micro sec
Copy_DeviceHostTime  : 33 micro sec
Total time           : 667 msec
==16240== Profiling application: ./1d_fdtd_cuda
==16240== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   53.54%  1.6835ms       400  4.2080us  3.5840us  6.3680us  cuda_update_magnetic(int, double, double*, double*, double*,
 int, double)
                   46.25%  1.4541ms       400  3.6350us  3.5510us  5.7280us  cuda_update_electric(int, double, double*, double*, double*,
 int, double)
                    0.14%  4.2880us         4  1.0720us     864ns  1.6320us  [CUDA memcpy HtoD]
                    0.07%  2.2400us         2  1.1200us  1.0560us  1.1840us  [CUDA memcpy DtoH]
```

Finally, using the full range of threads per block, which is 256 threads for Quadro P400, the updated time for one block with 256 threads was 3.8 ms and the total time was 629 msec.

```
PAPI_updateTime      : 3847 micro sec
cudaEvent_updateTime : 3820.192139 micro sec
Copy_HostDeviceTime  : 59 micro sec
Copy_DeviceHostTime  : 31 micro sec
Total time           : 629 msec
==18540== Profiling application: ./1d_fdtd_cuda
==18540== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   53.61%  1.6855ms       400  4.2130us  3.5840us  6.3040us  cuda_update_magnetic(int, double, double*, double*, double*,
 int, double)
                   46.17%  1.4513ms       400  3.6280us  3.5200us  5.6320us  cuda_update_electric(int, double, double*, double*, double*,
```

The following table (3.3.) shows the performance of running one-dimensional FDTD problem with different values of execution configurations. Equations (3.2), and (3.3) are used to compute speedups and parallel efficiencies.

CUDA QUADRO P400 EP machine				
217 Yee cells, 400 Steps				
Version	Update Time(ms)	Total Time(ms)	Speedup	Efficiency
1 Block , 1 Thread	136.000	614		
217 Block , 1 Thread	3.176	667	43	20%
1 Block , 217 Thread	3.140	660	43	20%
1 Block , 256 Thread	3.148	629	43	20%

Table 3.3. Performance of CUDA with 217 cells and 400 steps

Grid size 217			
Iterations	**GPU** update Time-one thread (ms)	Update Time- 256 threads (ms)	speedup
400	136	3.1	43
8193	2331	52.5	44.4
10000	2838	64.0	44.3

Table 3.4. Performance of CUDA of 217 cells with 400,8193, and 10000 steps

The simulation of the totally parallel (number of threads = grid size) code with grid size of 217 has been checked with ($1.42e^{-9}$, $2.928e^{-8}$, and $3.573e^{-8}$ sec) and verified with average speedup of 44.

3.8.1.2. Grid Size 617 cells. In this case, The CUDA code tested with a bigger problem size. The number of cells needed to represent the FDTD space Nz is 617, and the number of blocks = Nz +block size -1 /block size = 617 blocks will be running on 256 GPU- cores. As shown below, in Table (3.5.), when expanding the problem space (217 cells – 617 cells) the GPU throughput increases as shown in Table (3.5).

Grid size 617				
Iterations	Update Time-one thread (ms)	Update Time- 256 threads (ms)	Speedup	Parallel Efficiency
400	483.6	3.3	146	57%
10000	8125	68.9	118	46%
22049	1767.4	150	118	46%

Table 3.5. Performance of CUDA of 617 cells with 400,10000, and 22049 steps

With 617 grid sizes, the parallel CUDA shows better speedup and better parallel efficiency with an average speed up and average proficiency of 127 (44 with 217 size), and 50% (20% with size 217) respectfully.

3.8.1.3. Grid Size 4017 cells. We can expand our FDTD problem size to 4017 grid cells with a fixed compute capability of 256 cores of NVIDEA GPU. The following table (3.6.) shows the total times needed to update the electromagnetic fields with 400, 50000, and 139829 steps.

Grid size 4017				
Iterations	Time-one thread (ms)	Time- 256 threads (ms)	Speedup	Parallel Efficiency
400	3572	8.6	415	162%
50000	273803	670.8	409	160%
139829	738341	1794.6	411	161%

Table 3.6. Performance of CUDA of 4017 cells with 400,50000, and 139829 steps

As it can be seen in the following table (3.7.), speedup of up to 500 can be achieved in case of 80000 grid cells and 10000 iterations when launching more processors (256 instead of one) in the problem.

10000 iterations			
Grid Size	GPU-1 thread	GPU-256 thread	speedup
217	3225	454	7
617	8462	462	18
4017	67476	535	126
10000	213078	857	249
20000	458816	1282	358
40000	980718	2181	450
80000	1975507	3912	505

Table 3.7. GPU-GPU Performance with 10000 iterations with different grid sizes

3.8.2. CPU-GPU Performance

The second method to measure the performance of parallelizing the FDTD sequential code is **CPU-GPU** method. The C code of the FDTD is first executed on the CPU then parallelized and executed on GPU with many threads. The following table (3.8.) and figure (3.12.) show the total running times in (msec) of the one-dimensional FDTD code in three platforms (C, MATLAB, and CUDA) with 10000 iterations and different grid sizes (217, 617, 4017, 10000, 20000, 40000, and 80000).

10000 iterations						
	Grid Size	Time(msec) CPU-C	Time(msec) MATLAB	Time(msec) CUDA	Speedup/CPU-CUDA	Parallel Efficiency
1	217	37	48	454	0.08	0.03%
2	617	78	70	462	0.17	0.07%
3	4017	1287	373	535	2.40	0.94%
4	10000	4323	902	857	5.04	1.97%
5	20000	4873	1809	1282	3.80	1.48%
6	40000	5920	3641	2181	2.71	1.06%
7	80000	7669	7257	3912	1.96	0.77%

Table 3.8. CPU-GPU Performance with 10000 iterations in different grid sizes

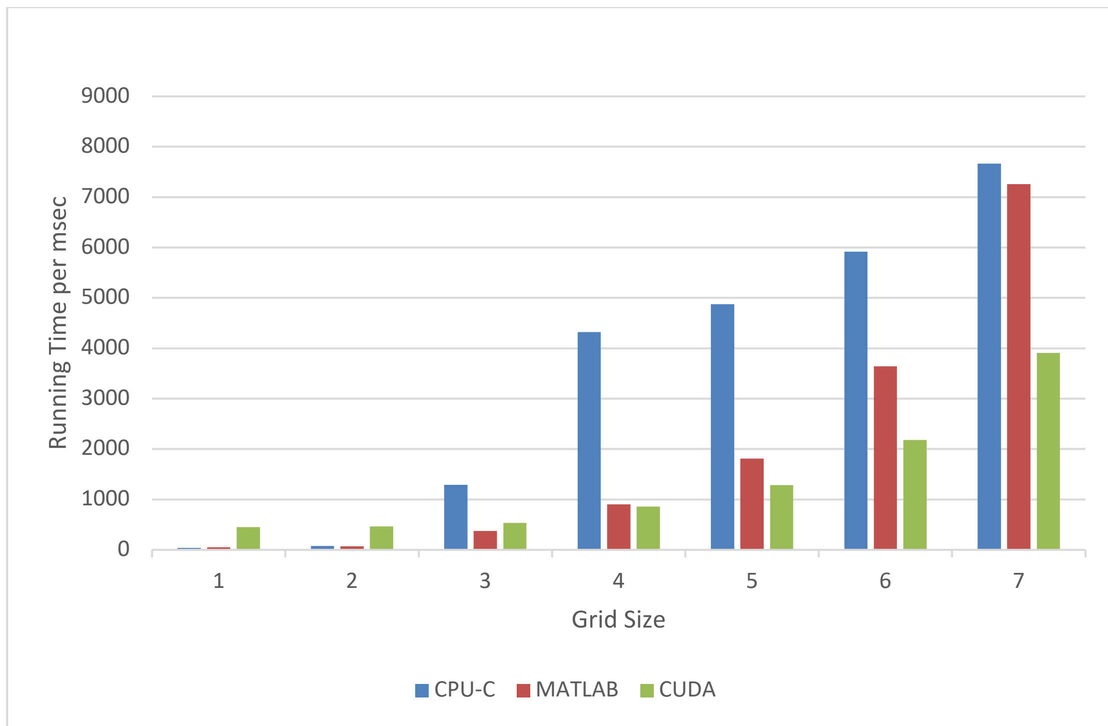


Figure 3.12. C, MATLAB, and CUDA running time with 10000 iterations and different grid size

In the case of one-dimensional FDTD problems, the CPU-GPU performance shows speedup of running time up to 5, as shown in above table (3.8.), and Figure (3.12.).

The speedup of the parallel code increased when we increased the FDTD size with weak scaling as shown in the following Figure (3.13.) when the number of the CUDA threads used to accelerate the code (256 threads or cores in Quadro-P400) was fixed while increasing the size of the problem (217, 417, 617, 10000, 20000, and 40000).

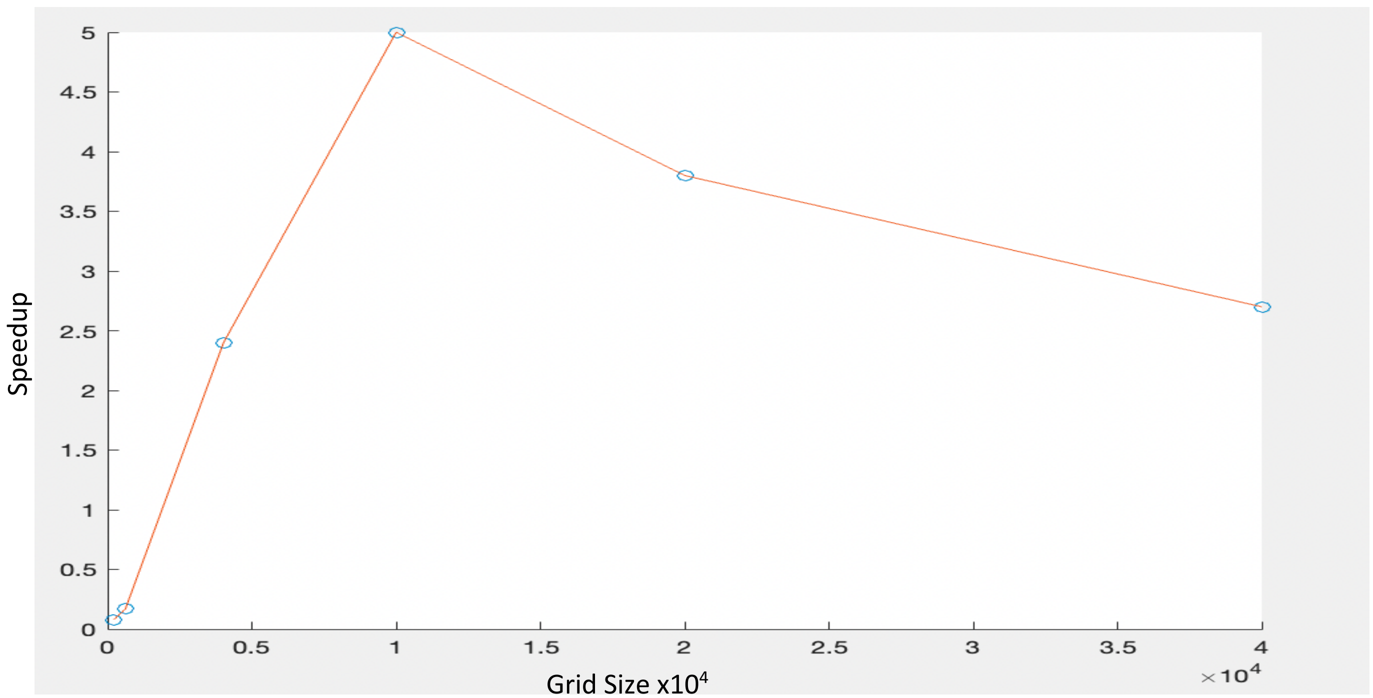


Figure 3.13. CUDA performance when increase grid size with fixed threads number (256 threads)

3.9. Conclusion of CPU-GPU method

Parallelizing the FDTD code using GPU-CUDA is efficient under the condition that the running time of the code using a single CPU is more than the time that CUDA needs to allocate memory on GPU. In other words, if the problem size is not big enough, then the parallelization will be inefficient. The following Figure (3.14.) shows a sample of the times that CUDA needed to run the FDTD code.

```
Grid size=217
cuda mem allocate      : 248195 micro sec
Copy_HostDeviceTime    : 63 micro sec
PAPI_updateTime        : 137477 micro sec
cudaEvent_updateTime   : 137444.796875 micro sec
Copy_DeviceHostTime    : 34 micro sec
cudaFree_Time          : 154 micro sec
Total time             : 385 msec
```

Figure 3.14. An example of the times that CUDA needed to perform FDTD code

As shown in the above Figure (3.14.), CUDA spent about 248.195 msec of 385 msec (64%) in allocating the GPU's memory, while spending 137.477 msec of 385 (35%) on the computation process. See the following figure (3.15.).

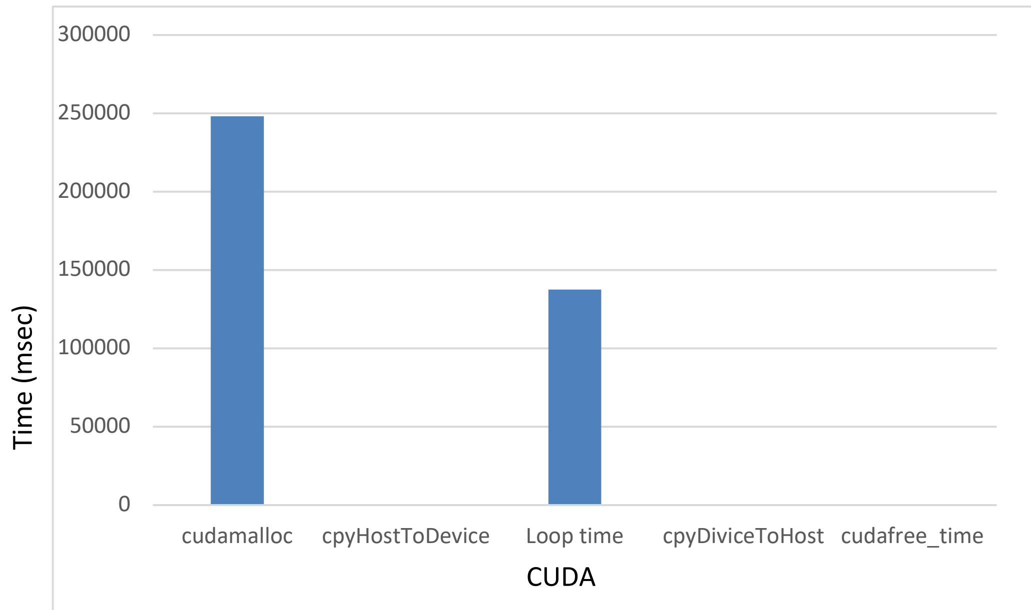


Figure 3.15. speedup of the CUDA with different grid sizes

When the time for allocating the GPU memory is more than the time for computation as in case of 217 and 417 grid size, as showed in Table (3.8.), then the performance of CUDA code was inefficient. We were able to speed up the running time and got better parallel efficiency when the problem size expanded to 4017 cells or higher.

.

CHAPTER 4

4. HARDWARE ACCELERATION OF THE FDTD METHOD – OPENCL

4.1. Introduction

The OpenCL (Open Computing Language) is an open-source software package that is used for accelerating parallel computation. In contrast to CUDA, which only supports NVIDIA and enabled GPUs, OpenCL supports a wide range of parallel processing platforms such as CPUs, GPUs, DSPs, FPGAs, and others.

In addition to graphics pipeline applications, OpenCL is a powerful tool to use on other parallel computational algorithms. In particular, the programmer can use OpenCL to write general purpose portable and efficient programs that can be executed on different devices and architectures.

4.2. GPU Programming Using OpenCL

The architecture of OpenCL can be divided into four models: platform model, memory model, execution model, and programming model.

4.2.1. Platform Model

The way that OpenCL defines the hardware and connection between the host device and the GPU can be described under the platform model. Compute devices can be defined as any devices that have one or more processors that execute parallel programs. These include CPUs, GPUs, DSPs, … etc. When the programmer writes an OpenCL application, the code is divided into two parts: one part to be executed (sequential part) on the CPU or **the host** and the second part to be run on the OpenCL devices, **the kernel**. The host is where the sequential part of the program is executed, and the kernel is where the parallel computation is performed. The OpenCL API connects the host with the kernel device. The kernel device is designed to have a set of **compute units** (CUs), each with one or more **processing elements** (PEs), memories, and schedulers.

Figure (4.1.)“below shows a typical OpenCL platform model with one host plus one or more compute devices each with one or more compute units composed of one or more processing elements” [17].

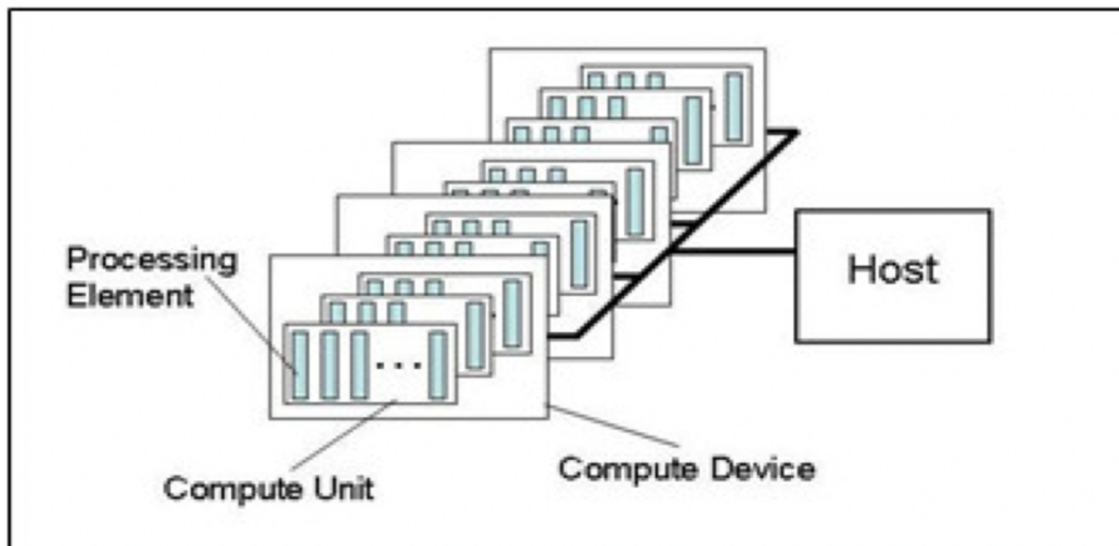


Figure 4.1. Platform Model[17]

4.2.2. Memory Model

There are several levels of memory that may be available during an OpenCL application. OpenCL has four memory domains as shown in Figure (4.2.), private, local, global, and constant. Host global memory and OpenCL devices memory can be divided into **global memory** (accessed for reading/writing by all independent but not synchronized work-items), **constant memory (**constant memory within the global memory for the host to allocate and initialize memory objects), **local memory** (shared and synchronized -memory fences or barriers-by work-items within workgroups), and **private memory** (per work-item). Table (4.1.) illustrates the difference between each type of memory.

Memory	Description
Private	*Specific for work-item*
Local	*Specific for workgroup (shared per work-items within the workgroup) , very low-latency, at least one order of magnitude higher effective bandwidth than global*
Global	*accessible to the host(r/w and map) and to all work-items*
Constant	*accessible to the host (allocates, and initializes memory objects) and to the OpenCL devices for reading only*

Table 4.1. Description of OpenCL device's memories [20]

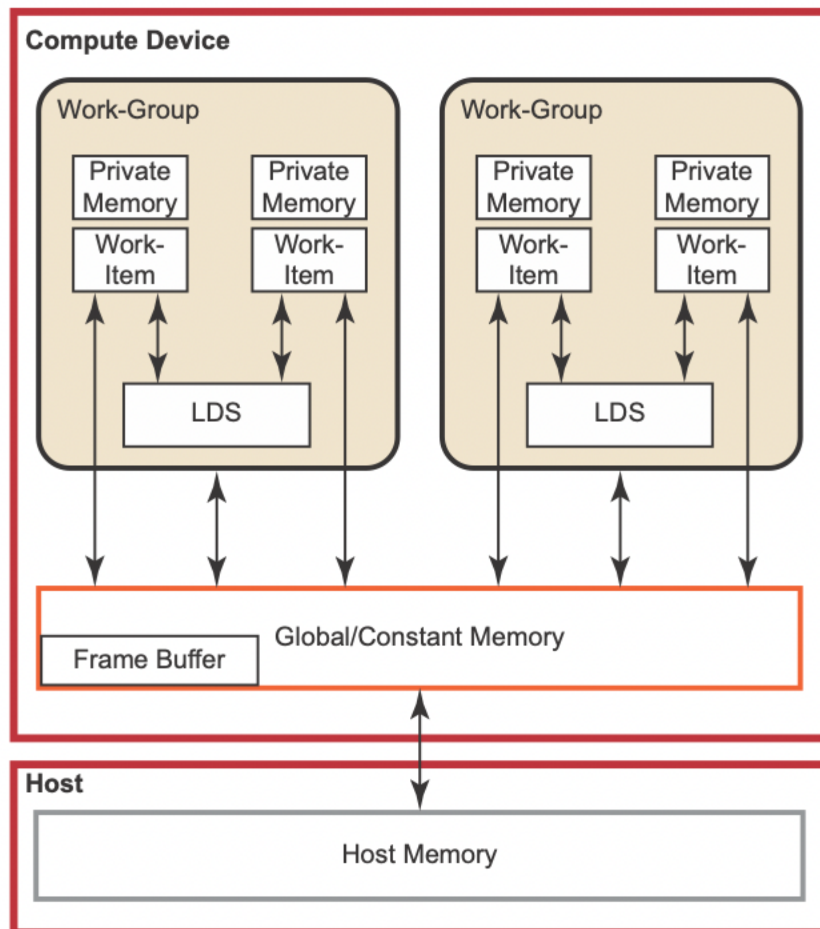


Figure 4.2. Memory Configuration [20]

4.2.2. Execution Model

In this mode, three key items need to be defined: *kernel*, *host program,* and *applications queue*

Kernel execution instances.

Kernel is the parallel part of the program which will be executed concurrently in OpenCL devices. A kernel function, specified by using the kernel keyword, is a data-parallel or task-parallel function executed for each **work-item** (the equivalent of threads in CUDA). A group of 64 work-items, on most AMD GPUs [20], is called a wavefront. **Workgroups** (the equivalent of thread blocks in CUDA) are composed of one or more wavefronts. Workgroups are then assigned to one or more computing units (CUs).

The host program uses the OpenCL API to launch and define the execution configuration (the context) of the devices through a ***command-queue***.

The index space used to assign work items is called **NDRange** (N-Dimensional index Range). NDRange can be one-, two-, or three- dimensional. The programmer can specify the total number of work-items to run at the same time, as well as the workgroups required on NDRange to perform the computation. Figure (4.3.) and (4.4.) provide examples of an NDRange.

In OpenCL, the total range of work-items needed for fully parallel computation (for FDTD there is one work-item per cell) is called a **global** domain and can be called using the function get_global_id(n=0,1,2) which returns the current position (the address) of the work-item. There is one unique global id for each work-items domain. Other helpful functions are get_work_dim, which returns the dimensions of the work (for one dimension work = 0), and get_global_size, which returns the size of the work.

The size of the work, returned by get_global_size, is divided into several local groups (that can be found from get_num_group) with size (get_local_size) so the global work size is equal to the number of local groups multiplied by the size of the local group. Every local group has a unique id that can be accessed using the function get_group_id() , and every work-item has a unique id inside the local

group which can be accessed by using get_local_id(). So, every work-item has two ids, global and local, returned by get_global_id and get_local_id.

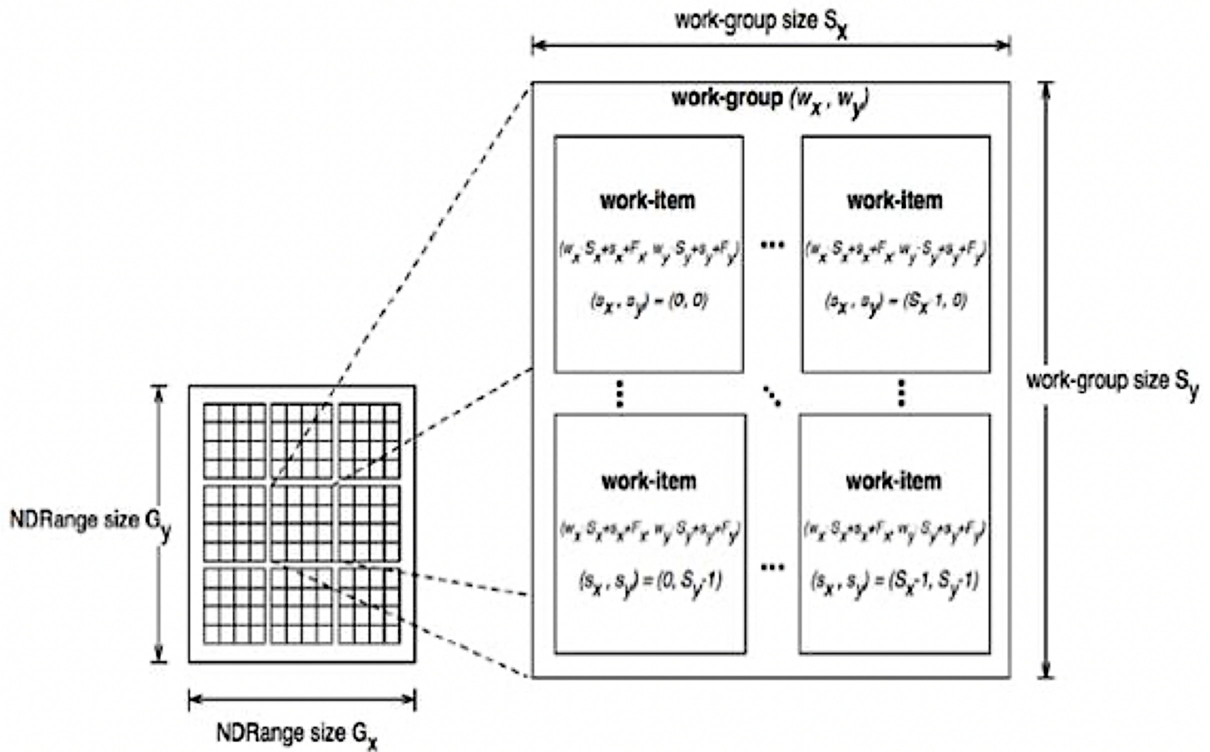


Figure 4.3. An example of two dimensional NDRange [17]

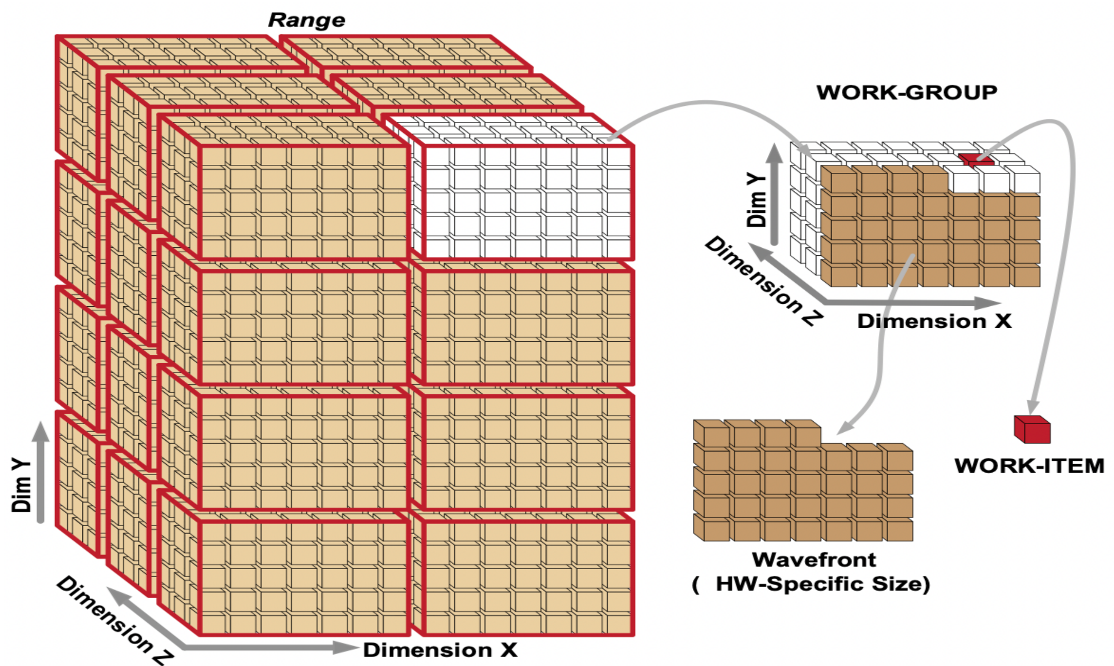


Figure 4.4. An example of three-dimensional NDRange (AMD)[18]

4.2.3. Programming Model

The main steps to run an OpenCL computation are initialization, allocate resources, creating and compiling programs and kernels, executions, and cleanup.

4.2.3.1 Initialization. The OpenCL application needs to define and set up the platform and devices available, create context, and create command queues. The order of operation is:

1- Get the platform and the devices

2- Create a context

3- Create command queue(s)

1: The list of platforms and devices available can be obtained using the functions

```
cl_uint num_platforms;
cl_uint num_devices;
cl_device_id devices;
err = clGetPlatformIDs(MAX_PLATFORMS,platform,&num_platforms);
err = clGetDeviceIDs(platform[which_platform], CL_DEVICE_TYPE_ALL, MAX_DEVICES, devices,
&num_devices);
                                              CL_DEVICE_TYPE_CPU
                                              CL_DEVICE_TYPE_ACCELERATOR
                                              CL_DEVICE_TYPE_GPU
```

Where:

MAX_PLATFORMS = size of the platform array

Platform = pointer to platform array

&num_platforms = number of platforms available

platform[which_platform] = cl_platform_id platform,

CL_DEVICE_TYPE_ALL = cl_device_type device_type,

MAX_DEVICES = cl_uint num_entries,

devices = cl_device_id* devices,

&num_devices = cl_uint* num_devices);

There are many helpful functions that can be used to get more information about the device such as local memory size, global memory size, maximum compute units, max clock frequency, and many others [17]. Knowing compute device information helps one choose the best device for the computation application needed. Calling the device information function is straightforward, and the below function shows how to get the size of the local memory.

```
clGetDeviceInfo(device_id, CL_DEVICE_LOCAL_MEM_SIZE, sizeof(device_local_mem_size),
&device_local_mem_size, &returned_size).
```

2: To create the context call the function, clCreateContext, devices within the same context may share memory objects.

```
Cl_context context;

context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
```

Where:

NULL, means selected platform is implementation-defined

1, Number of devices specified before

&device_id is a pointer to a list of unique devices returned by clGetDevice

NULL, no callback function is registered.

NULL, user-data

&error will return an appropriate error code

3: Create command queue(s) call the function (we need at least one queue for every device)

```
cl_command_queue commands

commands = clCreateCommandQueue(context, device_id, 0, &err);
```

4.2.3.2 Allocating memory. To create memory objects, we used the function in figure (4.5.)

```
cl_mem dev_Hx;

dev_Hx = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(double)* Nz, NULL,NULL);

err = clEnqueueWriteBuffer(commands, dev_Hx, CL_TRUE, 0, sizeof(double) * size, Hx, 0,  NULL,

NULL);

clFinish(command);
```

Figure 4.5. Create buffer function

The buffer memory is associated with a context. The buffer is for reading and/or writing and has a size of the size in bytes of a double number multiplied by the number of elements of the Hx array.

To access memory object (read, write, map, and copy) data, we use the following functions:

```
clEnqueueReadBuffer(queue, object, blocking, offset, size, *ptr, …)

clEnqueueWriteBuffer(queue, object, blocking, offset, size, *ptr, …)

clEnqueueMapBuffer(queue, object, blocking, flags, offset, size, …)

clEnqueueCopyBuffer(queue, srcobj, dstobj, src_offset, dst_offset, …)
```

operate synchronously (blocking = CL_TRUE) or asynchronously

4.2.3.3. Program and Kernel Objects: After allocating memory objects, the next step will be creating the program object which consists of all the executable kernels. This requires the use of two functions (clCreateProgramWithSource and clBuildProgram) to create and build the program and (clCreateKernel) to create the kernel as shown in Figure (4.6.).

```
program_updateHx = clCreateProgramWithSource(context, 1,(const char **) &KernelSource_up
                                  dateHx, NULL, &err);
err = clBuildProgram(program_updateHx, 0, NULL, NULL, NULL, NULL);
kernel_updateHx = clCreateKernel(program_updateHx,"opencl_updateHx", &err);
```

Figure 4.6. Create and Build (program and kernel) functions

The kernel of updating FDTD E and H fields is as shown below in figure (4.7.)

```
const char *KernelSource_updateHx = "                              \n" \
"__kernel void opencl_updateHx(                                    \n" \
"   int n,                                                         \n" \
"   double dz,                                                     \n" \
" __global double *Ey,                                             \n" \
" __global double *Hx,                                             \n" \
" __global double *mHx,                                            \n" \
"   int src,                                                       \n" \
"   double Esrc)                                                   \n" \
"{                                                                 \n" \
"   int tid = get_global_id(0);                                    \n" \
"   double e1=0.,e2=0.,e3=0.;                                      \n" \
"   if (tid<n-1) {                                                 \n" \
"   Hx[tid]=Hx[tid]+mHx[tid]*(Ey[tid+1]-Ey[tid])/dz;               \n" \
"   }                                                              \n" \
"   if(tid==(n-1)) {                                               \n" \
"   e3 =e2,e2=e1,e1=Ey[n-1];                                       \n" \
"   Hx[src-1]=Hx[src-1]-(mHx[src-1]/dz)*Esrc;                      \n" \
"   Hx[tid] = Hx[tid] + mHx[tid]*(e3-Ey[tid])/dz;                  \n" \
" }     }                                                          \n" \
"\n";
```

Figure 4.7. FDTD Updating Kernel

4.2.3.4 Execution. For the execution step, arguments to the kernel need to be set, and the command queue needs to be set as well. An example is shown below in figure (4.8.):

```
Size_t global_work_size[1] , local_work_size[1];

Global_work_size[1] = Nz;

Local_work_size[1] = Nz/2;

err |= clSetKernelArg(kernel_updateHx, 0, sizeof(unsigned int), &Nz);

err |= clSetKernelArg(kernel_updateHx, 1, sizeof(double), &dz);

err |= clSetKernelArg(kernel_updateHx, 2, sizeof(cl_mem), &dev_Ey);

err |= clSetKernelArg(kernel_updateHx, 3, sizeof(cl_mem), &dev_Hx);

err |= clSetKernelArg(kernel_updateHx, 4, sizeof(cl_mem), &dev_mHx);

err |= clSetKernelArg(kernel_updateHx, 5, sizeof(unsigned int), &src);

err |= clSetKernelArg(kernel_updateHx, 6, sizeof(double), &Esrc[T])


err = clEnqueueNDRangeKernel(commands, kernel_updateHx, 1, NULL, &global, NULL, 0, NULL,

                                        NULL);
```

Figure 4.8. Kernel's Arguments

4.3. Performance of One-, and Two-Dimensional FDTD Problems

Three platforms have used to perform the FDTD computation using OpenCL as showing in the following figure (4.9.)

```
OpenCL platform information, found 3 platforms:

Platform 0:
        Platform name:      NVIDIA CUDA
        Platform version:   OpenCL 1.2 CUDA 11.2.162
        Platform profile:   FULL_PROFILE
        Platform extensions: cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_lo
extended_atomics cl_khr_fp64 cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing cl_nv_compiler_optio
ma_unroll cl_nv_copy_opts cl_khr_gl_event cl_nv_create_buffer cl_khr_int64_base_atomics cl_khr_int64_extende
vice_uuid

Platform 1:
        Platform name:      Intel(R) CPU Runtime for OpenCL(TM) Applications
        Platform version:   OpenCL 2.1 LINUX
        Platform profile:   FULL_PROFILE
        Platform extensions: cl_khr_icd cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomic
ocal_int32_extended_atomics cl_khr_byte_addressable_store cl_khr_depth_images cl_khr_3d_image_writes cl_inte
fp64 cl_khr_image2d_from_buffer cl_intel_vec_len_hint

Platform 2:
        Platform name:      Portable Computing Language
        Platform version:   OpenCL 1.2 pocl 1.6, None+Asserts, LLVM 9.0.1, RELOC, SLEEF, DISTRO, POCL_DEBUG
        Platform profile:   FULL_PROFILE
        Platform extensions: cl_khr_icd
```

Figure 4.9. OpenCL platforms

The OpenCL function clGetDeviceInfo , as in figure(4.10), can be used to get important information about the device available, such as: compute units, maximum frequency, global memory, local memory.

```
cl_uint device_max_compute_units;

cl_uint device_max_clock_frequency;

cl_ulong device_global_mem_size;

cl_ulong device_local_mem_size;

err = clGetDeviceInfo(device_id,

CL_DEVICE_COMPUTE_UNITS,sizeof(device_max_compute_units),
```

Figure 4.10. OpenCL platforms

In this thesis, three platforms (CUDA, Intel CPU, and PCL) were used to run the OpenCL code. The portable computing language (Platform 2), as in following table (4.2.), has same global mem size as the Intel with a higher operating frequency (3400 MHz). So, one would expect to get more speed up with the PCL platform. Also, we should mention that both the Intel CPU and the PCL backends are running on the CPUs on the machine with different implementations of OpenCL.

	Device Specifications	Platform 0 GPU CUDA	Platform 1 Intel CPU	Platform 2 Portable Computing Language
1	Device max compute units	2 units	32 units	32 units
2	Device max clock frequency	1252 MHz	2600 MHz	3400 MHz
3	Device global memory size	2097479680 bytes	84320153600 bytes	82172669952 bytes
4	Device local memory size	49152 bytes	32768 bytes	262144 bytes

Table 4.2. OpenCL platforms

The FDTD code was tested with the three platforms and the results verified with zero numeric error. As expected, the time of platform 2 (PCL) was faster as the device is more powerful. We were able to increase the speed up to 22.4 with 10000 iterations and 150000 grid sizes. The following table (4.3) showed the speedup of the three platforms with different grid sizes.

10000 iterations						
	Grid Size	CPU-C	CUDA	Intel-CPU	PCL	Speedup CPU-PCL
1	217	37	494	560	383	0.1
2	617	78	488	492	385	0.2
3	4017	1287	499	550	447	2.9
4	10000	4323	489	525	480	9.0
5	20000	4873	556	508	482	10.1
6	40000	5920	588	531	518	11.4
7	80000	7669	661	582	477	16.1
8	100000	8950	680	508	516	17.3
9	150000	11554	749	532	515	22.4

Table 4.3. OpenCL Performance

CHAPTER 5

5. CONCLUSIONS AND FUTURE WORK

5.1 FDTD Method

The Finite-Difference Time-Domain (FDTD) method is one simple way to find the numerical solutions of Maxwell's equations. The partial spatial and temporal derivatives form of Maxwell's equations can be approximated by using second-order central differences along with Yee's algorithm that offsets electrical and magnetic field's components within half time and space step offset. For example, in three-dimensional FDTD space, consider the point (i, j, k) the electric field components $E_{x,} E_y, and\ E_z$ will be staggered at $(i + \Delta x, j + \Delta y, k + \Delta z)$ and the magnetic field components $H_{x,} H_y, and\ H_z$ will be staggered at $\left(i + \frac{\Delta x}{2}, j + \frac{\Delta y}{2}, k + \frac{\Delta z}{2}\right)$ or vice versa, as explained previously in Figure (2.4.). Depending on the features and the materials of the device under simulation, we may need thousands of grid cells (or more) to implement both the electrical and magnetic fields' components in the three-dimensional space. All cells need to be updated every half time step ($\frac{\Delta t}{2}$) using the set of updating equations illustrated before in Figure (2.17.). Knowing that the time step (dt) must be chosen in very small increments to satisfy the stability condition (courant condition)[3], the total number of iterations depends on the device and its properties. For example, highly resonant devices may require thousands of times as many time steps to be modeled.

Yee's algorithm is elegant as it implicitly satisfies the two divergence equations of Maxwell's set and the physical boundary conditions. Since field components are modeled physically in different locations, caution needs to be taken in case the components reside in different materials.

The FDTD method, as with other numerical computational methods, needs large memory size and high-speed computational devices to efficiently simulate complicated EM engineering problems. Engineers and researchers have studied many ways to optimize and speed up FDTD performance. High Performance scientific Computers (HPC) open new opportunities for researchers to develop and

expand FDTD applications with acceptable running times. Graphics Processing Units (GPUs), (especially since the release of NVIDIA's Compute Unified Device Architecture (CUDA)), have been increasingly used for numerical computing methods.

In this thesis we found that the updated equations of the fields' components are related independently to the values of each other making the FDTD well suited to be parallelized and executed using parallel architecture's computation devices. We have been testing the one- and two-dimensional FDTD code on different platforms (Compute Unified Device Architecture CUDA and Open Computing Language OpenCL).

In the case of the CUDA environment, the FDTD CPU program launches the kernel grid along with execution configurations to run in parallel. The blocks of the grid are divided and distributed to be executed on multithreaded streaming multiprocessors (SMs) of the NVIDIA GPU. In the case of NVIDIA Quadro P400 (256 core), the elapsed time was recorded using perf tool, CUDA events and PAPI. The GPU-GPU performance (of 217, 617, and 4017 gride size with 400 steps) was measured and the speedup (of 43, 117, and 411) was recorded.

The CPU-GPU comparison showed weak performance of GPU especially with problem that were generally small in terms of grid size. This can be attributed to the time needed to copy fields' values to GPU memory and back when done. Thus, CUDA is best suited to leads to large FDTD problems.

For the OpenCL platform, the FDTD update equation's loops are transferred to kernels that run concurrently on GPUs. The grid cells represented by **work-items** (the equivalent of threads in CUDA) are grouped into **Workgroups** (the equivalent of thread blocks in CUDA) and assigned to one or more computing units (CUs) on the GPU.

Finally, using CUDA to implement parallel FDTD code is extremely powerful, but it does works only with NVIDIA-Compatible devices. However, OpenCL has an open architecture, allowing the code to be written in high level language like C/C++ easily executed on any OpenCL- GPUs available. The code is portable, and the programmer has more flexibility to choose the most suitable GPU-device available.

The serial C code was implemented in OpenCL language and tested within the available platforms (CUDA, Intel CPU, and PCL). Different problem sizes with 10000 iterations showed that up to 22.3 of speed factor can be obtained in case of 150000 grid size.

5.2 Future Work

5.2.1. FDTD Approximation

Section 2.2. to approximate the partial derivatives by finite differences we used the Taylor series expansion of the function over a negligibly small interval in the x-direction (holding all other dimensions constant). The forward finite differences approximation of the derivative is the Taylor series expansion of: $f\left(\times+\frac{\Delta\times}{2}\right)=f(x)+\frac{\Delta\times}{1!}\frac{\partial f(\times)}{\partial x}+\frac{(\Delta\times)^2}{2!}\frac{\partial^2 f(\times)}{\partial x^2}+\frac{(\Delta\times)^3}{3!}\frac{\partial^3 f(\times)}{\partial x^3}+\cdots.$

A path of future exploration would be to take the Taylor expansion of the function at the four sample points nearest to x to obtain:

$$f\left(\times+\frac{\Delta\times}{2}\right)=f(x)+\frac{\Delta\times}{1!}\frac{\partial f(\times)}{\partial x}+\frac{(\Delta\times)^2}{2!}\frac{\partial^2 f(\times)}{\partial x^2}+\frac{(\Delta\times)^3}{3!}\frac{\partial^3 f(\times)}{\partial x^3}+\cdots. \quad (5.1)$$

$$f\left(\times+\frac{3\Delta\times}{2}\right)=f(x)+\frac{3\Delta\times}{1!}\frac{\partial f(\times)}{\partial x}+\frac{(3\Delta\times)^2}{2!}\frac{\partial^2 f(\times)}{\partial x^2}+\frac{(3\Delta\times)^3}{3!}\frac{\partial^3 f(\times)}{\partial x^3}+\cdots. \quad (5.2)$$

$$f\left(\times-\frac{\Delta\times}{2}\right)=f(x)-\frac{\Delta\times}{1!}\frac{\partial f(\times)}{\partial x}+\frac{(\Delta\times)^2}{2!}\frac{\partial^2 f(\times)}{\partial x^2}-\frac{(\Delta\times)^3}{3!}\frac{\partial^3 f(\times)}{\partial x^3}+\cdots. \quad (5.3)$$

$$f\left(\times-\frac{3\Delta\times}{2}\right)=f(x)-\frac{3\Delta\times}{1!}\frac{\partial f(\times)}{\partial x}+\frac{(3\Delta\times)^2}{2!}\frac{\partial^2 f(\times)}{\partial x^2}-\frac{(3\Delta\times)^3}{3!}\frac{\partial^3 f(\times)}{\partial x^3}+\cdots. \quad (5.4)$$

Now subtracting (5.1) from (5.3) and (5.2) from (5.4) yields:

$$f\left(\times+\frac{\Delta\times}{2}\right)-f\left(\times-\frac{\Delta\times}{2}\right)=\Delta\times\frac{\partial f(\times)}{\partial x}+\frac{2}{3!}\left(\frac{\Delta\times}{2}\right)^2\frac{\partial^3 f(\times)}{\partial x^3}+\cdots \quad (5.5)$$

$$f\left(\times+\frac{3\Delta\times}{2}\right)-f\left(\times-\frac{3\Delta\times}{2}\right)=3\,\Delta\times\frac{\partial f(\times)}{\partial x}+\frac{2}{3!}\left(\frac{3\Delta\times}{2}\right)^2\frac{\partial^3 f(\times)}{\partial x^3}+\cdots \quad (5.6)$$

Solving the above equations for $\frac{\partial f(\times)}{\partial x}$ and eliminating $\frac{\partial^3 f(\times)}{\partial x^3}$. The result is:

$$\frac{\partial f(\times)}{\partial x}=\left(\frac{9f\left(\times+\frac{\Delta\times}{2}\right)-f\left(\times-\frac{\Delta\times}{2}\right)}{8\Delta\times}\right)-\left(\frac{1f\left(\times+\frac{3\Delta\times}{2}\right)-f\left(\times-\frac{3\Delta\times}{2}\right)}{24\Delta\times}\right)+O(\Delta\times)^4 \quad (5.7)$$

For future work, using Equation (5.7) to solve Maxwell's equations should result in more accuracy, or the ability to use a larger grid size to obtain the same accuracy.

5.2.2. CUDA

Before CUDA 6, both the CPU and GPU had their own separated memories and the programmer needed to explicitly copy the data from CPU memory to the GPU memory for the kernel to do the computation process. This process is expensive. With CUDA 6, a unified memory concept has emerged whereby both GPU and CPU share some memory in common and copying data is no longer required. In the future it will be interesting to run the FDTD code using a CUDA 6 GPU.

5.2.3. OpenCL

Future work with the use of OpenCL could include the exploration of optimizing the OpenCL code by saving the fields' data using local memory rather than global memory. Since local memory is closer to the cores it would be expected to be faster than global memory. Writing codes to take advantage of local memory may lead to faster overall performance.

5.2.4. Parallelizing Techniques

Although this thesis focused on GPU parallelization, other techniques could be explored. Options include using other methods like pthreads, OpenMp, and MPI. This would allow the comparison the speedup and the efficiency among them.

5.3 Conclusion

This thesis explored the FDTD method for electromagnetic simulation. It looked at one-, and two-dimensional problems and explored the speedup and efficiency of parallelizing the code, most notably using Graphics Processing Units and CUDA or OpenCL. The accuracy of the parallel code was the same as the serial code and the speedup increased for larger problem sets. Overall, electromagnetic simulation is well suited to parallel processing using finite difference methods.

This thesis is intended to provide the basis for simulating electromagnetic problems in one, two or three dimensions. and the results show that the use of GPUs for such problems can be highly beneficial with respect to reducing the computational time.

BIBLIOGRAPHY

1. A. Taflove and S. C. Hagness, Computational Electrodynamics: The Finite Difference Time Domain Method, 3rd edition. Norwood, MA: Artech House Publishers, 2005.

2. K. S. Yee, ‘‘Numerical solution of initial boundary value problems involving Maxwell’s equations in isotropic media,’’ IEEE Transactions on Antennas and Propagation, vol. 14, pp. 302–307, 1966.

3. R. Courant, K. Friedrichs, and H. Lewy, ‘‘On the partial difference equations of mathematical physics,’’ IBM Journal of Research and Development, vol. 11, no. 2, pp. 215–234, 1967.

4. J. A. Kong, Electromagnetic Wave Theory. Cambridge, MA: EMW Publishing, 2000.

5. Elsherbeni and Demir – The Finite-Difference Time-Domain Method for Electromagnetics with MATLAB Simulations, 2nd Edition (2015)

6. Two-Dimensional Finite-Difference Time-Domain (FDTD) with MATLAB. EMPossible. (2020, October 14). https://empossible.net/.

7. Stewart, James (2012). Calculus - Early Transcendentals (7th ed.). Brooks/Cole Cengage Learning. p. 1122. ISBN 978-0-538-49790-9.

8. “AMPERE'S LAW.” Https://Web.iit.edu, web.iit.edu/.../pdfs/Amperes_law.pdf.

9. NVIDIA Developer. 2021. CUDA Zone. [online] Available at: <https://developer.nvidia.com/cuda-zone>.

10. NVIDIA. (2021). Cuda C++ Programming Guide (Vol. 11.4). NVIDIA.

11. Dachuan, S., 2013. GPU-BASED ACCELERATION ON ACENET FOR FDTD METHOD OF ELECTROMAGNETIC FIELD ANALYSIS. Nova Scotia, pp.99-100.

12. Z. L. He, K. Huang, Y. Zhang, Y. Yan, C. H. Liang, "Study on High Performance of MPI-Based Parallel FDTD from WorkStation to Supercomputer Platform", International Journal of Antennas and Propagation, vol. 2012, Article ID 659509, 7 pages, 2012.

13. https://doi.org/10.1155/2012/659509

14. Min Li, "Parallel FDTD simulation using CUDA," 2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering, Changchun, China, 2010, pp. 196-199, doi: 10.1109/CMCE.2010.5610469.

15. Weaver, V. (2021, March). Lecture 15. Cluster computing.

16. Weaver, V. (2021, March). Lecture 3. P (2-9) Cluster computing.

17. Group, K. O. C. L. W. (2021, June 30). OpenCL API Specification, Version v3.0.8. The OpenCL™ Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html.
18. Tompson, Jonathan; Schlachter, Kristofer (2012). "An introduction to the OpenCL Programming Model." New York University Media Research Lab. Archived from the original (PDF) on July 6, 2015. Retrieved July 6, 2015.

19. Weaver, V. (2021, March). Lecture 17. Cluster computing.

20. "AMD APP SDK OpenCL User Guide." Https://Developer.amd.com, AMD, Aug. 2015, developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf.

21. "AMD APP SDK OpenCL User Guide." Https://Developer.amd.com, AMD, Aug. 2015, developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf.

22. Feynman, Richard P." The Feynman Lectures on Physics Vol. II" www.feynmanlectures.caltech.edu. Retrieved 2020-11-07.

23. Maxwell, J. C. (n.d.). *File:on physical lines of force.pdf - Wikimedia commons*. https://commons.wikimedia.org/wiki/File:On_Physical_Lines_of_Force.pdf.

APPENDIX A

One-Dimensional FDTD sample code in MATLAB

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  INITIALIZE MATLAB
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
close all; clc; clear all.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  DEFINE UNITS AND CONSTANTS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% units
meters      = 1.
centimeters = 1e-2 * meters.
millimeters = 1e-3 * meters.
inches      = 2.54 * centimeters.
feet        = 12 * inches.


seconds     = 1.
hertz       = 1/seconds.
kilohertz   = 1e3 * hertz.
megahertz   = 1e6 * hertz.
gigahertz   = 1e9 * hertz.


% CONSTANTS
e0 = 8.85418782e-12 * 1/meters.
```

```
u0 = 1.25663706e-6  * 1/meters.

N0 = sqrt(u0/e0).

c0 = 299792458 * meters/seconds.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% DASHBOARD

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% SOURCE PARAMETERS

fmax = 2 * gigahertz.

NFREQ= 1000.

FREQ = linspace(0,fmax,NFREQ).


%DEVICE PARAMETERS

 er1 = 12.0;        %relative permitivity device

 er2 = 1;           %relative permitivity air

 ur1 = 1.

 ur2 = 1.

 L   = 3.0 * centimeters;  % thickness of the slab


% GRID PARAMETERS

NRES   = 20;       % number of points per wavelength

NRES_D = 2.

ermax  = max([er1 er2]).

urmax  = max([ur1 ur2]).
```

```
nmax   = sqrt(ermax*urmax).

NSPC   = ([100 100]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% CALCULATE OPTIMIZED GRID

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% % COMPUTE GRID RESOLUTION

lambda_min = c0/fmax/nmax.

dz1       = lambda_min/NRES.

dz2        = L/NRES_D.

dz        = min([dz1 dz2]).


%

% % SNAP GRID TO CRITICAL DIMENSIONS

nz = ceil(L/dz).

dz = L/nz.


% % CALCULATE NUMBER OF GRID CELLS

Nz = nz + sum(NSPC) + 3.


% COMPUTE GRID AXES (FOR GRAPHICS)

xa = [0:Nz-1]*dz.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%% BUILD DEVICE ON GRID
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% INITIALIZE MATERIALS
ERyy = er2* ones(1,Nz).
URxx = ur2* ones(1,Nz).
nz1  = 2 + NSPC(1)+1.
nz2  = nz1 + nz -1.

ERyy(nz1:nz2) = er1.
URxx(nz1:nz2) = ur1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% CALCULATE SOURCE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% COMPUTE TIME STEP
dt   = dz/(2*c0).

% Calculate GAUSSIAN
tau  = sqrt(2.3)/pi/fmax.
t0   = 5 * tau.

%CALCULATE NUMBER OF STEPS
d     = (Nz*dz).
```

```
tg    = 2*t0 + 5*nmax*d/c0.

STEPS = ceil(tg/dt).

%STEPS = 50000.

% CALCULATE GAUSSIAN SOURCE

t     = [0:STEPS - 1] *dt.

delay  = (dz/(2*c0))+ dt/2;          % total delay between E and H

A     = -1;       %-sqrt(eps_rltv_src/mu_rltv_src)

Esrc   = exp(-((t - t0) / tau).^2).

Hsrc   = A*exp(-((t-t0+delay)/tau).^2);     % H field source


src = 2.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Initialize Fourier transforms

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

K      = exp((-1i * 2 * pi * dt) * FREQ).

Eref    = zeros(1,NFREQ).

Etrn    = zeros(1,NFREQ).

Src     = zeros(1,NFREQ).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% CALCULATE UPDATE COEFFICIENTS

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% UPDATE COEFFICIENTS
```

```
mHx =  c0*dt./URxx.

mEy =  c0*dt./ERyy;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% INITIALIZE FDTD DATA

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% INITIALIZE FIELDS

Hx = zeros(1,Nz);

Ey = zeros(1,Nz);


h3=0;h2=0;h1=0;e3=0;e2=0;e1=0;

tic

for T = 1 :2



  % Update H from E


  for nz = 1 : (Nz-1)

     Hx(nz) = Hx(nz) + mHx(nz)*(Ey(nz+1) - Ey(nz))/dz;

  end

   e3=e2;e2 = e1;e1 = Ey(Nz);

  % Handle H source

  Hx(src-1)= Hx(src-1)-(mHx(src-1)/dz) * Esrc(T);

  Hx(Nz) = Hx(Nz) + mHx(Nz) * (e3 - Ey(Nz))/dz;
```

```
h3 = h2 ; h2 = h1; h1 = Hx(1);


% Update E from H


Ey(1) = Ey(1) + mEy(1) * (Hx(1) - h3)/dz;

for nz = 2 : Nz

   Ey(nz) = Ey(nz) + mEy(nz)*(Hx(nz) - Hx(nz - 1))/dz;

end

Ey(src)=Ey(src) - ((mEy(src)/dz) * Hsrc(T));
```

BIOGRAPHY OF THE AUTHOR

Atheer Oufi was born in Baghdad, Iraq on March 09, 1978. He graduated from Jumhouriya High School in 1996. He attended the University of Technology of Iraq and graduated in 2000 with a bachelor's degree in Computer and Systems Engineering. He moved to Maine, in the United States, as a refugee in 2016 and entered the Electrical Engineering graduate program at The University of Maine in the Fall of 2019. He is a candidate for the Master of Science degree in Electrical Engineering from the University of Maine in August 2021.