INVESTIGATION OF GENERAL-PURPOSE COMPUTING ON GRAPHICS PROCESSING UNITS AND ITS APPLICATION TO THE FINITE ELEMENT ANALYSIS OF ELECTROMAGNETIC PROBLEMS

BY

HUAN-TING MENG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor Jianming Jin, Chair
Professor Jose Schutt-Aine
Associate Professor Deming Chen
Assistant Professor Andreas Kloeckner

ABSTRACT

In this dissertation, the hardware and API architectures of GPUs are investigated, and the corresponding acceleration techniques are applied on the traditional frequency domain finite element method (FEM), the element-level time-domain methods, and the nonlinear discontinuous Galerkin method. First, the assembly and the solution phases of the FEM are parallelized and mapped onto the granular GPU processors. Efficient parallelization strategies for the finite element matrix assembly on a single GPU and on multiple GPUs are proposed. The parallelization strategies for the finite element matrix solution, in conjunction with parallelizable preconditioners are investigated to reduce the total solution time. Second, the element-level dual-field domain decomposition (DFDD-ELD) method is parallelized on GPU. The element-level algorithms treat each finite element as a subdomain, where the elements march the fields in time by exchanging fields and fluxes on the element boundary interfaces with the neighboring elements. The proposed parallelization framework is readily applicable to similar element-level algorithms, where the application to the discontinuous Galerkin time-domain (DGTD) methods show good acceleration results. Third, the element-level parallelization framework is further adapted to the acceleration of nonlinear DGTD algorithm, which has potential applications in the field of optics. The proposed nonlinear DGTD algorithm describes the third-order instantaneous nonlinear effect between the electromagnetic field and the medium permittivity. The Newton-Raphson method is incorporated to reduce the number of nonlinear iterations through its quadratic convergence. Various nonlinear examples are presented to show the different Kerr effects observed through the third-order nonlinearity. With the acceleration using MPI+GPU under large cluster environments, the solution times for the various linear and nonlinear examples are significantly reduced.

ACKNOWLEDGMENTS

To Professor Jian-Ming Jin and Professor Deming Chen who guided me through all these years.

To Professor Jose Schutt-Aine and Professor Andreas Kloeckner who taught and advised me as my PhD committee members.

To all the teachers and mentors whom I learn from.

To all the friends and people whom I receive support from.

To my parents, my brother, and my family.

To my dear Veola and the two lovely bears.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 The GPU Environment 1
1.2 GPU Acceleration of the Finite Element Method 3
1.3 GPU Acceleration of the Time Domain Methods 5
1.4 GPU Acceleration of the Nonlinear Discontinuous Galerkin Method 6
CHAPTER 2 THE GPU ENVIRONMENT 9
2.1 Introduction 9
2.2 The GPU Architecture 9
2.3 The CUDA API Architecture 10
2.4 Performance Gauging and Acceleration Techniques 12
2.4.1 Performance Gauging 12
2.4.2 Acceleration Techniques 13
2.5 Figures 15
CHAPTER 3 GPU ACCELERATION OF THE FINITE ELEMENT METHOD 16
3.1 Introduction 16
3.2 Finite Element Formulation 16
3.3 GPU Implementation — Assembly 18
3.3.1 The Traditional CPU Approach 18
3.3.2 Parallelization Scheme 1 — One Element per Thread on a Single GPU with Atomic Assembly 20
3.3.3 Parallelization Scheme 2 — One Element per Thread on Single GPU with Assembly-by-DOF 20
3.3.4 Parallelization Scheme 3 — One DOF per Thread on Multi-GPUs 22
3.4 GPU Implementation — Matrix Solution 23
3.4.1 Iterative Solver on Single GPU 23
3.4.2 Parallelized Preconditioners 25
3.4.3 Iterative Solver on Multi-GPUs 27
3.5 Examples and Speedups 27
3.5.1 Single-GPU Solution + Serial Preconditioner 27
3.5.2 Single-GPU with Atomic Assembly + Single-GPU Solution 29
3.5.3 Single-GPU Assembly-by-DOF + Solution with Parallel Preconditioner 31
3.5.4 Multi-GPU Assembly + Multi-GPU Solution + Parallel Preconditioner 32
3.6 Figures and Tables 36

CHAPTER 4 GPU ACCELERATION OF THE ELEMENT-LEVEL TIME-DOMAIN METHODS 48
4.1 Introduction 48
4.2 DFDD-ELD Formulation 48
4.3 GPU Implementation 52
4.3.1 Time-Marching Step Breakdown 52
4.3.2 Parallelization Using CUDA 53
4.3.3 Random Access Acceleration 55
4.3.4 Incorporation of the Local Time-Stepping Technique 56
4.3.5 Parallelization Using Hybrid MPI-CUDA 57
4.3.6 Extension to Arbitrarily Large Finite Arrays 58
4.4 DGTD Formulation and Acceleration Comparison 59
4.4.1 DGTD-Central 59
4.4.2 DGTD-Upwind 61
4.4.3 GPU Timing and Acceleration Comparison 62
4.5 Examples and Speedups 64
4.5.1 Parallel-Plate Transmission Line 64
4.5.2 Microstrip Patch Array 65
4.5.3 Antipodal Fermi Antenna 67
4.5.4 Monopole Antenna Array 69
4.5.5 Vivaldi Antenna Array 70
4.5.6 Three Vias in a Three-Layer PCB 70
4.5.7 Branch-Line Coupler 71
4.6 Figures and Tables 72
CHAPTER 5 GPU ACCELERATION OF THE NONLINEAR DISCONTINUOUS GALERKIN METHOD 89
5.1 Introduction 89
5.2 Formulation 89
5.2.1 Nonlinear Formulation Using the Newton-Raphson Method 91
5.2.2 Comparison between the Newton-Raphson and the Fixed-Point Methods 94
5.3 GPU Implementation 95
5.4 Examples and Speedups 96
5.4.1 Coaxial Cable Demonstrating the Self-Phase Modulation and the Third-Harmonic Generation 96
5.4.2 Photonic Amplifier Demonstrating the Frequency Mixing Effect 98

5.4.3 Propagation in Bulk Medium Demonstrating the Self-Focusing Effect 100
5.5 Figures and Tables 102
CHAPTER 6 CONCLUSION AND FUTURE RESEARCH 110
REFERENCES 113

CHAPTER 1
INTRODUCTION

1.1 The GPU Environment

General-purpose computing on graphics processing units (GPGPU) computing environments, such as the Compute Unified Device Architecture (CUDA) [1] by NVIDIA, provides easy access to massively parallel programming, which was previously available only in the form of expensive supercomputers and large clusters, to the common users with limited budget and resources. Before the introduction of the GPGPU concept, and more generally the concept of multi-threading computing, the majority of the software applications were serial programs, designed largely under the assumption of the never-failing Moore's law and the corresponding ever-growing CPU clock rate to boost their runtime. However, due to various physical design limitations, the increase in the CPU clock rate has slowed down during the early 2000s, and alternative solutions need to be explored to keep up with the growing complexity of modern software applications [1].

Two schools of thought have emerged as the potential candidates for the solution of the problem, which are vastly different from each other. These two diverging trends are the multi-core architectures and the many-core architectures. Advocates of multi-core systems, borrowing from the concept of large supercomputers, argue that by incorporating multiple existing standalone CPU processors into a single CPU chip, a CPU can practically behave like a small cluster, gaining computing capability by putting different processes of a software program onto the different processor cores of the CPU to lighten the load of a single processor, and thus cutting down the total runtime. This architecture is classified as multiple instruction, multiple data (MIMD) in Flynn's taxonomy [2], where the processors execute asynchronously and independently from each other using the same or different instructions at any time. On the other hand, the idea behind many-core systems derives from existing GPU architectures, where numerous light computing units are employed for graphics processing and rendering. The idea is that since many applications and algorithms involve repetitive processing of independent data, it is preferable to parallelize the processing of these data through a large batch of identical computing units on a single chip, even at the cost of reducing the capability and the speed of

each of these units. This architecture is classified as single instruction, multiple data (SIMD), where the processors act in unity to carry out a single instruction, each one processing a small fraction of the software process using different data. The GPUs are developed based on the latter system, and are first put to use in graphics rendering, where each ray of light takes the same identical instructions from the graphics engine, but is processed independently from the others on separate data.

To utilize the GPU architecture for general computing purposes, existing graphical shading languages, such as DirectX and OpenGL, have been exploited to map the data and algorithm of interest onto the various graphics-specific texture maps, shader models and graphics pipelines [3]-[6]. One of the first attempts of a high-level general API adapting to the existing hardware shading architecture was the C for Graphics (Cg) language by NVIDIA, which was an extended C language designed to facilitate the implementation of general purpose computing on GPU by compiling higher-level programs from the users and automatically mapping them into the existing graphical shading languages, thereby eliminating the need for the users to program the GPU by directly controlling the low-level programmable vertex and fragment processors [7].

While high-level shading languages like Cg have significantly simplified general-purpose programming on GPUs, the fundamental functionality of the GPU is still rooted in graphics processing, with high-level shading languages serving merely as translators to allow general-purpose users to utilize the existing architecture for graphics. The real revolution came with the introduction of the first dedicated GPGPU API, CUDA, which has not only transformed the way general users program GPUs, but has also shifted the philosophy of GPU hardware design from the dedicated graphics pipelines to the more general many-core computing frameworks [8]. Instead of relying on the shading languages as a medium of mapping the underlying algorithms to the hardware graphics engines, the users can now directly control the various hierarchies of GPU resources for general-purpose computing. Since the inception of GPGPU and NVIDIA's proprietary CUDA, other contestants, notably the Open Computing Language (OpenCL) framework by Apple Inc. and the Khronos Group, have also gained popularity due to their open-source licensing and cross-vendor GPU support. Despite the difference in the framework and the language, these two dominant GPGPU APIs share the same underlying software abstractions to the GPU hardware and the parallelization philosophy.

With the enablement of GPGPU through dedicated API, GPUs are able to assist the parallelization of general-purpose algorithms in two ways. One way is to view the addition of one or more GPUs on a personal workstation as an alternative to medium-sized clusters; the latter are bulky and expensive while the former serve as a budget-friendly replacement. The other way is to view the incorporation of one or more GPUs per node onto existing supercomputers as an enhancement to the existing computing infrastructure, where the peak computational capability of the supercomputers can be significantly boosted by the utilization of GPUs. In this dissertation, we will introduce the GPU hardware architecture and the software API based on NVIDIA's GPU and its proprietary CUDA API.

1.2 GPU Acceleration of the Finite Element Method

GPGPU offers the capability to accelerate the solution process of computational electromagnetic analysis. Due to their parallelized architecture, GPUs are highly desirable for the acceleration of computationally intensive or highly parallelizable algorithms such as the method of moments, as well as for numerical methods where data communication overhead tends to be light, such as the finite-difference time-domain method on a uniform grid [9]-[22]. However, the performance is more limited for the finite element method because the computation workload for calculating elemental matrices, especially when low-order finite elements are used, is relatively small compared to the more intensive data communication needed for element assembly and matrix solution. Due to the communication-intensive nature [23], both the finite element assembly and the solution phases cannot be implemented via fine-grained many-core GPU processors in a straightforward manner. Most research has been focused on either explicit algorithms or the standard finite element method [24]-[30]. For the standard finite element method, researchers have concentrated on the parallelization of the assembly and the solution phases.

The assembly of finite element matrices on GPUs has been studied in prior work [28]-[30]. Novel algorithms have been proposed to alleviate problems associated with the serial nature of the assembly process. Cecka, Lew, and Darve [28] have compared multiple strategies for the finite element assembly of two-dimensional structures. Markall et al. [29] have developed a "local matrix approach," which omits the assembly of the global system matrix, while introducing extra matrix mapping during the solution phase. Dziekonski et al. [30] have

parallelized the construction of local elemental matrices, and used atomic operations to assemble the elements into the final system matrix.

The solution of the general sparse finite element matrix systems on GPU has received great attention because of its importance. It is well known that the solution for a very large system of equations is best computed using an iterative method, and recently much research has been focused on the implementation of iterative methods using GPUs [31]-[39], such as the multigrid solver [24]; the conjugate gradient (CG) solver [32], [33], [38], [39], [40]; the biconjugate gradient (BiCG) solver [35]; and the generalized minimal residual method (GMRES) [33], [34]. Research on GPU-based iterative solvers is accompanied by the development of preconditioners on GPUs [35]-[39], such as the incomplete LU (ILU) preconditioner [35] and the multigrid preconditioner [36]. Xu et al. [37] and Benzi and Tuma [41] have sampled a wide variety of preconditioners from the family of the approximate inverse (AI) preconditioners on GPUs, which have also been investigated by Xu et al. [38] and Helfenstein and Koko [39].

All of the past research tended to focus on specific aspects of the finite element algorithm and numerical experiments have shown promising speedups by utilizing GPUs in various fields of study. The objective of our work is to carry out a comprehensive study to identify the bottlenecks in various phases of the implementation of the finite element method for electromagnetic problems, and propose practical remedies to alleviate these bottlenecks to achieve an effective GPU utilization [42]. Our work focuses on three practical goals. First, the modification to an existing CPU code has to be kept minimal. Second, the code's GPU memory usage has to be conserved. Third, good parallelization utilizing the hardware advantages of GPUs has to be achieved. In this work, we identify the bottlenecks in the GPU parallelization of the finite element method for electromagnetic analysis, and propose potential solutions to alleviate the bottlenecks. We first discuss efficient parallelization strategies for the finite element matrix assembly on a single GPU and on multiple GPUs. We then explore parallelization strategies for the finite element matrix solution, in conjunction with parallelizable preconditioners to reduce the total solution time. We show that with a proper parallelization and implementation, GPUs are able to achieve significant speedup over OpenMP enabled multi-core CPUs.

1.3 GPU Acceleration of the Time Domain Methods

To overcome the parallelization bottleneck of the finite-element method on GPU, domain-decomposition algorithms have been developed to break a problem into small subdomains to compute the solution in a divide-and-conquer fashion. In the frequency domain, the finite-element tearing and interconnecting (FETI) method [43]-[46] has been developed, which uses Lagrange multipliers on the subdomain interface to formulate a reduced global problem. In the time domain, the discontinuous Galerkin time-domain (DGTD) method [47]-[51] has received much attention due to its element-level decomposition, the ease of granular parallelization, and its wide adaptation in many disciplines. Kloeckner et al. [47] accelerated a nodal DGTD algorithm for the computation of scattering problems on a single GPU. Goedel et el. [24] first parallelized a higher-order nodal DGTD using multiple GPUs on a shared memory architecture, then adapted the Adams-Bashforth multirate algorithm [48] to further accelerate the time-marching process on GPUs. Dosopoulos et al. [51] accelerated a vector DGTD with a local time-stepping algorithm on GPUs in a distributed memory cluster environment. In general, finite element time-domain-based domain decomposition methods march the elements at a synchronous rate, with a time step restricted usually by the smallest element on the interfaces between the subdomains. For the element-level decomposition methods, this translates to the smallest element in the computation domain. The multirate and the local time-stepping algorithms are methods allowing different time step sizes to be used in different parts of the computation domain, i.e. different elements in the element-level decomposition scheme.

An alternative domain-decomposition algorithm for the time-domain FEM has been proposed by Lou and Jin [52] as the dual-field domain decomposition (DFDD) method. The DFDD divides a problem into smaller subdomains, then updates the electric and magnetic fields in a staggered fashion based on the dual-field second-order vector wave equations. The subdomains are coupled through equivalent surface currents rather than numerical fluxes as found in the DGTD algorithm. An explicit algorithm (the element-level time-domain method) is obtained when an element-level decomposition (ELD) is employed in the DFDD, and the resulting algorithm is referred to here as the DFDD-ELD algorithm [53]. The DFDD-ELD algorithm was compared and shown in [54] to have a comparable accuracy and efficiency against the two commonly used DGTD algorithms based on central and upwind fluxes. Recently, Li and

Jin [55] enhanced the capability of the DFDD algorithm by incorporating the treatment for lossy and dispersive materials using recursive convolutions. This further enriches the capability of the DFDD algorithm to handle complex materials.

Our work focuses on the adaptation of the DFDD-ELD method to the GPU environment. Specifically, a carefully arranged data structure and GPU thread allocation are discussed in detail to ensure a maximum bandwidth and a high acceleration on GPUs [56]. The recursive convolution algorithm is applied in the DFDD-ELD algorithm, where good parallelization can also be achieved on the GPU. Through its element-level subdomains, the DFDD-ELD computation can be easily mapped onto the GPU's granular processors and is thus highly parallelizable. Various electromagnetics problems are simulated to demonstrate the speedup and scalability of DFDD-ELD on a GPU cluster. With a careful GPU memory arrangement and thread allocation, we are able to achieve a significant speedup by utilizing GPUs in an MPI-based cluster environment. We have experimented with the local time-stepping technique to further speed up the time marching process. To test the parallelization techniques and the resulting efficiency, we also compared the acceleration of the DFDD-ELD algorithm on GPU against the acceleration of the DGTD-Upwind and the DGTD-Central algorithm, both with enhanced capability of handling lossy and dispersive materials by using the recursive convolution algorithm.

1.4 GPU Acceleration of the Nonlinear Discontinuous Galerkin Method

Nonlinear phenomena in computational electromagnetics generally involve changes in the material properties due to the presence of the electromagnetic fields. The changes in the material properties, whether permittivity, permeability, or conductivity, in turn modify the state of the original electromagnetic fields in the medium. Since the material properties and the contained fields interact with each other at any given instance, it is most natural to describe and model these interactions in the time domain, where at each time step, changes in the fields induce nonlinear modifications in both the material properties and the fields themselves. The changes induced in the current time step then set a completely different initial condition for the field-medium interaction in the following steps. While the nonlinear effect is best modeled in the time domain, many other phenomena, such as the dispersion effect in the aforementioned section, are best modeled in the frequency domain due to their intrinsic frequency-dependent properties.

Nonlinear optical effects are the study of optical properties in material. The optical phenomenon can be described with very high frequency electromagnetic waves in the order of hundreds of terahertz, and the intensity of the waves usually needs to be sufficiently large to change the optical properties of the material. One of the most studied and applied optical effects is the nonlinear optical Kerr effect [57]. The nonlinear optical Kerr effect describes the third-order interaction between the electric field and the permittivity of the material, and the third-order nonlinearity optical susceptibility is denoted using the quantity $\chi^{(3)}$. The optical Kerr effect produces a variety of nonlinear phenomena [57], [58], such as third-harmonic generation (THG), self-phase modulation (SPM), self-focusing, and frequency mixing. Given an excitation wave at a particular frequency or frequency band, the third-harmonic generation effect produces waves at triples of the original frequency. The self-phase modulation effect induces phase shift and broadens the spectral bandwidth of the excitation. Given an excitation of a pulse of wave, the self-focusing effect induces lensing in the bulk material and produces a waveguiding system counteracting diffraction, creating optical spatial solitons that travel without losing the original shape [59]. Lastly, given an excitation wave comprising two or more distinct frequencies, the frequency mixing effect generates waves at various combinations of the excitation frequencies.

Much investigation has been carried out for the simulation of the nonlinear optical effects [60]-[69]. An approach to model the nonlinear effect in 1-D propagation problems, such as the study of optical fibers, is the nonlinear Schrödinger equation (NLSE) [60]. It is a simple and efficient solution to 1-D nonlinear optical applications. However, the application of the NLSE is a reduction of the wave equation which relies on many assumptions, and is shown to produce inaccurate solutions in higher-dimensional problems [61]. In contrast, the simulation of the nonlinear optical effects with the FDTD algorithms [61]-[64] has gained much attention due to its straightforward implementation in the algorithms. In the proposed nonlinear FDTD algorithms, Yee's scheme leapfrogs the $\vec{H}$ and $\vec{D}$ fields, then uses the nonlinear constitutive relation of the $\vec{D}$ field to iteratively or explicitly arrive at an $\vec{E}$ field solution, which is then used to update the magnetic field at the next time step. For the simulation of the nonlinear optical effects with the FEM algorithms [65]-[69], Polstyanko and Lee [65] first implemented a 3-D FEM Kerr-type nonlinearity solver in the frequency domain. Brandao and Figueroa [67] used the beam propagation method to march nonlinear FEM in the time domain. Fisher et al. [69] introduced Debye time relaxation equations in the time-domain FEM to model finite nonlinearity

delay. To solve the nonlinearity convergence involved in the formulation, the existing nonlinear FEM algorithms either use simple fixed-point iteration methods, or omit iterations altogether with the incorporation of finite delays in the material modeling. However, fixed-point iteration methods in the traditional time-domain FEM face two challenges. First, as the size of the problem grows, the computation involved for the assembly and inversion of the nonlinear system matrix in each nonlinear iteration step grows exponentially. Second, as the nonlinearity in the problem intensifies, the number of fixed-point iterations in each time step becomes very large, where convergence might not even be guaranteed.

Our work focuses on the incorporation of the third-order Kerr instantaneous nonlinearity into the DGTD algorithm. Specifically, both the simple fixed-point and the Newton-Raphson iteration methods are presented, with quadratic convergence achieved for the Newton-Raphson method. The proposed algorithm is desirable for solving nonlinear problems using the DGTD algorithm, since at each nonlinear iteration step only small elemental matrices need to be assembled and inverted. In addition, through the Newton-Raphson method, the number of nonlinear iterations per time step is significantly reduced, especially when large nonlinearity is present, such as the study of high power laser optical devices. The proposed algorithm is geared towards instantaneous nonlinearity, assuming the medium is lossless and dispersionless [57]. This assumption is valid for most practical purposes where the material nonlinear responses are much faster than the optical pulses [70]. If the frequency of the source is very high, where the material nonlinear response time is comparable to the time resolution of the wave pulse, one could follow approaches similar to [61] and [64] to incorporate dispersion into the framework of the nonlinear DGTD algorithm using the auxiliary differential equation approach. To speed up the solution time, the MPI+GPU framework developed in this dissertation is adapted to accelerate the nonlinear DGTD algorithm.

CHAPTER 2
THE GPU ENVIRONMENT

2.1 Introduction

In this chapter, the GPU computing environment is introduced, with emphasis on the architectural design of the hardware and NVIDIA's proprietary GPU application programming interface (API), CUDA. Section 2.2 presents the GPU architecture. Section 2.3 details the CUDA API architecture. Section 2.4 provides the relevant acceleration techniques when applying GPUs for GPGPU purposes.

2.2 The GPU Architecture

The GPU architecture is fundamentally distinct from that of the CPU architecture. A CPU, denoted as the *host*, employs multi-core architecture in which only a few processing cores are present. The CPU is equipped with large on-chip cache and sophisticated control unit, as shown in Fig. 2.1(a). On the other hand, a GPU, denoted as the *device*, employs a many-core architecture containing hundreds of processing units which execute the same command simultaneously in an SIMD fashion. The GPU processing units are organized in batches of processing units, where each is managed by a small control unit accompanied by a small cache, as shown in Fig. 2.1(b). The batches of GPU processing units are arranged in hierarchical layers. Each GPU contains a number of *Streaming Multiprocessors* (SM), each of which contains a number of *Scalar Processors* (SP) as the actual fine-grained processing units of the GPU, as shown in Fig. 2.2(a). By executing the hierarchy of processing units in an organized fashion, the GPU carries out simultaneous computations on its SIMD, many-core system.

This hierarchical design can be found in GPU's various memory units as well, as shown again in Fig. 2.2(a). Each type of memory in the hierarchy has different latency, which is the number of clock cycles needed to access the specific memory. At the lowest level, each SP has its own local *registers* for fast caching. These registers have very low latency and can be accessed within a couple of clock cycles. They are very small and are intended to store temporary data during computation. The SPs in each SM are capable of accessing a common

shared memory residing in each SM. It is larger in size and is used as user-managed cache to load a pool of raw data shared by the SPs. This shared memory has a latency of several clock cycles. On top of the on-chip memories, all SPs in all SMs have access to the *global memory* residing off of the GPU chip. It is equivalent to the standard CPU DRAM, which is very large but has a latency of hundreds of clock cycles. These three memories are the most commonly used GPU memories in different stages of the computation process. In addition to these three types of memories, there also exist other less commonly-used memories for fine-tuning. For example, a read-only constant cache that is accessible by every SP, a texture memory and texture read-only cache which are suitable for grid-like access pattern, and remnants of the shader memory modules.

All modern GPUs carry architecture similar to that described above, but different GPUs come with different capabilities, whether it be the number of SMs or the amount of registers per SP, depending on their generation and model. For NVIDIA GPUs, small capability differences can be found in different GPU models of the same generation, whereas large hardware configurations can be found in different GPU architecture generations. These generation differences are classified according to a metric known as *compute capability* [71]. Despite the differences in the exact hardware configuration, from the hardware perspective, the acceleration capability of a GPU comes from the parallel architecture of its numerous processors and the appropriate use of the hierarchy of memories.

2.3 The CUDA API Architecture

CUDA is the proprietary software API for NVIDIA GPUs. The process flow of CUDA is as follows: once the host reaches the parallelizable portion of the program, the data on the host memory is sent to the different memory locations in the device. Next, CUDA will invoke a *kernel*, which calls the interface function to initialize and allocate controllable virtual elements arranged in a hierarchical order of a *grid* of *blocks* of *threads*, as shown in Fig. 2.2(b). When the kernel executes the instructions, the blocks and threads in a kernel are mapped a batch at a time onto the physical SMs and SPs. The blocks and threads can be controlled by their unique indices for parallel computation, where the various unique indices can be combined in different ways to access the memory or to carry out computation in unique patterns required by the underlying algorithm. At the end of each kernel, the computed results are transferred back from the device

memory to the host memory, thus completing the parallel portion of the program. This parallelization process can repeat as many times as needed until the end of the program runtime.

Analogous to the hardware memory hierarchy, the CUDA API also carries a list of accessible memories, as shown again in Fig. 2.2(b). A CUDA thread carries its own local registers and the threads in the same block share the same shared memory. The local registers and shared memory have very low latency, ranging from one to a few clock cycles. On the other hand, the device global memory that is accessible to all blocks and threads has very high latency, which is around several hundred clock cycles. Other physical memories such as the constant cache and texture memory can find their analogous entities in the CUDA API.

To map the conceptual blocks and threads onto the physical SMs and SPs, CUDA employs the concept of *warps*. Each warp contains 32 parallel threads, and is scheduled by the *warp scheduler* hardware to physically execute the same instruction on the same SMs together [71]. At any given moment there can exist one or more warps per SM. The number of warps that can be processed at the same time by an SM depends on the amount of on-chip registers and shared memory used by the kernel function, which is the amount of available memory resources per SM divided by the amount of memory resources requested by a single warp.

Through the concept of warps and the granular processing units, GPUs operate in a unique architecture called single instruction, multiple thread (SIMT) which differs slightly from the original SIMD architecture. While the SIMD architecture creates multiple datapaths for a single instruction at any given moment, the SIMT architecture bundles the numerous threads into warps that allow multiple independent instructions to be executed concurrently through the use of SMs, by assigning one or more warps to execute a separate instruction at any given moment. With that, perhaps the most significant difference between the SIMD and the SIMT architectures lies in the way the underlying process is coded by the user and perceived by the threads. The SIMD architecture takes a bird’s eye view during the parallelization process, since all instructions are carried out in an identical fashion – that is, when coding the algorithm processes under the SIMD architecture, one has to keep all threads in mind at the same time as a single computing entity. On the other hand, since SIMT allows the execution of different instructions carried out simultaneously by different warp batches, the SIMT architecture takes the view of a single thread – that is, rather than treating lines of code as a whole, every single line of the code should be strictly targeted to a single GPU thread.

2.4 Performance Gauging and Acceleration Techniques

2.4.1 Performance Gauging

To accelerate any algorithm on a GPU, one must keep in mind the GPU hardware architecture in order to map the algorithm processes onto the hierarchy of the GPU processing and memory resources for optimized GPU acceleration. Before getting into the acceleration techniques, it is worthwhile to first look at the performance gauging criteria for the many-core and multi-core parallel systems. In general, most of the algorithms can be categorized into two distinct groups: the processor-intensive algorithms and the memory-intensive algorithms. Due to the difference in their computation processes, it is most natural to gauge their parallelization performance based on different benchmarks. On one hand, processor-intensive algorithms carry heavy computations based on a limited amount of data (e.g., method of moments, Fourier transform, video encoding, etc.). Since the majority of the total computation time is on the computation, it is best to gauge the processor-intensive algorithms with the rate of the achievable arithmetic operation in the form of giga-floating-point operations per second (GFLOP/s). On the other hand, memory-intensive algorithms carry out light computations on a vast amount of data (e.g. finite-difference time-domain method, finite-element methods, Monte Carlo, etc.). Since the majority of the total computation time is on the reading and writing of the memory, it is best to gauge the memory-intensive algorithms with the memory access bandwidth in the form of giga-bytes per second (GB/s).

Despite the appropriateness of the two performance gauging benchmarks, the ultimate question of interest is how *well* GPUs can outperform CPUs in parallel computation, that is, how much *speedup* the addition of GPUs can bring on top of the existing computing resources. After all, speedup against CPU is the only practical indicator in real life. Nevertheless, the discussions of FLOPS and bandwidth still has its merits. For example, it is often difficult to tell from the speedup how "well" an algorithm is parallelized, since speedup is a relative number depending on multiple factors such as the benchmark CPU hardware platform and the inherent parallelizability of the entire algorithm. On the other hand, FLOPS and bandwidth tell us precisely how well each process in the algorithm is mapped onto the targeted GPU architecture, and how much of the various GPU processors and resources are utilized for the proposed

parallelization techniques compared to the maximum FLOPS and bandwidth of the specific GPU hardware.

2.4.2 Acceleration Techniques

According to the above discussion, a GPU usually specifies its capability with the maximum achievable FLOPS and bandwidth due to the specific hardware configuration, where these two gauging benchmarks are also used for the two categories of algorithms. Thus we target our acceleration techniques primarily at maximizing these two criteria.

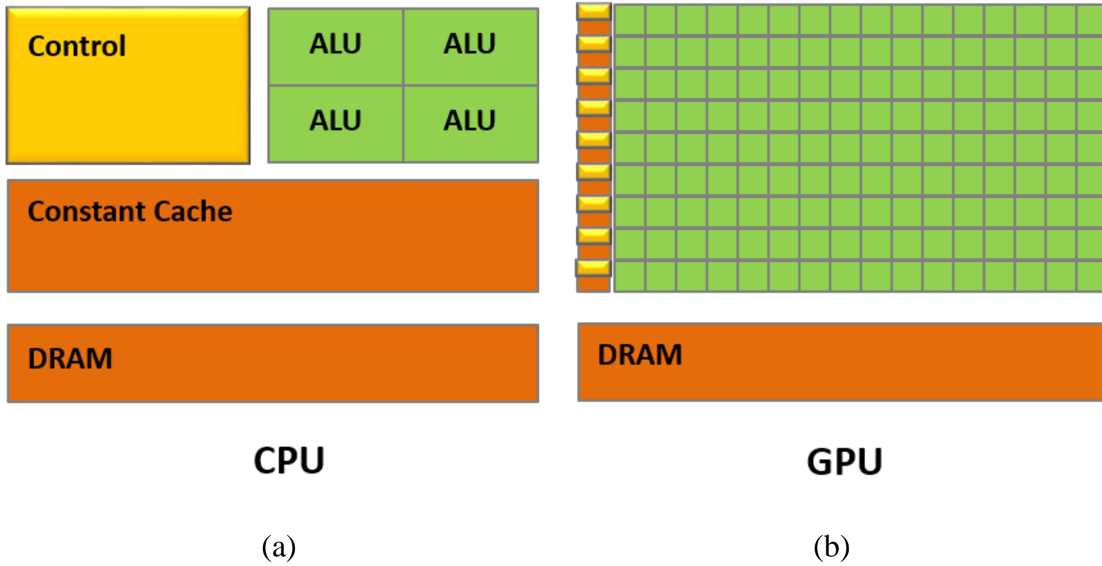
Warp-level parallelization and warp scheduler occupation are the foremost acceleration considerations for an algorithm to achieve high FLOPS. When a kernel spawns the blocks and threads for computation, the warp schedulers in each SM take on a few warps at a time and assign them a line of instruction at a time. While the first batch of warps goes off with its assignments, the next batch of warps is scheduled and sent off by the warp schedulers. It is not until any batch of warps finishes its assignment that the warp schedulers assign it with the next instruction. Since it is the assignment of instructions that carries out the actual computation, it is important to keep the warp schedulers busy with available warps. One way to achieve this is to spawn as many blocks and threads as possible for a given kernel, providing that the algorithm can be massively parallelized. Also, it is important to prevent warp divergence. For example, the execution of branching statements within a warp will be serialized across all branches, and thus needs to be avoided. Once the number of warps and the instructions are fixed, another way to keep the warp schedulers busy is to utilize the hierarchy of memories so that each warp is able to complete its assignment in less time.

Memory coalescing is a very important contributor to GPU acceleration for any algorithm to achieve high bandwidth. The high latency of the global memory access can be hidden with coalesced global memory access by using the concept of warps. When the 32 threads of a warp access a consecutive segment from the global memory, a consecutive 128 bytes of device global memory are accessed within a single instruction, effectively speeding up the memory access by 32 times. However, if the underlying algorithm requires random memory access, each warp will still access a consecutive memory segment in a single transaction, regardless of the amount of the actual data needed in the segment. The worst scenario is to serialize all 32 memory accesses, which can significantly reduce the overall acceleration. Fortunately, the Fermi architecture with

Compute Capability 2.0 and beyond introduces L1 and L2 on-chip caches, much like the ones found in the CPU architecture. Each SM carries its own small L1 cache (maximum 48 KB), where the same L2 cache is shared across all SMs (maximum 768 KB). These caches temporarily store the fetched memory segments, allowing them to be used if any of them are accessed in the near future, although they are smaller than those found on a CPU (64 KB L1, 256 KB L2, and 8MB L3 for Xeon X5560). As will be shown in Section 4.3.3, the incorporation of the L1 and L2 caches is able to alleviate the latency caused by random memory access.

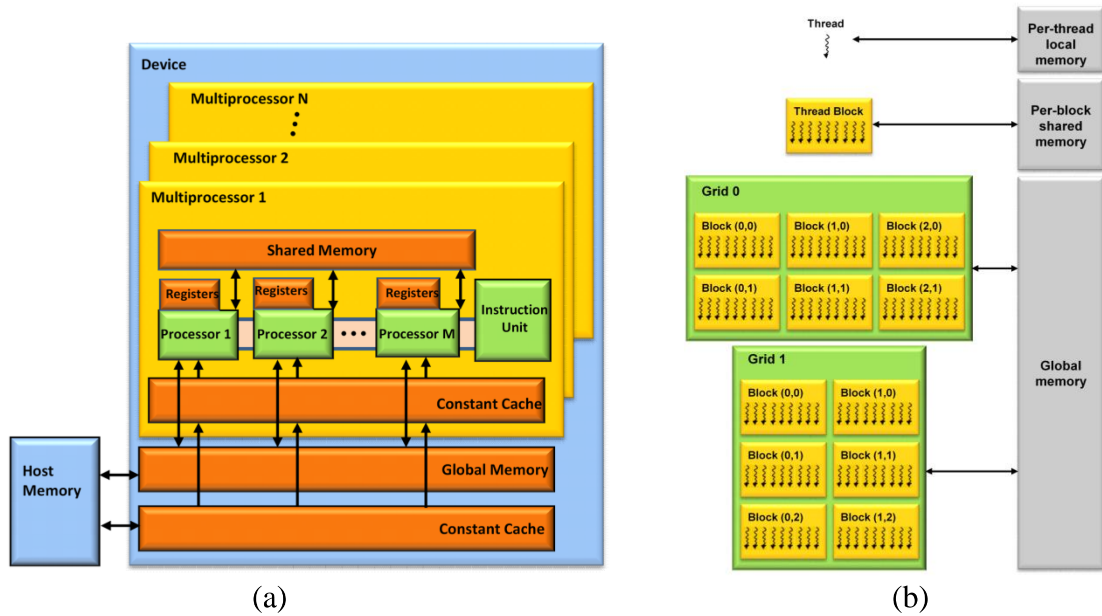
Other than the two criteria mentioned above, there also exist other ways to further accelerate an algorithm process. For example, on newer-generation GPUs (post-Compute Capability 2.0 for NVIDIA GPUs), concurrent execution of kernel launches and host-device memory transfer can be performed [71], where one can hide the delay from memory transfers between the host and device by splitting a single large kernel to several smaller kernels, and perform memory transfer on the portion of the memory which has been processed by the previous kernel concurrently as one launches the next kernel for the portion of the unprocessed memory. Another subtler acceleration technique is to avoid *bank conflict* when accessing the shared memory and the constant cache, in which the memories are divided into several *banks* accessible by a single memory request at a time. This acceleration technique requires thorough understanding of the specific GPU architecture and model, and is not covered in this dissertation.

2.5 Figures



(a) (b)

Fig. 2.1: (a) CPU relevant component size. (b) GPU relevant component size.



(a) (b)

Fig. 2.2: NVIDIA (a) GPU hardware architecture. (b) CUDA software API architecture.

CHAPTER 3
GPU ACCELERATION OF THE FINITE ELEMENT METHOD

3.1 Introduction

In this chapter, the formulation of the fundamental finite-element algorithms will be investigated and accelerated by the GPUs under OpenMP-CUDA environment, with the goal of near-complete parallelization of the finite-element algorithm. This study sheds light on the nature of each of the finite-element processes in terms of the parallelization efficiency and bottlenecks, which will be useful for the parallelization of more advanced finite-element algorithms, such as the various domain-decomposition algorithms. Section 3.2 presents the finite element algorithms for electromagnetic problems in the time domain and frequency domain. Section 3.3 discusses the finite element assembly stage on the traditional CPU and proposes new schemes for GPU parallelization. Section 3.4 discusses the implementation of the finite element matrix solution stage on single- and multi-GPUs, which includes iterative solvers and preconditioners. Section 3.5 presents various examples of electromagnetic analysis and the resulting GPU speedups for each case. The GPU hardware used in this chapter is, unless otherwise specified, NVIDIA's Tesla C2050 of Compute Capability 2.0, with 3GB of memory. For the CUDA software, this chapter uses CUDA v.4.2.

3.2 Finite Element Formulation

To formulate the fundamental finite-element algorithms in the time- and frequency-domain, consider an electromagnetics problem defined in a volume denoted by Ω and bounded by a perfect conducting surface, where the volume consists of a medium that is characterized by permittivity ε, permeability μ, and conductivity σ, and it contains an excitation electric current denoted by $\bar{J}_{\text{imp}}$. The finite element analysis of this problem yields the system of equations

$$[S]\{u\}+[B]\frac{d\{u\}}{dt}+[M]\frac{d^2\{u\}}{dt^2}=\frac{d\{f\}}{dt} \tag{3.1}$$

in the time domain, and

$$\left([S]+j\omega[B]-\omega^2[M]\right)\{u\}=j\omega\{f\} \tag{3.2}$$

in the frequency domain. In these two equations, the matrix and vector elements are given by

$$S_{IJ} = \iiint_{\Omega} \frac{1}{\mu} (\nabla \times \vec{N}_I) \cdot (\nabla \times \vec{N}_J) \, dV \tag{3.3}$$

$$B_{IJ} = \iiint_{\Omega} \sigma \vec{N}_I \cdot \vec{N}_J dV \tag{3.4}$$

$$M_{IJ} = \iiint_{\Omega} \varepsilon \vec{N}_I \cdot \vec{N}_J dV \tag{3.5}$$

$$f_I = -\iiint_{\Omega} \vec{N}_I \cdot \vec{J}_{\text{imp}} dV \tag{3.6}$$

and $\{u\}$ is the unknown vector that contains the expansion coefficients, $\vec{N}_I$ and $\vec{N}_J$ $(I, J = 1,\ 2,\ ...,\ N)$ represent interpolatory vector basis functions [23] for the expansion of the electric field, where N denotes the total number of such basis functions.

For finite element implementation, the global matrices and excitation vector are assembled based on the elemental matrices and vectors as

$$S_{IJ} = \sum_{e=1}^{P} \iiint_{\Omega_e} \frac{1}{\mu^{(e)}} (\nabla \times \vec{N}_i^{(e)}) \cdot (\nabla \times \vec{N}_j^{(e)}) \, dV \tag{3.7}$$

$$B_{IJ} = \sum_{e=1}^{P} \iiint_{\Omega_e} \sigma^{(e)} \vec{N}_i^{(e)} \cdot \vec{N}_j^{(e)} dV \tag{3.8}$$

$$M_{IJ} = \sum_{e=1}^{P} \iiint_{\Omega_e} \varepsilon^{(e)} \vec{N}_i^{(e)} \cdot \vec{N}_j^{(e)} dV \tag{3.9}$$

$$f_I = \sum_{e=1}^{P} \iiint_{\Omega_e} \vec{N}_i^{(e)} \cdot \vec{J}_{\text{imp}}^{(e)} dV \tag{3.10}$$

where e denotes the element number, Ω_e denotes the volume of element e, P denotes the total number of elements, and (i, j) denote the local indices, which are related to the global indices (I, J) by $I = n(i,e)$ and $J = n(j,e)$, with $n(i,e)$ and $n(j,e)$ denoting the connectivity array.

Equation (3.1) can be further discretized in the time domain using the Newmark-Beta method to obtain a time-marching equation that can be used to calculate $\{u\}$ in each time step. The Newmark-Beta method is unconditionally stable and has a second-order accuracy, where

$$\left. \frac{d^2 y}{dt^2} \right|_{t=n\Delta t} = \frac{y^{n+1} - 2y^n + y^{n-1}}{(\Delta t)^2} \tag{3.11}$$

$$\left.\frac{dy}{dt}\right|_{t=n\Delta t} = \frac{y^{n+1} - y^{n-1}}{2\Delta t} \tag{3.12}$$

$$y\big|_{t=n\Delta t} = \frac{1}{4} y^{n+1} + \frac{1}{2} y^{n} + \frac{1}{4} y^{n-1} \tag{3.13}$$

For time-harmonic fields, Equation (3.2) can be solved directly in the frequency domain for the unknown vector $\{u\}$. Nonetheless, in either a time-domain or a frequency-domain solution, we have to solve a matrix equation using a direct or an iterative solver. This matrix is a combination of $[S]$, $[B]$, and $[M]$, which is often very large but highly sparse, where only a few nonzero matrix elements are present in each row and column.

3.3 GPU Implementation — Assembly

The GPU implementation of the finite element assembly is discussed in detail in this section, as well as all of the data structures involved in the parallelized assembly process. The traditional assembly process on CPU is not suitable for GPU implementation anymore, and modification of the data structures is needed. This section lays out the traditional finite element structures, points out the difficulty in their direct application to GPU, and proposes the necessary modifications. For simplicity, the lowest-order finite elements are used for implementation and testing. However, it is expected that even greater acceleration can be achieved with a GPU with higher-order finite elements, as higher-order finite elements require more calculations.

3.3.1 The Traditional CPU Approach

A traditional finite element algorithm on CPU begins by reading the mesh file into a data structure, and then deduces the relevant intermediate data from the geometry to prepare for the construction of the elemental matrices. Traditionally, the CPU loops over each finite element and boundary element, computes their corresponding elemental matrices, then assembles them into the global system matrix one by one. This serial approach cannot be readily parallelized using GPU due to the following bottlenecks.

The first bottleneck is related to the calculation of the elemental matrices. This calculation requires that intermediate data be deduced from the geometry, which is often stored in the form of array of structures. These data structures hold the relevant information of each

element in individual memory allocations, which are then combined together to form an array. These data structures are burdensome to transfer between CPU and GPU. Because the transfer rate between CPU and GPU memories is low, this poses a bottleneck for any meaningful acceleration if bulky data structures are to be transferred. As GPU global memory is usually much smaller than CPU memory, it is impractical to transfer all of the data structures from CPU to GPU. Furthermore, bulky memory classes by nature hinder GPU parallelization. Aside from many-core parallelization, one of the important acceleration advantages of a GPU is the coalesced memory access pattern. If data are accessed from a single array with large structures, each data access will not be consecutive and thus will not be coalesced, reducing the overall GPU efficiency.

The second bottleneck is related to the assembly of the elemental matrices into the global matrix. The parallel assembly of the elemental matrices involves potential race conditions. Since most of the non-zeros in the global matrix are contributed by more than one element, the contributions cannot be assembled simultaneously into the same memory location due to memory access conflicts. This pervasive race condition can dramatically reduce the achievable GPU acceleration.

There are several parallelization strategies that can potentially alleviate these bottlenecks. The first scheme is to parallelize the computation of the elemental matrices on a single GPU, with one GPU thread calculating one element, and then using atomic operations to assemble the elements according to the predetermined hash indices, which map each of the computed elemental matrix elements to their corresponding locations in the final system matrix. The second scheme still parallelizes the computation of the elemental matrices, but then assembles the final system matrix by degrees of freedom (DOF) instead of by elements. The third scheme is designed for a multi-GPU environment, where the computation of the interactions between a DOF with its adjacent DOFs is parallelized by row. This effectively assigns each GPU to handle a number of DOFs, and one GPU thread calculating a single DOF, where the construction and the assembly of partial elemental matrices related to a single DOF is performed. The aforementioned approaches are discussed below in more detail.

3.3.2 Parallelization Scheme 1 — One Element per Thread on a Single GPU with Atomic Assembly

The single GPU parallelization scheme is chosen as the most basic approach to bypass the two bottlenecks discussed in Section 3.3.1. First, since host/device data transfer needs to be kept to a minimum, only essential geometry information should be transferred from CPU to GPU, while all other intermediate data are computed on the GPU. In place of the traditional bulky classes, small data structures are formed to store the geometric nodes and elements in separate arrays to ensure GPU memory coalescing. Figure 3.1 shows the comparison between the traditional element structures on the CPU and the arrays of small GPU structures which are beneficial for GPU acceleration. The resulting size of the transferred geometry information takes only about 10% of the memory usage of the traditional CPU geometry structure, thereby greatly reducing the memory transfer time.

The atomic assembly of the elemental matrices is straightforward. The CPU first loops over all the elements and determines the hash indices of each element of the elemental matrices in the final compressed global system matrix format, such as the popular compressed sparse row (CSR) format. The hash table is then transferred to the GPU, where the GPU threads use the indices to assemble the calculated elemental matrices. However, this assembly process involves numerous atomic operations of two or more GPU threads writing to the same global matrix location, where the GPU hardware queues the threads from their memory access, effectively slowing down the GPU acceleration. Furthermore, the pre-determination of the hash indices on the CPU is not only time-consuming but also bulky, which reduces the overall GPU acceleration. As a remedy, the assembly-by-DOF solution is proposed next.

3.3.3 Parallelization Scheme 2 — One Element per Thread on Single GPU with Assembly-by-DOF

The assembly-by-DOF approach eliminates the needs for building the hash table on the CPU and the atomic operations on the GPU. Notice that each row of the global system matrix consists of interactions between a DOF with all of its adjacent DOFs. One apparent parallelization scheme from this viewpoint is then to parallelize the assembly by DOFs. Specifically, each thread handles all of the DOF-to-DOFs coupling from all adjacent elements,

which consists of all the interactions between the target DOF and all of its neighboring DOFs.

For the assembly of elemental matrices with lowest-order finite elements, this reduces to the interaction between each edge and its adjacent edges. To proceed with assembly by edges, we first construct an edge-to-element table in the edge class during the edge construction phase. The layout of the edge class is shown in Fig. 3.2. A pre-processing GPU kernel first determines the number of non-zero elements in each system matrix row in parallel. This process is shown in Fig. 3.3, where the prefix sum is accumulated on the CPU to form the index array for the CSR sparse matrix format. As an alternative, the accumulation process can be carried out with parallel reduction on the GPU. The assembly kernel is then invoked, and each GPU thread loops over an edge's adjacent elements to obtain the adjacent edge indices and their corresponding non-zero values. Each of the resulting global matrix rows is first assembled in the shared memory for fast memory access, and then stored in the global memory to form the final CSR matrix according to the pre-determined sparsity pattern.

The flow of the finite element program on a GPU is then as follows: The mesh information is first stored in arrays of geometric nodes, connectivity, and boundary conditions, etc. The CPU will still be used to form the edge and element structures due to the serial nature of the process. After all the relevant data are transferred onto the device global memory, different GPU kernels are invoked in sequence to compute the sparsity pattern, tetrahedral elemental matrices, and triangular elemental matrices, and store them in the global memory, using one GPU thread per element. Notice that small global constants such as frequencies and quadrature integration points and weights can be stored in the device's global constant memory for cached access. The assembly kernel is then invoked to parallelize the assembly by rows using one GPU thread per DOF. Each assembled row is stored temporarily in the shared memory before exporting to the global memory. At this point the assembly of the system matrix and the right-hand-side is completed. Figure 3.4 illustrates the program flow. Note that the formation of the edge structures introduces a small but unavoidable overhead which is not found in the traditional process on the CPU, but this scheme overall is capable of accelerating the entire finite-element assembly process. It is worth noting that when higher order basis functions are used, this overhead becomes insignificant as the amount of matrix computation increases.

The assembly-by-DOF method is as straightforward as the first method, but it eliminates delays caused by the construction of the hash indices and the atomic operations. However, this

method poses a potential problem due to the memory limitation of the GPU. The number of elements increases with problem size, and the dimension of the elemental matrices increases with basis function order. Both scenarios will overload the GPU memory with the storage of the elemental matrices. To alleviate the memory constraint, a multi-GPU scheme is proposed next.

3.3.4 Parallelization Scheme 3 — One DOF per Thread on Multi-GPUs

For multi-GPU parallelization, a slightly different scheme from that of the single GPU is used. Due to the memory constraint of the GPU, a single GPU is not capable of handling very large problems with the number of unknowns ranging in the millions. The multi-GPU approach aims to provide this capability by distributing both the assembly and the solution phases of the finite element analysis across multiple GPUs.

To distribute the assembly of the global system matrix across all GPUs, it is natural to split the system matrix horizontally into chunks, with each constructed and residing independently on a single GPU. This distribution ensures the ease of the matrix operations in the solution phase, which involves numerous matrix-vector multiplications. During the assembly stage, the rows of a single elemental matrix will be assembled into different system matrix rows, which might be residing on different GPUs. To achieve distributed assembly across multiple GPUs, each elemental matrix row will have to be computed directly by its relevant system matrix row instead of parallelizing the construction of the elemental matrices and then assembling the matrices by rows. With the independent assembly of each system matrix row, the multi-GPU scheme is also ideal for elements using higher order basis functions. Using the same set of geometric data, a single thread is able to compute more non-zero entries per row in parallel. This natural parallelization of higher order basis will further speed up the assembly process when higher order basis functions are used.

The multi-GPU scheme is implemented as follows. Each GPU thread is responsible for the construction of a single system matrix row, which is equivalent to the interaction between the target DOF and its adjacent DOFs in the neighboring elements. Here each GPU thread serially loops through the neighboring elements and computes the relevant partial elemental matrices with dimension of $1 \times k$, where k is the degrees of freedom per element. The threads then assemble the partial elemental matrices into their corresponding destination in the system matrix row. With this multi-GPU approach, the construction of the system matrix is row-independent

and will not produce any memory conflict between the GPU threads. The process is illustrated in Fig. 3.5.

3.4 GPU Implementation — Matrix Solution

This section discusses the solution process of the finite element matrix equation and GPU parallelization. The finite element matrix equation can be solved by either a direct or an iterative solver. It is well known that the solution complexity and memory requirements scale exponentially with the dimension of the matrix equation for a direct solver. Therefore most large-scale finite element analyses employ an iterative solver in the solution phase, where easily-parallelizable matrix-vector multiplications dominate the solution process instead of intrinsically-serial forward-backward substitutions. Here we consider the biconjugate gradient stabilized (BiCGStab) method due to its simplicity and its efficiency compared to other iterative solvers, but the parallelization scheme can be easily applied to other iterative solvers since they are also dominated by matrix-vector multiplications.

3.4.1 Iterative Solver on Single GPU

The BiCGStab requires two sparse matrix-vector (SpMV) multiplications and several vector-vector calculations within each step in its iterative process. Among these calculations, the two SpMVs take up the majority of the elapsed computation time. In order to accelerate the solution phase using a GPU, we first need to investigate efficient SpMV GPU algorithms. It is discovered that the efficiency of the SpMV depends on the GPU threads' memory access pattern, which is determined by the sparse matrix storage format. Although the CSR format is the commonly used sparse matrix format due to its straightforward implementation, it has intrinsic load-balancing disadvantages if the number of non-zeros in each matrix row is highly uneven. As the GPU threads multiply each matrix row with the multiplier vector in parallel, this reduces the overall efficiency of the parallelization since many threads would be idle as they wait for other threads during the process.

The JAD format avoids this problem by rearranging the rows and columns to balance the load. It is reported in [72]-[74] that the ELLPACK storage format outperforms the CSR format on GPUs, but the Jagged Diagonal (JAD, or JDS in some literature) storage format outperforms both. Therefore, we first investigate the JAD format for the finite element computation, and

compare the resulting speedup to that of the CSR format, which is shown in Fig. 3.6(a). The JAD sparse matrix storage format is shown in Fig. 3.6(b). While CSR compresses the nonzero elements in each row and concatenates them into a continuous array, JAD takes a further step to first permute the rows according to the number of nonzero elements, then concatenate the resulting compressed and permuted rows in a column-major order. During a parallelized JAD SpMV, the GPU threads operate on the entire span of the matrix for each row-permuted column. The permuted column operations requires progressively fewer parallel threads as the number of nonzeros per permuted column decreases. The permutation ensures that each thread accesses the same row from the matrix, where the results can be temporarily accumulated in the on-chip cached memories for quick access.

For the SpMV multiplication on the device, it was observed that the JAD format provides a more coalesced memory access by the device threads, and thus reduces the SpMV time for the finite element matrices. The speedup of the JAD's SpMV is compared to that of the CSR format using the standard CuSparse library from NVIDIA CUDA Toolkit, which is implemented with a very efficient CSR SpMV algorithm. Table 3.1 shows the SpMV speedup on GPU for a finite element matrix (from scattering analysis of a sphere in the frequency domain) with different dimensions using Quadro FX5800 for GPU computations.

The results in Table 3.1 show that the JAD format outperforms the CSR format for the finite element matrix, although the speedup is more apparent for smaller matrices. It is difficult to improve this speedup further, since the JAD format intrinsically loses its formatting efficiency as the number of non-zero elements in the matrix increases. Since in general the finite element matrices are not terribly unbalanced, the speedup obtained is not as effective as other matrices discussed in the literature, such as the one resulting from the hybrid finite element-boundary integral method. In addition, it is not easy to directly interpret the JAD matrix for constructing preconditioners, which is critical for time reduction in an iterative solver. Due to the above reasons, CuSparse using CSR is still preferable over the JAD format for SpMV on GPU for practical purposes. For vector-vector calculations on GPUs, NVIDIA's CuBLAS library is highly efficient and can be used directly.

With the use of CuBLAS and CuSparse library functions, the implementation of the BiCGStab on GPU is then straightforward, and bears similarities with [35]. However, to obtain an efficient iterative solution, we have to employ an effective preconditioner to decrease the

number of iterations, and below we discuss the construction of such a preconditioner on GPU.

3.4.2 Parallelized Preconditioners

For the most efficient GPU implementation there are two criteria for the selection of a suitable preconditioner. First, its construction has to be parallelizable in order to be best performed on a GPU. Second, its application in the iterative solver has to be parallelized in order to achieve a good speedup. As an example, the commonly used incomplete LU (ILU) preconditioner is not suitable for the GPU because both of its construction and application processes are inherently serial. The application of the ILU preconditioner in the iterative solver involves backward and forward substitutions, which contain very few parallelizable processes despite their availability in the CuSparse library. It is worth noting that GPU backward and forward substitutions for large dense matrices (*csrmv_solve* from the CuSparse library) and full matrices (*trsm* from the CuBLAS library), such as those from the moment methods, still provide reasonable speedup compared to CPU. However, since finite-element matrices are extremely sparse, the backward and forward substitutions on GPU will end up being slower than that on the CPU and thus it is not desirable to apply them. Among the two criteria, the parallel application of the preconditioner is most important since it occupies the majority of the solution time, where it is not absolutely necessary for the construction of the preconditioner to be parallelizable.

To obtain good parallelization, the most obvious choice is the Jacobi (diagonal scaling) [40] preconditioner, in which the construction and application processes are both inherently parallel and very cheap. However, the Jacobi preconditioner does not provide good enough preconditioning. Sometimes, it does not even help with achieving convergence at all, especially in the analysis of complicated antennas and microwave devices. A better choice is the Symmetric Successive Over-Relaxation (SSOR) [40] preconditioner due to its parallelizable and cheap construction. However, just like the ILU preconditioner, the serial nature of SSOR's application in the iterative solver limits its parallel efficiency. This drawback is shown in Table 3.2 for a 1m-sphere scattering simulation with 176,522 unknowns at 300MHz, where it can be seen that although the SSOR preconditioner successfully reduced the number of iteration steps by a factor of four compared to the Jacobi preconditioner, the actual speedup achieved with the SSOR preconditioner on the GPU was actually less than the simple Jacobi preconditioner. A quad-core Xeon W3520 and Quadro FX5800 are used as the computing CPU and GPU,

respectively.

Through further investigation, it was found that the family of approximate inverse (AI) preconditioners matches our criteria. These preconditioners use mathematical approximations to invert the preconditioning matrix; therefore, their incorporation in the iterative solver involves only additional SpMV multiplications instead of the time-consuming backward and forward substitutions. The SSOR-AI preconditioner [39] can be constructed by expanding the inverse of the SSOR preconditioner with a Neumann series, and then truncating the series by taking its first-order approximation. The construction process of the SSOR-AI is also inherently parallel. Moreover, with the incorporation of diagonal scaling before applying the SSOR-AI preconditioner, the number of iterations can be further reduced. The total number of iteration steps in terms of the matrix condition number can be controlled through a careful selection of the SSOR relaxation parameter ω. This relaxation parameter ω ranges from 0 to 2 and its optimal value corresponds to the point with a minimum number of iterations. Figure 3.7 shows the number of iterations versus the relaxation parameter ω in the BiCGStab solution using the SSOR-AI preconditioner for the finite element analysis of scattering by a perfect electrically conducting (PEC) cube having a side length of 1m using 537,066 unknowns for 300MHz and 1,337,400 unknowns for 500MHz. The figure shows ω in the range $0.1 < \omega < 0.6$. Other values are either way off from the valley or do not produce a converged solution. It can be seen that the valley is somewhere around $\omega = 0.4$, and that it is invariant with the size of the problem. Similar sweeping was done with a monopole and a Vivaldi antenna for the antenna analysis, and the valley was again found to be around $\omega = 0.4$. Since it is difficult to numerically estimate ω, the experimental result of $\omega = 0.42$ will be used in the examples in the next section.

The performance of the SSOR-AI preconditioner is shown in Table 3.2. Due to the approximation introduced through AI, the number of iterations is increased and the total CPU computation time is also increased. The GPU computation time is decreased significantly for the reasons explained earlier. It is worth noting again that although the capability for preconditioner parallel construction is desired, it is not absolutely necessary since the preconditioner application occupies the majority of the solution time. Therefore, using another AI preconditioner with serialized construction on GPU may be as effective as the SSOR-AI.

Thus far in this section, we have considered the acceleration of an iterative solver along with the proper preconditioner for a single GPU computation. Next, we will discuss multi-GPU

applications for the iterative solution of the system matrix.

3.4.3 Iterative Solver on Multi-GPUs

For a multi-GPU implementation, the most natural parallelization scheme in the solution phase is to split the system matrix horizontally into chunks, and then compute partial matrix-vector products in parallel on the GPUs. The partial product vectors then have to be gathered to form a complete product vector, which in turn needs to be redistributed onto each GPU for the following SpMV or vector-vector calculations in the same iteration. This approach requires CPU-GPU memory transfers of partial and complete product vectors in each matrix-vector multiplication, as shown in Fig. 3.8. The family of AI preconditioners can also be applied in a similar manner to reduce the number of iteration steps, which is not possible with the conventional serially-applied preconditioners. It will be shown later in Section 3.5.4 that the latency of memory transfer plus inter-GPU synchronization poses the biggest bottleneck for multi-GPU computation. However, without using any advanced domain decomposition strategy, these latencies are unavoidable in the multi-GPU implementation. One such domain decomposition strategy can be found in [23, Section 10.2.2, pp. 371-376] and has been adopted for multi-GPU computation with excellent performance [75].

3.5 Examples and Speedups

To test the schemes proposed for the acceleration of the finite element computation on GPU, various electromagnetics problems are considered below. The GPU device used is the NVIDIA Tesla C2050, and for comparison, Xeon W3520 quad-core CPUs are used. The SpMV and the BLAS libraries in the Intel Math Kernel Library (MKL, v10.3 at the time of the work) with OpenMP capability are used for SpMV and vector-vector calculations on the CPU.

3.5.1 Single-GPU Solution + Serial Preconditioner

The first group of examples utilizes the simplest GPU scheme to simulate electromagnetic compatibility (EMC) problems using the finite-element time-domain (FETD) method. The assembly phase is kept on the CPU, where the ILU preconditioner (ILUT, $\text{max-fill} = 20$, $\text{tolerance} = 10^{-2}$) of the final finite element matrix is constructed using the MKL

library on the CPU. The system matrix and the preconditioner are then transferred to the GPU for the solution phase using the BiCGStab iterative solver, and the CuSparse library functions are used for the backward and forward substitutions of the ILU preconditioner. The timing of this simple GPU scheme is compared to that of the MKL direct sparse solver (Pardiso) on an OpenMP enabled multi-core CPU because this CPU solution is the most efficient solution we have tested so far for problems of this size.

Problem #1: Electromagnetic compatibility problem with a shielding enclosure.

For validation purposes, the first example is an EMC shielding problem with a rectangular PEC enclosure. The enclosure has a dimension of $0.22\text{m}\times 0.3\text{m}\times 0.14\text{m}$, as shown in Fig. 3.9(a), with and without a slot. A coaxial cable feeds the excitation into the enclosure, and is terminated by a 47Ω resistor connected to the enclosure. The finite element discretization yielded 609,451 unknowns for the case without the slot and 627,213 unknowns for the case with the slot. The transient response is computed in the time domain, and then transformed into the frequency response using the Fourier transform. The resulting delivered power into the box without the slot is shown in Fig. 3.9(b), and that with the slot is shown in Fig. 3.9(c). In both cases, the simulated results agree well with the measured data [76], thus validating our GPU-based finite element analysis.

The timing comparison of the ILU preconditioned iterative solver on a GPU against the parallelized MKL direct sparse solver on a CPU is given in Table 3.3 for the case without the slot and in Table 3.4 for the case with the slot. In both cases, the solution is marched over 10,000 time steps, and the residual tolerance in the BiCGStab solution is set to $\varepsilon = 10^{-6}$. A careful examination of the data reveals two points. First, since the backward and forward substitutions consume most computation time in the entire solution and are difficult to parallelize, the speedup using more CPU cores is not significant and in both cases considered here it is only about 2.5 times when the number of CPU cores is increased from 1 to 12. Second, since the finite element system for the case with the slot is better conditioned (because of radiation through the slot), it requires fewer iterations in the BiCGStab solution. This results in a better speedup for the GPU-BiCGStab over the CPU-MKL. Since the problem is relatively small and the OpenMP-enabled Pardiso is very efficient on CPU, the overall speedup is relatively low. As will be seen later, the speedup can increase significantly for larger problems.

Problem #2: Electromagnetic compatibility problem of a shielding enclosure with apertures.

The second example is an EMC shielding problem of an enclosure with rectangular apertures [77]. The enclosure has a dimension of 0.5m×0.4m×0.2m, with an aperture array on one side, as shown in Fig. 3.10(a). A coaxial cable feeds the excitation into the enclosure, and is terminated by a 47Ω resistor connected to the enclosure. The problem is modeled with 873,005 unknowns for the simulation from 0.3 to 1.2 GHz and 1,674,458 unknowns for the simulation from 1.2 to 2.1 GHz. Both simulations are carried out for 40,000 time steps, and the resulting delivered power is shown in Fig. 3.10(b) and Fig. 3.10(c). Table 3.5 and Table 3.6 give the timing comparison of the ILU preconditioned iterative GPU solver against the parallelized MKL direct sparse CPU solver for the two simulations. The results indicate again that computation with CPU multi-cores does not significantly reduce the computation time because of the serial nature of the direct solver. On the other hand, by using an iterative solver on a single GPU with the ILU preconditioner, we are able to achieve a significant speedup, which increases with the number of unknowns. In the case with 1,674,458 unknowns, a single GPU with the ILU preconditioner is 22 times faster than the direct sparse solver on a single-core CPU and 10 times faster than a parallelized direct sparse solver on a quad-core CPU.

3.5.2 Single-GPU with Atomic Assembly + Single-GPU Solution

The second group of examples utilizes a single GPU in both the assembly and the solution phases for the finite element analysis of radiation problems in the frequency domain. To demonstrate the GPU's capability, both the CPU and the GPU use the same BiCGStab iterative solver for the solution phase. Simple diagonal scaling is added in the solution phase to slightly reduce the number of iteration steps.

Problem #3: 2D horn antenna radiation.

The first example is a 2D horn antenna with a port boundary at the waveguide feed end. The horn opens into free space, which is truncated sufficiently far away with the first-order absorbing boundary condition. Table 3.7 and Table 3.8 show the timing comparison between a single and a quad-core CPU and a single GPU. Figure 3.11 shows the geometry of the horn and

the computation domain and the resulting field distribution at 300, 500, and 800MHz, respectively. The residual is set at $\varepsilon = 10^{-3}$ to terminate the BiCGStab iteration.

The results in Table 3.7 and Table 3.8 show a complete utilization of the GPU in both the assembly and the solution phases. For the assembly phase, the majority of the time taken is in the preprocessing step, which initializes the GPU as well as computing the hash indices on the CPU. It is shown that the GPU preprocessing time surpassed the entire CPU assembly time because the calculation of the elemental matrices using the lowest-order finite elements on CPU is relatively fast compared with the preprocessing step which includes the construction of the hash indices. The actual assembly time with GPU is 16+ times faster than the CPU, and that acceleration will further increase if higher-order finite elements, which require more computation per unknown, are used. Since the assembly phase is very short compared to the solution phase, the acceleration of the solution phase has a much greater effect on the overall GPU acceleration performance. For the complete finite element analysis, a single GPU in this parallelized scheme is able to speed up the solution by a factor of 13+ against a single-core CPU and 6+ against a quad-core CPU. The simulations are also performed using the parallelized MKL direct solver for comparison, and the total simulation time is 2s, 5s, and 15s for 300, 500, and 800MHz, respectively on a quad-core Xeon W3520 CPU. It is shown that the direct solver solution is significantly faster than the iterative solver due to the extremely sparse system matrix and the 2D nature of the problem. (As will be seen, for 3D problems the computation time is much longer even for a similar number of unknowns.)

Problem #4: Field profiling using a point source.

For an example of field profiling from antenna radiation, the field distribution from a point source in a room setting is computed at several frequencies. Figure 3.12 shows the resulting field distribution at 300, 500, and 1000MHz, and Table 3.9 shows the speedup obtained. The residual is set to $\varepsilon = 10^{-3}$ to terminate the BiCGStab iteration. Once again, the majority of the time taken in the assembly phase is for hash index construction and preprocessing. For the entire finite element analysis, the single GPU is able to speed up the solution by a factor of 6 against a quad-core CPU. The parallelized MKL direct sparse solver results in a total computation time of 1s, 4s, and 18s for 300, 500, and 1000MHz, respectively on a quad-core Xeon W3520 CPU, which is again significantly faster than the iterative solver (again because the

problem considered is 2D and the system matrix is extremely sparse).

3.5.3 Single-GPU Assembly-by-DOF + Solution with Parallel Preconditioner

Our third group of examples demonstrates the effectiveness of the assembly-by-DOF approach, as well as the SSOR-AI parallel preconditioners. As discussed in the previous section, the assembly-by-DOF approach eliminates hash index construction and atomic operations, which can otherwise reduce the GPU acceleration. Moreover, with matrix preconditioning, the GPU solution time can be further reduced.

Problem #5: 3D monopole antenna radiation pattern.

The first example is a single monopole placed on the top of an infinite conducting plate. The height of the monopole is $h = 1\text{m}$, the inner radius of the coaxial waveguide is $a = 0.1\text{m}$, and the outer radius is $b = 0.23\text{m}$. For simplicity, the relative permittivity of the waveguide is set to $\varepsilon_r = 1.0$. A first-order spherical absorbing boundary is set at an appropriate distance away from the monopole. Table 3.10 shows the effect of the preconditioners and the GPU speedup for the monopole simulation with 436,920 unknowns at 75MHz. In this case, the assembly time on the CPU is 1.93s, and the preprocessing time on the single GPU is 1.26s and the assembly time is 0.36s. The GPU achieves a total of 1.19 times speedup for the assembly phase. Notice that the GPU achieves a speedup in the assembly phase by using the assembly-by-DOF approach compared to the no-speedup case for the atomic assembly approach in Section 3.5.2. However, both approaches will achieve a higher positive speedup if higher-order finite elements are used. The timing comparison in Table 3.10 is against the computation on an OpenMP enabled quad-core CPU, and the residual is set at $\varepsilon = 10^{-3}$ to terminate the BiCGStab iteration.

As discussed in the previous section, the construction and application of commonly used preconditioners such as SSOR is not suitable for further GPU acceleration. Although they are efficient in reducing the number of iteration steps, the total computation time is actually longer. From Table 3.10, it can be seen that truly parallel preconditioners, such as the SSOR-AI, are able to further reduce the total computation time on the GPU. The total simulation time using the parallelized MKL direct sparse solver takes 191.05s (in contrast to a few seconds for the 2D case) on a quad-core Xeon W3520 CPU, which is faster than the iterative CPU solver but slower

than the iterative GPU solver. Notice that although the relative speedup remains essentially unchanged, the total computation time after applying the SSOR-AI preconditioner with diagonal scaling on the GPU is further reduced, with the resulting total GPU computation time able to reach more than 8 times speedup compared to the simple diagonal scaling on the CPU. This speedup is even more apparent in the next example, where the size of the problem is much larger and the geometry of the problem is much more complex.

Problem #6: 3D Vivaldi antenna radiation.

The second example is a single Vivaldi antenna shown in Fig. 3.13 with $h = 33.3\text{mm}$, $w = 34\text{mm}$, and $d = 1.27\text{mm}$. The radius of the hollow circle is $R = 2.5\text{mm}$, and the taper is an exponential function with $w(z) = 0.25e^{0.123z}$ mm. The coaxial feed has an inner and outer radius of 0.375 and 0.875mm, respectively, and a relative permittivity of $\varepsilon_{r,\text{coax}} = 1.0$. The dielectric substrate has a relative permittivity of $\varepsilon_{r,\text{dielectric}} = 6.0$. The resulting finite element discretization has 1,221,796 unknowns at 3GHz. The assembly time on the CPU is 5.55s. The preprocessing time on the single GPU is 3.56s and the assembly time is 0.96s, resulting in a speedup of 1.23 times for the assembly phase. The timing comparison between a single GPU and the OpenMP enabled quad-core CPU is shown in Table 3.11.

In this case, the application of the simple Jacobi preconditioner failed to obtain a converged solution; however, the application of the SSOR-AI preconditioners managed to converge. Furthermore, by pre-applying the Jacobi diagonal scaling, the SSOR-AI preconditioner achieved a further reduction in the number of iterations and total computation time. As a comparison, the parallelized MKL direct sparse solver on CPU requires a huge amount of memory and computation time to obtain a solution due to the size of the problem and the 3D nature of the problem, which is not practical anymore compared to the iterative solver. The GPU accelerated the total computation by approximately 6 times when compared to the CPU, and by more than 20 times compared to the BiCGStab solution using only the SSOR-AI preconditioner on the CPU.

3.5.4 Multi-GPU Assembly + Multi-GPU Solution + Parallel Preconditioner

Examples in the last group demonstrate multi-GPU acceleration. As discussed in the

previous section, the multi-GPU scheme speeds up the assembly process by decomposing the computation of the final finite element matrix into horizontal chunks, and distributing them to multiple GPUs with each GPU handling one horizontal chunk. The gathering and redistribution of the product vectors link the GPUs in the solution step. The resulting speedups of 3D radiation simulations are presented below.

Problem #7: 3×3 monopole antenna array.

This example consists of a 3×3 array of monopole antennas spaced 1m apart, with the monopole length $h = 1\text{m}$, its inner radius $a = 0.1\text{m}$, and its outer radius $b = 0.23\text{m}$. The center monopole is excited at the port. The computation was carried out at 300MHz, which resulted in 2,085,876 unknowns. Since the single GPU scheme first computes and stores the elemental matrices in parallel before the assembly process, it has insufficient memory to solve this problem. Instead, multi-GPUs are used for this simulation and the results are compared in Table 3.12 against the computation using an OpenMP-enabled quad-core CPU. The SSOR-AI preconditioner with diagonal scaling is used to reduce the number of iterations.

From Table 3.12, it is seen that the speedup of the multi-GPU computations for this example is not higher than the single-GPU scheme for the previous examples because of the CPU-GPU memory transfer and GPU-GPU synchronization. The additional times taken for the transfer and synchronization are documented in Table 3.13 and Table 3.14. Specifically, Table 3.13 gives the average 2-GPU timing details for a single matrix-vector multiplication and the resulting product vector gathering and redistribution, and Table 3.14 gives the same information for the 4-GPU case. These timing results show that the memory transfer and synchronization step dominate the total time for each iteration, where the bottleneck lies in the vector redistribution transfer. As can be seen in Table 3.13 and Table 3.14, with twice the number of GPUs, 4 GPUs cut the SpMV phase by half. The gathering timing for the 2 GPUs is longer than for the 4 GPUs due to the transfer of larger vector segments ($1/2$ of the original vector comparing to $1/4$). However, although the vector size for the 4-GPU case is only slightly larger than the 2-GPU case ($3/4$ of the original vector comparing to $1/2$) in the redistribution phase, 4-GPU takes twice as much time as the 2-GPU. This is due to the fact that the redistribution phase comes right after the OpenMP barrier, which synchronizes the OpenMP threads that control each of the GPUs. While all GPUs begin their redistribution phase together, the PCI

Express bus connecting the host and the GPUs is packed with memory requests from all 4 GPUs, and thus slows down the redistribution speed. On the other hand, the gathering for each GPU takes place immediately after its SpMV; therefore, the memory transfer is more spread out and does not cause a traffic jam.

Lastly, since the finite element matrix is very sparse, the time saved in SpMV cannot fully compensate for the delay due to the memory transfer and synchronization. However, for very large problems that cannot be solved using a single GPU due to the memory constraint, the multi-GPU scheme will have to be used and can still accelerate the finite element solution although not as significantly as on a single GPU.

Problem #8: Larger monopole antenna arrays.

To really push the capability of the multi-GPU scheme, larger arrays are simulated. With the same configuration as the above 3×3 monopole array, 5×5, 7×7, and 9×9 arrays are simulated with the maximum capable unknown size for 1-, 2-, and 4-GPUs, respectively. The computations were carried out at 300MHz, which resulted in 3,135,869, 5,242,996, and 7,755,542 unknowns, respectively. Notice that when the one-unknown-per-thread approach for the multi-GPU scheme is applied to a single GPU, the maximum solvable problem size has increased significantly. This is due to the fact that the one-unknown-per-thread approach does not require an extra storage space for the elemental matrices. The simulation results are compared in Table 3.15-3.17 against computation using an OpenMP enabled quad-core CPU. The SSOR-AI preconditioner with diagonal scaling is used to reduce the number of iterations.

A few observations can be made from the results in the tables. First, for a fixed number of GPUs, the speedup increases with an increasing problem size. Second, when low-order finite elements are used which results in extremely sparse matrices, data transfer time dominates the iteration step. Therefore using more GPUs does not show a meaningful further speedup, as shown in the previous example. Lastly, it is seen that the maximum solvable problem size does not increase linearly with an increasing number of GPUs. This is due to the fact that the intermediate vectors in the BiCGStab iteration will gradually use up more space as the problem size increases. For example, for the 3×3 monopole array with 3,135,869 unknowns on a single GPU, the total memory occupation ratio of the vectors to the matrices (including the preconditioning matrices) is roughly 1:7, but the total memory occupation ratio for the 9×9 array

with 7,755,542 unknowns on 4-GPUs jumps to 1:2. Since the intermediate vectors gradually occupy more space as the problem size grows, it is not possible to reach a linear capability growth with an increasing number of GPUs. A potential solution to this problem is to employ a domain decomposition method, which would break down the finite element matrix and the intermediate vectors and distribute them among multi-GPUs based on physical subdomains.

In summary, with the multi-GPU algorithm presented here, it is preferable to use a minimum number of GPUs to solve a problem. For example, if a problem can be handled by one GPU, it is not advisable to use more GPUs because using multi-GPUs does not result in a further speedup due to the time-consuming data transfer. However, this does not imply a limitation on the size of problem that can be handled. In fact, with multi-GPUs one can deal with larger problems than that can be handled by a single GPU, and the maximum solvable problem size increases with the number of GPUs, although the increase is not in a linear fashion — a challenge that can potentially be alleviated by domain decomposition.

3.6 Figures and Tables

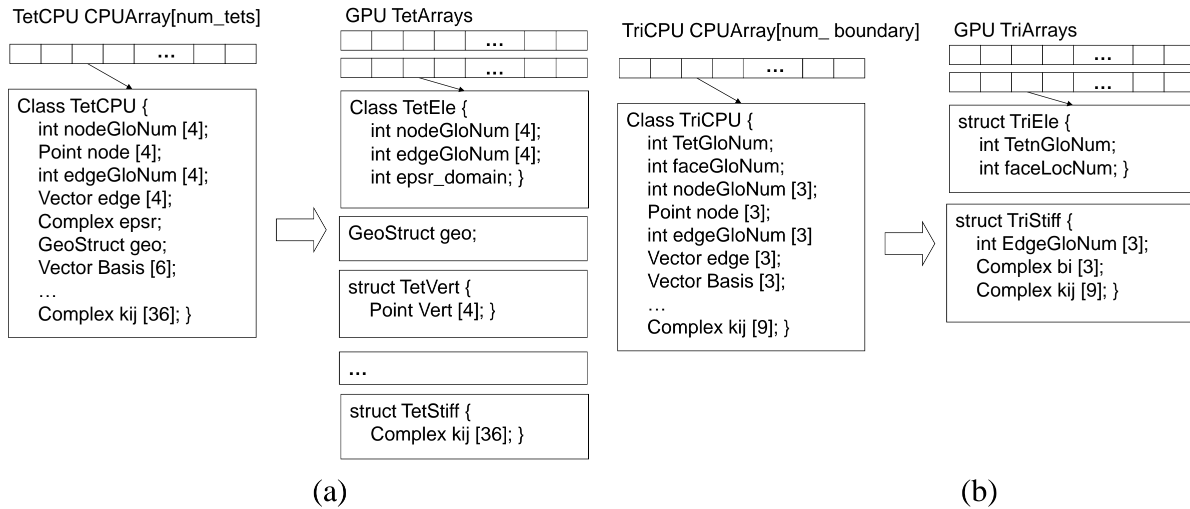


Fig. 3.1: (a) CPU tetrahedral element structure and GPU tetrahedral structure arrays. (b) CPU triangular element structure and GPU triangular structure arrays.

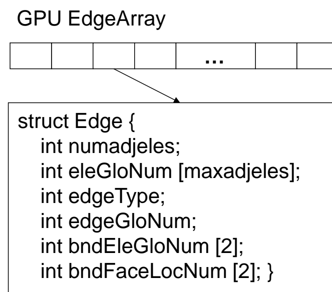


Fig. 3.2: Creation of the GPU edge classes.

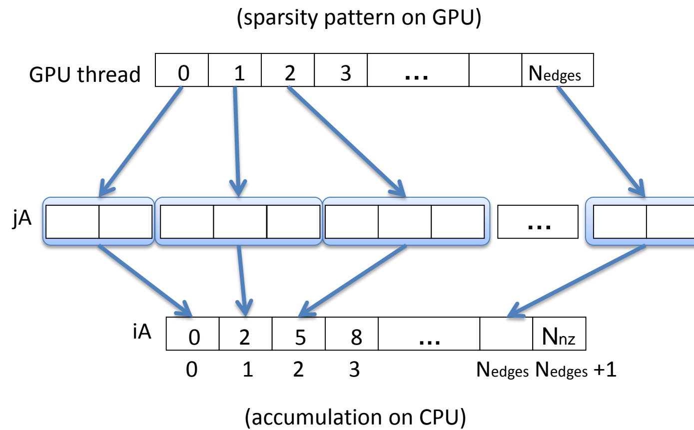


Fig. 3.3: Non-zero pattern assignments on a GPU, where each GPU thread will process a row and compute its number of non-zeros in the column array jA. The CPU then accumulates the number of non-zeros to obtain the sparsity pattern iA.

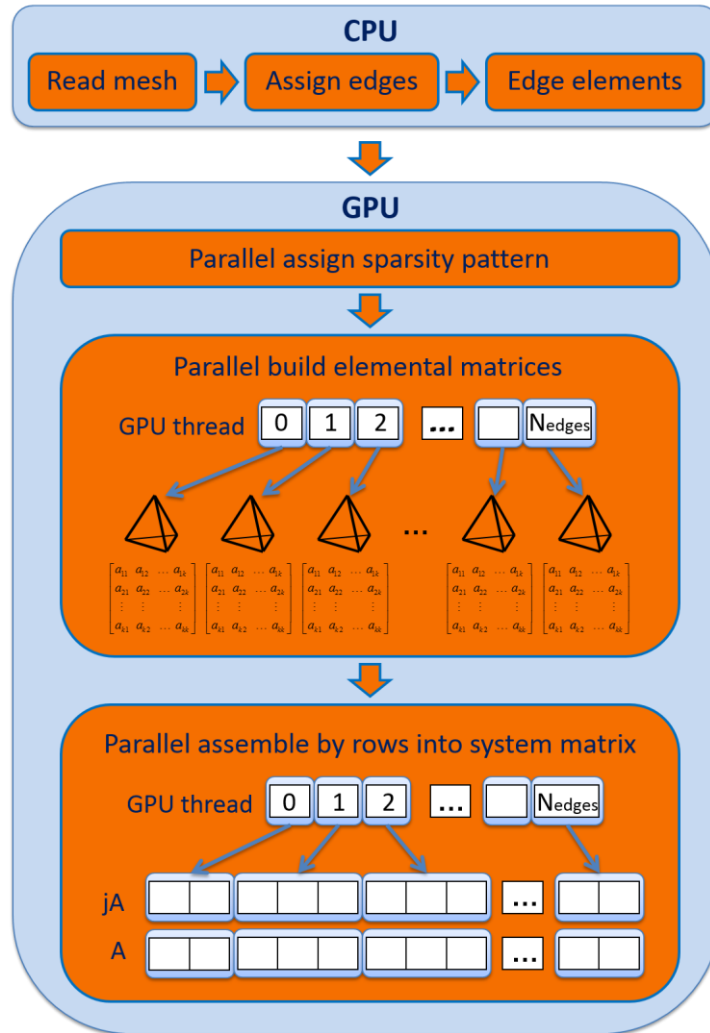


Fig. 3.4: Flow chart for the finite element assembly on a single GPU. After the assignment of sparsity pattern as in Fig. 3.3, a new GPU kernel is invoked with each thread computing one elemental matrix. Another GPU kernel will then be invoked with each thread assembling one row, fetching information from the relevant elemental matrices.

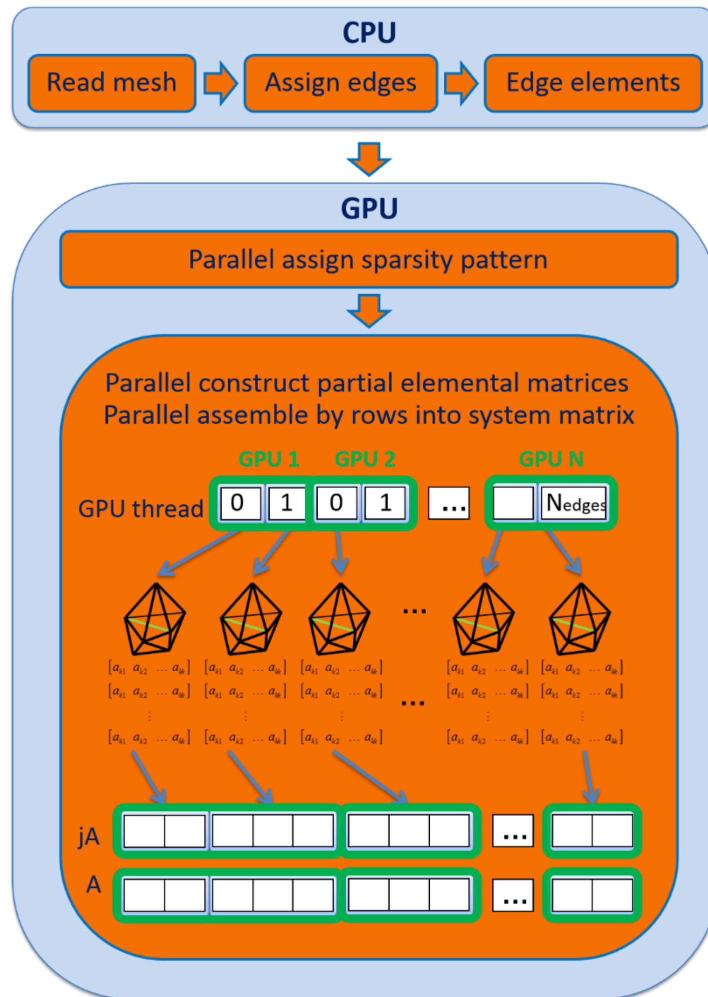


Fig. 3.5: Flow chart for the finite element assembly on multi-GPUs. A single kernel on each GPU is invoked with each thread assembling one row by directly computing the contributions from the relevant elemental matrices.

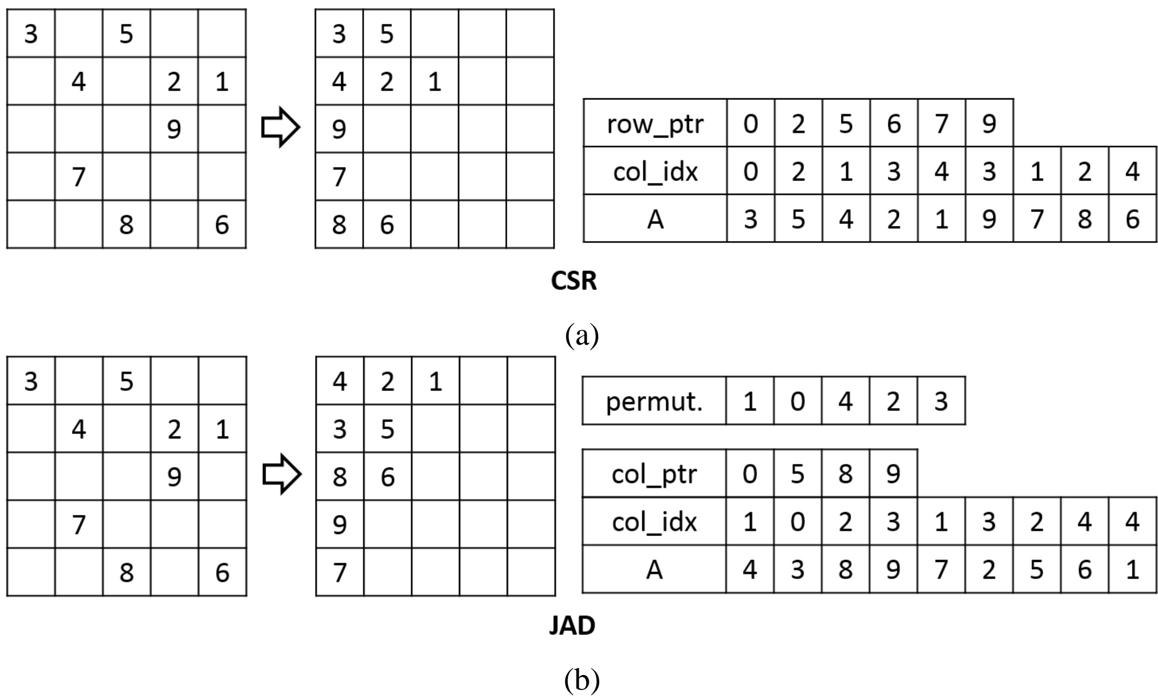


Fig. 3.6: Data structure for (a) CSR format. (b) JAD format.

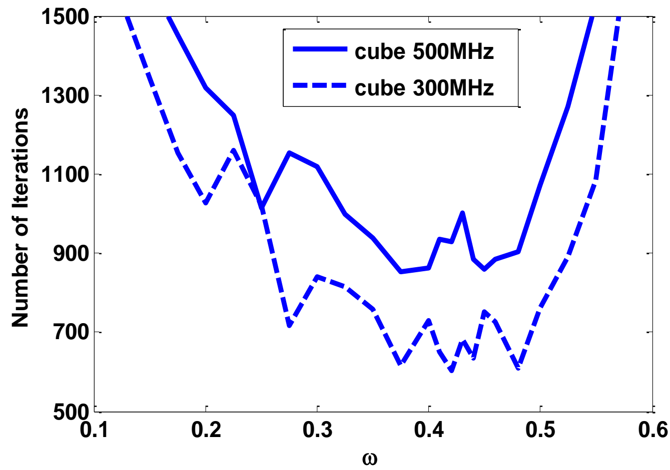


Fig. 3.7: Number of iterations with respect to the relaxation parameter ω.

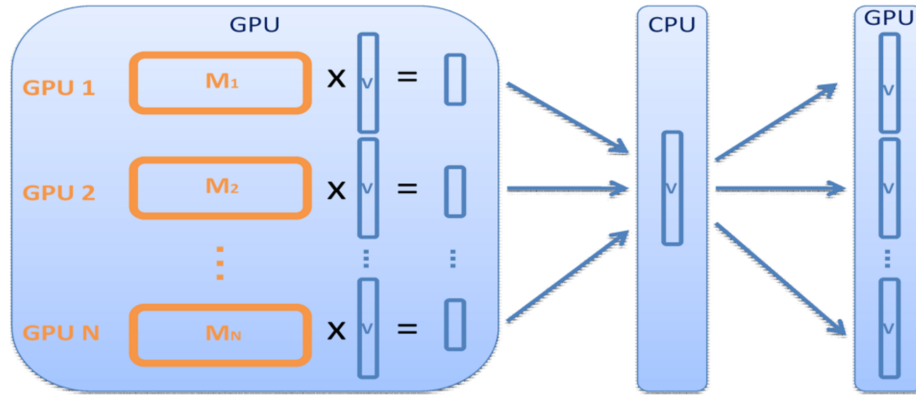
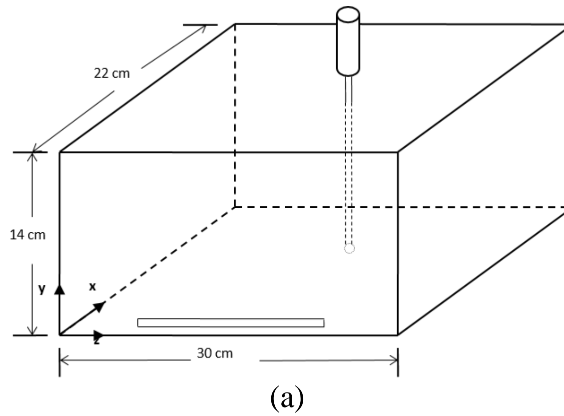
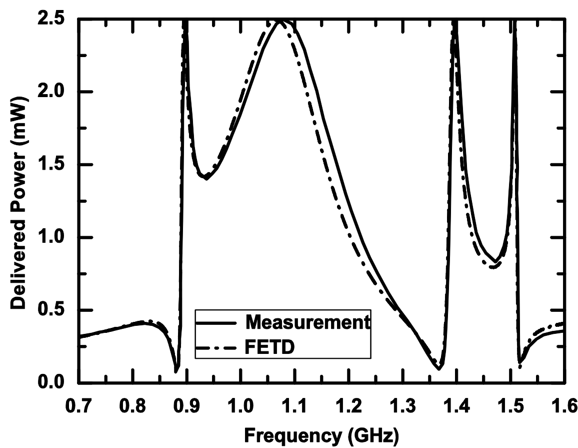
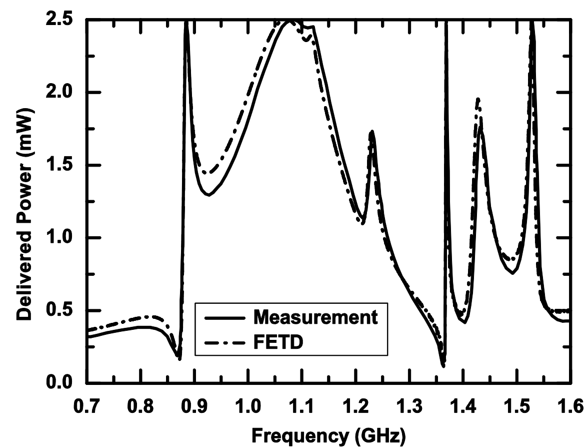


Fig. 3.8: Gathering and distribution of the product vectors through the CPU memory. The original $n \times n$ matrix is partitioned by rows into N matrices $\boldsymbol{M}_i$ of size $m_i \times n$ where $i = 1, 2, \ldots N$. The matrix multiplication $\boldsymbol{M}_i \times \boldsymbol{v}$ is computed on GPU i, and all partial results are gathered back to the CPU to form the complete product vector.



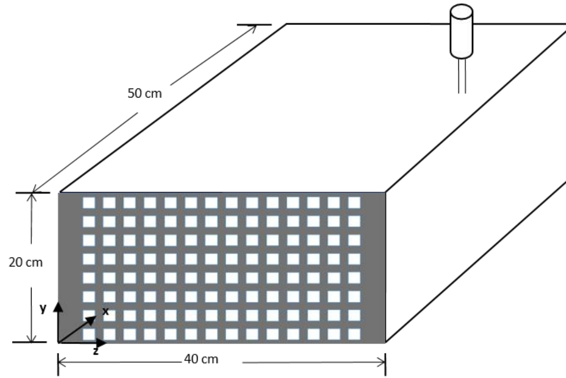
(a)



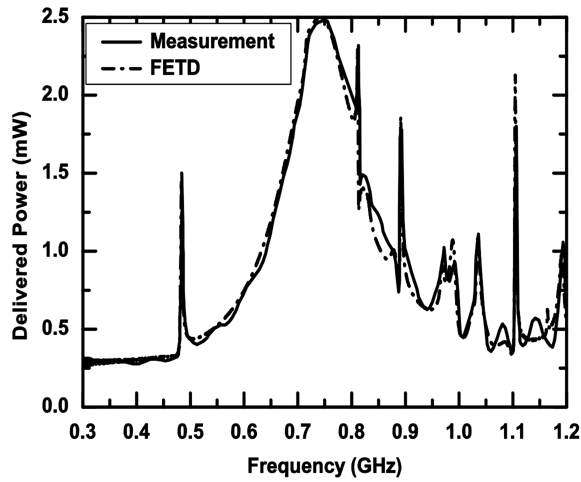


(b) (c)

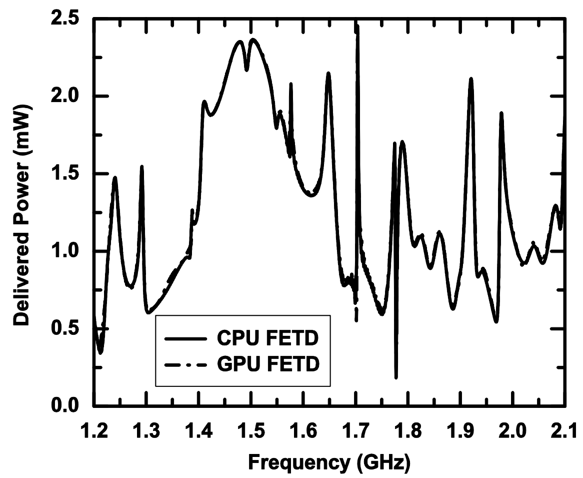
Fig. 3.9: (a) Geometry of the slotted shielding enclosure. (b) Delivered power at the coaxial feed from 0.7 to 1.6GHz without the slot. (c) Delivered power at the coaxial feed from 0.7 to 1.6GHz with the slot.



(a)



(b)



(c)

Fig. 3.10: (a) Geometry of the apertured shielding enclosure. (b) Delivered power at the coaxial feed from 0.3 to 1.2GHz. (c) Delivered power at the coaxial feed from 1.2 to 2.1GHz.

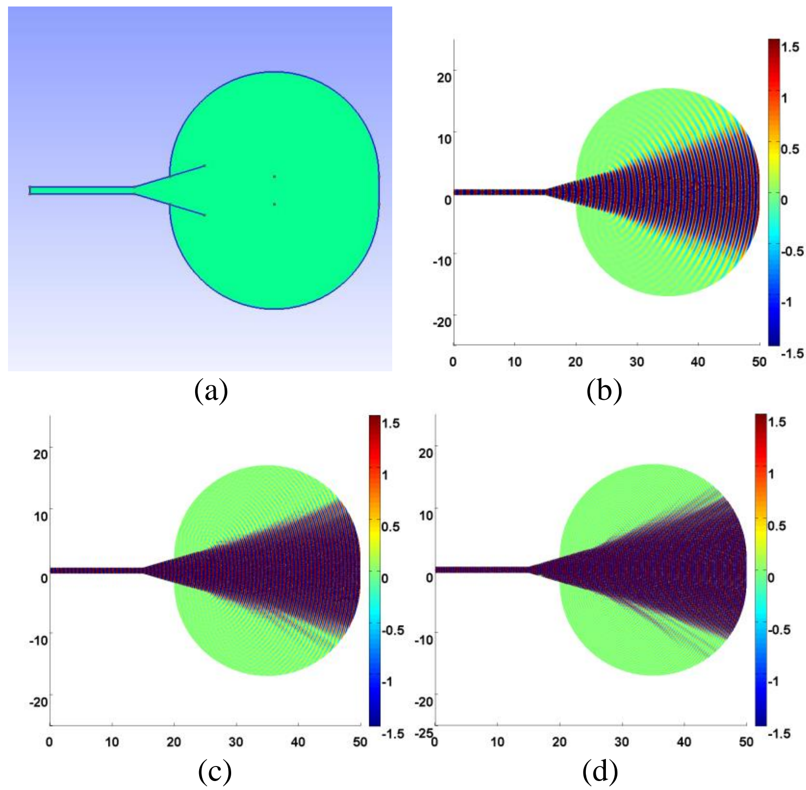


Fig. 3.11: (a) Geometry of the horn antenna and the computation domain. Resulting field distribution at (b) 300MHz, (c) 500MHz, and (d) 800MHz.

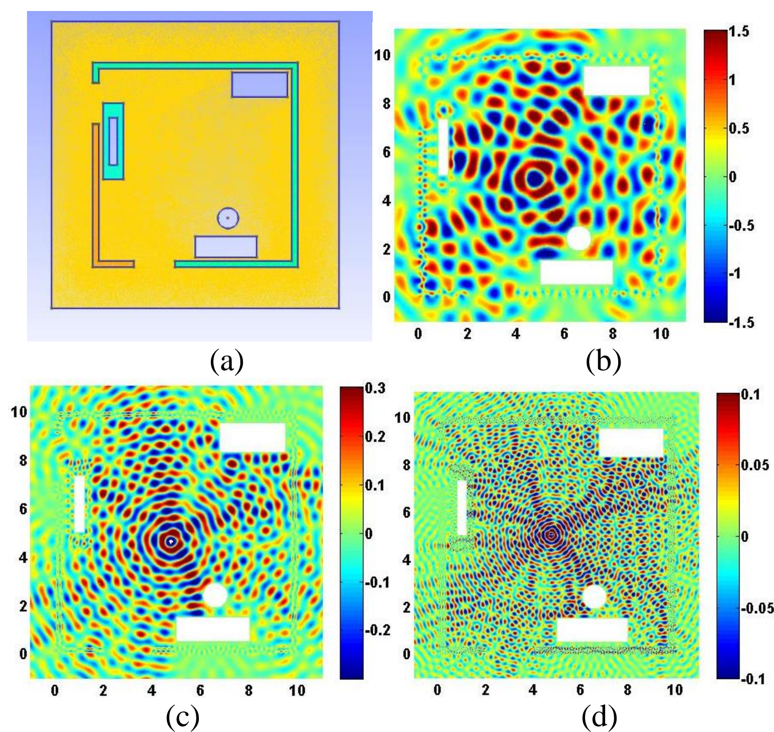


Fig. 3.12: (a) Geometry of the room. Resulting field distribution at (b) 300MHz, (c) 500MHz, and (d) 1000MHz.

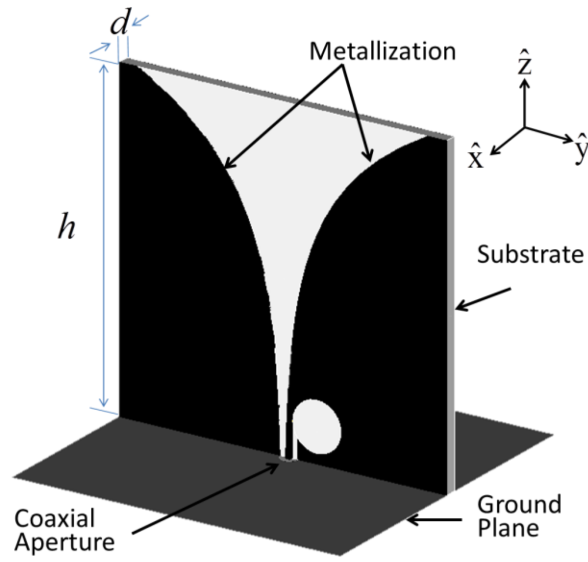


Fig. 3.13: Geometry of a Vivaldi antenna.

Table 3.1: Comparison of the total computation time using the JAD and the CSR storage formats.

Dimension	CuSparse (s)	Iterations	JAD (s)	Iterations	Speedup
172,067	6.35	1,450	5.10	1,523	**1.25**
720,051	29.12	1,793	25.63	1,664	**1.14**
1,605,614	81.11	2,422	78.79	2,291	**1.03**

Table 3.2: Comparison of Jacobi, SSOR, and SSOR-AI preconditioners.

Preconditioner	CPU (s)	Iterations	GPU (s)	Iterations	Speedup
Jacobi	59.31	1,369	6.35	1,545	**9.34**
SSOR	33.70	346	9.22	400	**3.66**
SSOR-AI	45.50	546	3.78	505	**12.04**

Table 3.3: Timing comparison between the MKL direct sparse solver on CPU and the ILU preconditioned iterative solver on GPU for the case without the slot.

Processor/Method	Factorization (s)	Iterations	Single step (s)	Total time (s)	Speedup
CPU/Pardiso (1 core)	155.0	N/A	2.1	21,206.0	**1**
CPU/Pardiso (4 cores)	53.4	N/A	1.2	12,104.4	**1.8**
CPU/Pardiso (12 cores)	34.2	N/A	0.84	8,485.2	**2.5**
GPU/BiCGStab + ILU	6.5	4~8	0.53	5,351.2	**4.0**

Table 3.4: Timing comparison between the MKL direct sparse solver on CPU and the ILU preconditioned iterative solver on GPU for the case with the slot.

Processor/Method	Factorization (s)	Iterations	Single step (s)	Total time (s)	Speedup
CPU/Pardiso (1 core)	111.3	N/A	1.9	19,163.3	**1**
CPU/Pardiso (4 cores)	41.2	N/A	1.0	10,493.2	**1.8**
CPU/Pardiso (12 cores)	29.5	N/A	0.72	7,281.5	**2.6**
GPU/BiCGStab + ILU	4.2	2~4	0.25	2,752.2	**7.5**

Table 3.5: Timing comparison between the MKL direct sparse CPU solver and the ILU preconditioned iterative GPU solver for a simulation with 873,005 unknowns.

Processor/Method	Factorization (s)	Iterations	Single step (s)	Total time (s)	Speedup
CPU/Pardiso (1 core)	265.0	N/A	3.1	124,350.0	**1**
CPU/Pardiso (4 cores)	90.8	N/A	1.8	70,575.8	**1.8**
CPU/Pardiso (12 cores)	58.3	N/A	1.4	56,943.3	**2.2**
GPU/BiCGStab + ILU	4.9	2~4	0.25	10,117.8	**12.3**

Table 3.6: Timing comparison between the MKL direct sparse CPU solver and the ILU preconditioned iterative GPU solver for a simulation with 1,674,458 unknowns.

Processor/Method	Factorization (s)	Iterations	Single step (s)	Total time (s)	Speedup
CPU/Pardiso (1 core)	1,177.2	N/A	9.0	361,330.2	**1**
CPU/Pardiso (4 cores)	358.0	N/A	4.4	175,711.0	**2.1**
CPU/Pardiso (12 cores)	223.4	N/A	3.4	135,576.4	**2.7**
GPU/BiCGStab + ILU	9.4	2~4	0.45	16,473.7	**21.9**

Table 3.7: Horn antenna simulation timing comparison between single-core CPU and single GPU.

Frequency (MHz)		300	500	800
Unknowns		100,189	280,578	717,179
CPU Timing (s)	**Assembly**	0.15	0.49	1.40
	Solution	87.53	701.34	6,428.81
	Iterations	6,196	14,601	31,998
GPU Timing (s)	**Preprocessing**	0.18	0.53	1.42
	Assembly	0.01	0.03	0.08
	Solution	20.13	97.48	461.75
	Iterations	5,585	15,037	32,587
Speedup		**4.31**	**7.16**	**13.88**

Table 3.8: Horn antenna simulation timing comparison between quad-core CPU and single GPU.

Frequency (MHz)		300	500	800
Unknowns		100,189	280,578	717,179
CPU Timing (s)	Assembly	0.15	0.49	1.40
	Solution	43.65	311.61	3,134.70
	Iterations	6,196	14,601	31,998
GPU Timing (s)	Preprocessing	0.18	0.53	1.42
	Assembly	0.01	0.03	0.08
	Solution	20.13	97.48	461.75
	Iterations	5,585	15,037	32,587
Speedup		**2.15**	**3.16**	**6.77**

Table 3.9: Field profiling timing comparison between quad-core CPU and single GPU.

Frequency (MHz)		300	500	1000
Unknowns		91,803	244,072	993,968
CPU Timing (s)	Assembly	0.11	0.39	1.43
	Solution	81.72	373.16	6,561.84
	Iterations	16,856	21,171	65,441
GPU Timing (s)	Preprocessing	0.14	0.38	1.74
	Assembly	0.01	0.02	0.09
	Solution	24.51	87.44	1,072.83
	Iterations	16,506	21,386	65,321
Speedup		**3.32**	**4.27**	**6.11**

Table 3.10: Simulation of a monopole antenna with 436,920 unknowns.

Preconditioner	CPU 4-cores (s)	Iterations	GPU (s)	Iterations	Speedup
Jacobi	617.17	11,050	131.52	11,781	**4.69**
SSOR-AI	389.30	3,475	76.55	3,613	**5.09**
SSOR-AI + Jacobi	351.92	3,226	72.96	3,410	**4.82**

Table 3.11: Simulation of a Vivaldi antenna with 1,221,796 unknowns.

Preconditioner	CPU 4-cores (s)	Iterations	GPU (s)	Iterations	Speedup
Jacobi	N/A	Failed	N/A	Failed	**N/A**
SSOR-AI	8,698.33	24,933	1,084.72	19,191	**8.02**
SSOR-AI +Jacobi	2,222.34	6,500	379.96	6,070	**5.85**

Table 3.12: Simulation of a monopole antenna array with 2,085,876 unknowns.

	CPU 4-cores	**1 GPU**	**2 GPU**	**4 GPU**
FEM Init (s)	3.66	N/A	6.00	6.00
CUDA Init (s)	N/A	N/A	4.30	12.43
Preassembly (s)	N/A	N/A	0.70	0.31
Assembly (s)	13.64	N/A	1.29	0.78
Iterations	859	N/A	981	791
Solution time (s)	600.37	N/A	119.25	86.70
Postprocessing (s)	10.95	N/A	0.17	0.13
Total time (s)	628.62	N/A	131.71	106.35
Speedup	**1**	**N/A**	**4.77**	**5.91**

Table 3.13: 2-GPU single SpMV and transfer timing.

	SpMV (ms)	**Gathering (ms)**	**Barrier (ms)**	**Redistribution (ms)**
GPU #1	7.53	3.97	1.82	3.99
GPU #2	7.93	3.97	1.49	3.89

Table 3.14: 4-GPU single SpMV and transfer timing.

	SpMV (ms)	**Gathering (ms)**	**Barrier (ms)**	**Redistribution (ms)**
GPU #1	3.72	2.57	1.33	8.90
GPU #2	3.95	2.70	0.91	8.97
GPU #3	4.14	3.47	1.05	7.87
GPU #4	3.94	3.41	1.30	7.87

Table 3.15: Simulation of a 5×5 monopole array with 3,135,869 unknowns.

	CPU 4-cores	**1 GPU**	**2 GPU**	**4 GPU**
FEM Init (s)	5.46	9.00	9.00	9.00
CUDA Init (s)	N/A	0.27	4.97	12.62
Preassembly (s)	N/A	1.83	1.05	0.47
Assembly (s)	16.92	3.54	2.11	1.26
Iterations	889	920	944	1,048
Solution time (s)	876.31	134.69	160.17	172.63
Postprocessing (s)	16.21	0.45	0.30	0.16
Total time (s)	**914.90**	**149.78**	**177.60**	**196.14**
Speedup	**1**	**6.11**	**5.15**	**4.66**

Table 3.16: Simulation of a 7×7 monopole array with 5,242,996 unknowns.

	CPU 4-cores	**1 GPU**	**2 GPU**	**4 GPU**
FEM Init (s)	9.41	N/A	15.00	15.00
CUDA Init (s)	N/A	N/A	5.27	15.29
Preassembly (s)	N/A	N/A	1.71	0.73
Assembly (s)	34.20	N/A	3.45	2.11
Iterations	1,211	N/A	1,425	1,317
Solution time (s)	2,085.99	N/A	376.62	359.14
Postprocessing (s)	30.10	N/A	0.39	0.25
Total time (s)	**2,159.70**	**N/A**	**402.44**	**392.52**
Speedup	**1**	**N/A**	**5.37**	**5.50**

Table 3.17: Simulation of a 9×9 monopole array with 7,755,542 unknowns.

	CPU 4-cores	**1 GPU**	**2 GPU**	**4 GPU**
FEM Init (s)	14.36	N/A	N/A	22.00
CUDA Init (s)	N/A	N/A	N/A	11.98
Preassembly (s)	N/A	N/A	N/A	1.05
Assembly (s)	51.52	N/A	N/A	3.04
Iterations	1,616	N/A	N/A	1,441
Solution time (s)	4,466.03	N/A	N/A	580.06
Postprocessing (s)	41.51	N/A	N/A	0.40
Total time (s)	**4,573.42**	**N/A**	**N/A**	**618.53**
Speedup	**1**	**N/A**	**N/A**	**7.39**

CHAPTER 4
GPU ACCELERATION OF THE ELEMENT-LEVEL TIME-DOMAIN METHODS

4.1 Introduction

This chapter focuses on the adaptation of the DFDD-ELD method to the GPU environment. Specifically, a carefully arranged data structure and GPU thread allocation are discussed in detail to ensure a maximum throughput and a high acceleration on GPUs. The chapter is organized as follows: Section 4.2 formulates the DFDD-ELD algorithm. Section 4.3 details the implementation of the DFDD-ELD algorithm on a large-scale hybrid MPI-CUDA computing system as well as the acceleration of the DFDD-ELD algorithm for arbitrarily large finite arrays. Section 4.4 formulates the DGTD-Central and DGTD-Upwind algorithms, and also analyzes the algorithms' process breakdowns and their expected acceleration performances. Section 4.5 presents examples of electromagnetic analysis and documents the GPU speedups for each case. The GPU hardware used in this chapter is, unless otherwise specified, NVIDIA's Fermi M2090 of Compute Capability 2.0, with 6GB of memory. For the CUDA software, this chapter uses CUDA v.5.0.

4.2 DFDD-ELD Formulation

The DFDD-ELD scheme is a variant of the DFDD algorithm, where the domain decomposition is carried all the way down to the element level. The DFDD-ELD scheme incorporating the capability of handling lossy and dispersive materials is very similar to that of the DFDD algorithm, whose derivation is detailed in [55]. The following derivation pertains to a finite element, which is treated as a single subdomain and interacts with its immediate adjacent elements. Consider a general isotropic, lossy, and dispersive medium with the constitutive relations given by

$$\vec{D}(t) = \varepsilon_0 \varepsilon_\infty \vec{E}(t) + \varepsilon_0 \chi_e(t) \otimes \vec{E}(t) \tag{4.1}$$

$$\vec{B}(t) = \mu_0 \mu_\infty \vec{H}(t) + \mu_0 \chi_m(t) \otimes \vec{H}(t) \tag{4.2}$$

where ε_∞ and μ_∞ are the relative permittivity and permeability of the medium at infinite frequency, and $\chi_e(t)$ and $\chi_m(t)$ are the time-dependent electric and magnetic susceptibilities convoluting with the fields in the time domain. Applying Equations (4.1) and (4.2) to Maxwell's equations, we obtain the second-order wave equation

$$\nabla\times\frac{\partial\vec{H}}{\partial t}-\varepsilon_0\varepsilon_\infty\frac{\partial^2\vec{E}}{\partial t^2}-\sigma_e\frac{\partial\vec{E}}{\partial t}-\varepsilon_0\chi_e(t)\otimes\frac{\partial^2\vec{E}}{\partial t^2}=\frac{\partial\vec{J}_{\text{imp}}}{\partial t} \tag{4.3}$$

where σ_e denotes the electric conductivity and $\vec{J}_{\text{imp}}$ the impressed electric current density. Next, testing Equation (4.3) with a vector basis function $\vec{N}_i^e$ and then integrating over the volume of element e, we obtain

$$\begin{aligned}&\iiint_{V_e}\left[\frac{1}{\mu_\infty}(\nabla\times\vec{N}_i^e)\cdot(\nabla\times\vec{E})-\frac{Z_0}{c_0}(\nabla\times\vec{N}_i^e)\cdot\vec{Q}+\frac{1}{c_0^2}\varepsilon_\infty\vec{N}_i^e\cdot\frac{\partial^2\vec{E}}{\partial t^2}+\frac{Z_0}{c_0}\sigma_e\vec{N}_i^e\cdot\frac{\partial\vec{E}}{\partial t}+\frac{1}{c_0^2}\vec{N}_i^e\cdot\vec{G}\right]dV\\&=\frac{Z_0}{c_0}\oint\!\!\oint_{S_e}(\hat{n}\times\vec{N}_i^e)\cdot\left(\hat{n}\times\frac{\partial\vec{J}_s}{\partial t}\right)dS-\frac{Z_0}{c_0}\iiint_{V_e}\vec{N}_i^e\cdot\frac{\partial\vec{J}_{\text{imp}}}{\partial t}dV\end{aligned} \tag{4.4}$$

where V_e denotes the volume of the element, S_e denotes its surface, and

$$\vec{J}_s=\hat{n}\times\vec{H} \tag{4.5}$$

$$\vec{G}=\chi_e(t)\otimes\frac{\partial^2\vec{E}}{\partial t^2} \tag{4.6}$$

$$\vec{Q}=-\frac{1}{\mu_0\mu_\infty}\left(\mu_0\chi_m\otimes\frac{\partial\vec{H}}{\partial t}+\sigma_m\vec{H}+\vec{M}_{\text{imp}}\right) \tag{4.7}$$

Clearly, $\vec{G}$ accounts for the effect of the electrical dispersion and $\vec{Q}$ accounts for the effect of the magnetic loss and dispersion. For a Debye or Lorentz medium, whose susceptibilities can be expressed as

$$\chi_e(t)=\text{Re}(a_e e^{-b_e\Delta t})u(t) \tag{4.8}$$

and

$$\chi_m(t)=\text{Re}(a_m e^{-b_m\Delta t})u(t) \tag{4.9}$$

where $u(t)$ denotes the unit step function, we can approximate the convolution in Equation (4.6) with an explicit recursive convolution [55] after expanding the electric field using vector basis functions into $\vec{E}=\sum_j \vec{N}_j^e e_j$, resulting in

$$\chi_e \otimes \left.\frac{\partial^2 \vec{E}}{\partial t^2}\right|_{t=n\Delta t} \approx \sum_j \vec{N}_j^e [\mathrm{Re}(a_e c_e)\ddot{e}_j^n + \mathrm{Re}(a_e d_e \varphi_j^n)] \tag{4.10}$$

where $\ddot{e}_j$ denotes the second time derivative of e_j, and

$$\varphi_j^n = e^{-b_e \Delta t}\varphi_j^{n-1} + \frac{e_j^n - 2e_j^{n-1} + e_j^{n-2}}{(\Delta t)^2} \tag{4.11}$$

$$c_e = \frac{1}{b_e}(1 - e^{-b_e \Delta t/2}) \tag{4.12}$$

$$d_e = \frac{1}{b_e}(1 - e^{-b_e \Delta t})e^{-b_e \Delta t/2} \tag{4.13}$$

Similarly, we can expand the magnetic field as $\vec{H} = \sum_j \vec{N}_j^e h_j$ and approximate the convolution in Equation (4.7) as

$$\chi_m \otimes \left.\frac{\partial \vec{H}}{\partial t}\right|_{t=n\Delta t} \approx \sum_j \vec{N}_j^e [\mathrm{Re}(a_m c_m)\dot{h}_j^n + \mathrm{Re}(a_m d_m \psi_j^n)] \tag{4.14}$$

where $\dot{h}_j$ denotes the first time derivative of h_j, and

$$\psi_j^n = e^{-b_m \Delta t}\psi_j^{n-1} + \frac{h_j^{n-1/2} - h_j^{n-3/2}}{\Delta t} \tag{4.15}$$

$$c_m = \frac{1}{b_m}(1 - e^{-b_m \Delta t/2}) \tag{4.16}$$

$$d_m = \frac{1}{b_m}(1 - e^{-b_m \Delta t})e^{-b_m \Delta t/2} \tag{4.17}$$

By substituting in the expansion of the electric field, Equation (4.4) is then discretized into

$$[S_e]\{e\} + [M_e]\frac{\partial^2 \{e\}}{\partial t^2} + [B_e]\frac{\partial \{e\}}{\partial t} = [P_e]\frac{\partial \{j_s\}}{\partial t} + \{f_e\} + [T_e]\{q\} + [U_e]\{g\} \tag{4.18}$$

where

$$S_e(i,j) = \iiint_{V_e} \frac{1}{\mu_\infty}(\nabla \times \vec{N}_i^e)\cdot(\nabla \times \vec{N}_j^e)dV \tag{4.19}$$

$$M_e(i,j) = \frac{1}{c_0^2}\iiint_{V_e} [\varepsilon_\infty + \mathrm{Re}(a_e c_e)]\vec{N}_i^e \cdot \vec{N}_j^e dV \tag{4.20}$$

$$B_e(i,j) = \frac{Z_0}{c_0} \iiint_{V_e} \sigma_e \vec{N}_i^e \cdot \vec{N}_j^e dV \tag{4.21}$$

$$P_e(i,j) = \frac{Z_0}{c_0} \iint_{S_e} (\text{n盗}\, \vec{N}_i^e) \cdot (n \times \vec{N}_j^s) dS \tag{4.22}$$

$$T_e(i,j) = \frac{Z_0}{c_0} \iiint_{V_e} (\nabla \times \vec{N}_i^e) \cdot \vec{N}_j^e dV \tag{4.23}$$

$$U_e(i,j) = -\frac{1}{c_0^2} \iiint_{V_e} \vec{N}_i^e \cdot \vec{N}_j^e dV \tag{4.24}$$

$$f_e(i) = -\frac{Z_0}{c_0} \iiint_{V_e} \vec{N}_i^e \cdot \frac{\partial \vec{J}_{\text{imp}}}{\partial t} dV \tag{4.25}$$

where we have also expanded $\vec{J}_s$ as $\vec{J}_s = \sum_j \vec{N}_j^s j_{s,j}$ and $\vec{Q}$ as $\vec{Q} = \sum_j \vec{N}_j^e q_j$, and the vector $\{g\}$ is given by $g_j = \text{Re}(a_e d_e \varphi_j)$. A similar procedure can be adopted for the discretization of the magnetic field equation, which serves as the dual of Equation (4.18). To couple the fields from the adjacent elements, we replace $\vec{J}_s$ and $\vec{M}_s$ with $\vec{J}_s = \hat{n} \times \vec{H}^+$ and $\vec{M}_s = \hat{n} \times \vec{E}^+$, where $\vec{H}^+$ and $\vec{E}^+$ are the magnetic and electric fields from the adjacent elements.

Like that in the original DFDD, the Newmark-beta method is used in the marching of Equation (4.18) in time domain. Note that in the DFDD formulation, the maximum time-step size is constrained by the smallest finite element on the subdomain interfaces, and therefore the maximum time-step size of the DFDD-ELD algorithm is constrained by the smallest finite element in the entire computation domain. Equation (4.18) can be further discretized into

$$\{e\}^{n+1} = [A_1]\{e\}^n - [A_2]\{e\}^{n-1} + [P](\{j_s\}^{n+1/2} - \{j_s\}^{n-1/2}) + [A_0]^{-1}\{f_e\}^n + [T]\{q\}^n + [U]\{g\}^n \tag{4.26}$$

where

$$[A_1] = [A_0]^{-1}\left[\frac{2}{(\Delta t)^2}[M_e] - \frac{1}{2}[S_e]\right] \tag{4.27}$$

$$[A_2] = [A_0]^{-1}\left[\frac{1}{(\Delta t)^2}[M_e] + \frac{1}{4}[S_e] - \frac{1}{2\Delta t}[B_e]\right] \tag{4.28}$$

$$[P] = \frac{1}{\Delta t}[A_0]^{-1}[P_e] \tag{4.29}$$

$$[T] = [A_0]^{-1}[T_e] \tag{4.30}$$

$$[U]=[A_0]^{-1}[U_e] \tag{4.31}$$

in which

$$[A_0]=\frac{1}{(\Delta t)^2}[M_e]+\frac{1}{4}[S_e]+\frac{1}{2\Delta t}[B_e] \tag{4.32}$$

The vectors $\{j_s\}^{n+1/2}$ and $\{j_s\}^{n-1/2}$ pertain to the equivalent surface current $\vec{J}_s=\hat{n}\times\vec{H}^+$ on the boundary between the element-level subdomains, and they are calculated from the dual equation of Equation (4.26) for the magnetic field and discretized at half time steps. Furthermore, the vector $\{f_e\}^n$ is contributed by the impressed current, and the vectors $\{g\}^n$ and $\{q\}^n$ are calculated recursively based on Equations (4.11) and (4.15), respectively. A similar procedure is adopted for the discretization of the magnetic field equation at half time steps, acting as the dual of Equation (4.26) and thus creating a leap-frog scheme for the DFDD-ELD algorithm.

4.3 GPU Implementation

The GPU implementation of the DFDD-ELD algorithm is presented in this section, as well as the data structure involved in accelerating the solution process. Section 4.3.1 breaks down the computation of a single time-step into several processes. Section 4.3.2 proposes an acceleration scheme applicable to the time-marching process. Section 4.3.3 discusses the remedy for the random access of device memory. Section 4.3.4 describes the local time-stepping technique. Section 4.3.5 applies the GPU scheme in a cluster environment with the combination of MPI and CUDA. Section 4.3.6 presents the modifications needed for problems with a finite array configuration. For clarity, the following presentation focuses only on the time-marching of the electric field, while the procedure for the time-marching of the magnetic field is completely analogous and is thus intentionally omitted.

4.3.1 Time-Marching Step Breakdown

The DFDD-ELD algorithm treats each tetrahedral element as an individual subdomain, with the surfaces of each element acting as the subdomain interface. Some observations can be drawn from Equation (4.26) to guide the parallelization effort. First, it is shown that for every time step, the system matrices remain constant and are localized to a single element. This means that the application of the updating matrices for each element can naturally be performed in

parallel. Second, it is observed that the solution update for the field unknowns within each element requires only the fields from the same element and adjacent elements. The proximity of the data needed induces a hidden benefit on the GPU architecture by utilizing the L1 and L2 caches. Lastly, since the system matrices in Equation (4.26) are very small, $[A_0]$ can be easily inverted and combined with the other matrices during the assembly stage. Thus during each time step, field unknowns are updated using simple matrix-vector multiplications instead of solving the system of equations as found in the ordinary finite element algorithms.

The acceleration of the DFDD-ELD algorithm begins with the construction of the system matrices on the host, which are then factorized and transferred to the device. This process is done only once before the time marching, and the computation is cheap compared to the time-marching process. Next, the solution vectors are allocated on the device memory, and GPU kernels are invoked to update the field unknowns in parallel. Since the tasks performed in each time step are completely identical, the main parallelization effort is directed to reducing the total computation time per time step. Looking at Equation (4.26), the DFDD-ELD updating algorithm mainly consists of matrix-vector multiplications which are memory-intensive, and therefore the speed of the time-marching process is limited by how fast the memory is accessed. This calls for a careful assignment of GPU threads as well as the corresponding memory data structure, where the obtainable bandwidth will eventually limit the maximum possible speedup.

4.3.2 Parallelization Using CUDA

To accelerate the matrix-vector multiplications, several criteria need to be met in order to fully utilize the architecture of the GPU. First, each warp needs to access the device memory in a coalesced fashion. To meet this criterion, each GPU thread is dedicated to the updating of a single field unknown in the solution vector. This ensures that the GPU threads read and write the solution vectors from the device memory in a coalesced fashion. Next, the number of threads invoked per block needs to be calculated precisely according to the degrees of freedom (or unknowns) per element, which is relevant to the order of the basis functions. This is due to the fact that in order to properly carry out the matrix-vector multiplications for each element, the unknowns of an element have to be grouped into the same block for coherent processing. Also, since the GPU threads are concurrently executed in warps of 32 threads, the number of solution unknowns assigned to a single block has to be a multiple of 32 in order to fully utilize the GPU's

parallel architecture. Therefore, once the basis order for a problem is specified, the number of unknowns per element is determined, and the number of threads allocated per block has to be divisible by both the number of unknowns per element and the number 32. For example, for mixed first-order, full first-order, and mixed second-order basis functions, the number of unknowns per element is 6, 12, and 20, respectively. To fully utilize every warp, the number of threads allocated per block then has to be a multiple of 96, 96, and 160, which translates to multiples of 16, 8, and 8 elements per block, respectively. As Tables 4.1-4.3 show, this utilization roughly provides a ~9% overall speedup compared to a hard-coded number, such as the typical 512 or 1024 threads per block. Figure 4.1 shows the thread and block allocation for the integration kernels, where the number of elements per block corresponds to the basis function order, and the number of blocks for the grid is obtained by dividing the total number of finite elements by the number of elements per block. Note that although both the blocks and threads can be allocated in three dimensions, we only allocate them in one dimension since our allocation is well within the bound of the allowed size of a dimension.

Finally, since device memory access carries a high latency, the number of device memory accesses needs to be kept to a minimum. The conventional matrix-vector multiplication multiplies and sums each matrix row with the same multiplier vector, creating a serial reduction as well as a redundant access to the vector. To overcome this, it is optimal to parallelize each matrix-vector multiplication by column instead. Figure 4.2 illustrates the new matrix-vector multiplication approach, which thoroughly utilizes the GPU threads while minimizing memory access. The threads first pre-fetch the multiplier vector in a coalesced fashion from the device memory to the local shared memory, then accumulate the resulting products in the shared memory by looping over each matrix column in yet another coalesced fashion. This creates a shared memory broadcast for the multiplier vector, and also avoids the need of performing reduction as found in the traditional matrix-vector multiplication by rows, which is only semi-parallelizable. The accumulated result vector in the shared memory is then transferred back to the device memory in a coalesced fashion. It can be deduced that when higher-order basis functions are used, the parallel scheme still retains the same bandwidth with its completely parallelized and coalesced features, with the only difference being the number of threads and blocks invoked per GPU kernel.

4.3.3 Random Access Acceleration

The above scheme is readily applicable to the GPU volume matrix-vector multiplication for the contribution of $\{e\}^n$ and $\{e\}^{n-1}$ using $[A_1]$ and $[A_2]$, respectively, while the contribution from the surface integral matrix $[P]$ requires a slight modification. Based on Equation (4.22), $[P]$ is related to the contribution of surface integrals of the magnetic fields from the neighboring elements. That is, the unknowns in an element are updated using the neighboring magnetic fields $\{h^+\}^{n+1/2}$ and $\{h^+\}^{n-1/2}$ from all four surfaces, and therefore the dimension of $[P]$ is $numTetDof \times (4 \times numTriDof)$, where $numTriDof$ is the degree of freedom on a triangular surface. Note that according to the proposed matrix-vector multiplication scheme, the GPU threads still access matrix $[P]$ in the same coalesced fashion, albeit performing column summations across all surface unknowns. The main difference lies in the fetching of the multiplier vector from the neighboring magnetic fields, which belong to separate elements and are thus stored in random locations in the solution vectors. Since accessing the multiplier vectors is no longer coalesced, this is the biggest bottleneck of the entire GPU scheme.

Fortunately, utilizing the GPU cache can slightly alleviate this problem. Recall that a GPU warp accesses a consecutive 128 bytes of device global memory and temporarily stores the segment in the on-chip L1 and L2 caches, regardless of the amount of the actual data needed. The idea is that if the neighboring fields are in physically close proximity, a single GPU block will have a higher chance of finding all the neighboring fields for its elements in fewer global memory accesses, since many of the neighboring fields would have been collaterally fetched by the neighboring threads and placed in the on-chip cache. To utilize the cached memory for random access, the graph partitioning program METIS [78] is employed to decompose the computation domain into regions of elements with physical proximity. Specifically, the number of elements per region should roughly be equivalent to the number of elements per block for better memory caching. Due to the limitation of the graph partitioning program, not every physical partition will end up containing precisely the number of elements per block. However, this utilization of the memory cache through physical proximity can still achieve a descent increase in the overall bandwidth. Tables 4.1-4.3 document the speedup achieved using a careful thread allocation and cache utilization through element rearrangement. It is shown that as the

number of total elements goes up and/or the complexity of the problem geometry increases, these two optimizations combined will give a higher speedup compared to a raw GPU acceleration. Once the solution of the electric field at time step $n+1$ is computed, field vectors at the next time step can be easily updated by simply rearranging the pointers to the solution vectors, so that the solution vector for $\{e\}^{n+1}$ becomes $\{e\}^{n}$, the solution vector for $\{e\}^{n}$ becomes $\{e\}^{n-1}$, and the solution vector for $\{e\}^{n-2}$ can be reused for the computation of the following time step at $n+2$.

To accelerate the lossy and dispersive related computation from each element, note from Equations (4.10) and (4.14) that the convolutions are approximated by contributions from the recursive terms plus the fields from the past time steps, all of which are localized within the same element. This localization is advantageous for the parallelization of the explicit convolution algorithm since no extra inter-element communications are needed for the incorporation of the lossy and dispersive contributions.

To update the lossy and dispersive contributions for the electric field, it is necessary to compute the first derivatives of the magnetic field for $\{q\}$ as in Equation (4.15), and the second derivative of the electric field for $\{g\}$ as in Equation (4.11). Since not all elements in the computation domain are necessarily lossy and dispersive, it is not desirable to compute the field derivatives for the entire computation domain. Instead, only the lossy and dispersive elements are parallelized for the corresponding contribution, and the field derivatives are computed on the fly using shared memory to reduce memory cost. The resulting vectors from the convolutions are first stored in the shared memory and then multiplied to $[T]$ and $[U]$ using the proposed matrix-vector multiplication scheme. Figure 4.3 illustrates this process. Note that the contributions have one-to-one correlation with the element unknowns and GPU threads, hence they are naturally parallelizable on GPU with coalesced matrix access. Moreover, since it is most likely that the lossy and dispersive elements are in close proximity to each other both geometrically and in terms of data structure, the device memory cache still retain its benefit to a certain degree when accessing the semi-contiguous multiplier vectors.

4.3.4 Incorporation of the Local Time-Stepping Technique

The local time-stepping (LTS) technique has been developed to alleviate the time step constraints due to the stability condition of the finite elements. It is well known that for explicit

time-stepping algorithms, the maximum global time step is determined by the smallest finite element in the computation domain. These small elements arise from the details of the object geometry, and are often much smaller compared to the wavelengths considered. These small elements restrict the global time step during the time marching of all the finite elements, most of which are far larger and thus have much more relaxed time step constraints.

While it is inefficient for every single element in the element-level decomposition algorithms to march at its own pace, the larger time steps from many of the elements can be leveraged by grouping the elements into different batches with time steps that are multiples of each other [48], [79], [80]. Figure 4.4 shows the proposed LTS technique for the DFDD-ELD algorithm, adapted from [79]. The LTS technique groups the elements into different size classes, with the maximum allowable time step for each succeeding class of larger element size being 3 times that of the preceding class. This arrangement synchronizes the E and H fields in a leap-frog fashion, which conforms to the DFDD-ELD algorithm. To adapt the LTS technique in GPU, each element is assigned a class index, which is correlated to the specific time step scaling factor of the class. The time step scaling factor begins with 1 for Class 1 elements (the smallest elements), then jumps to 3 for Class 2 elements (since the time step for Class 2 elements is 3 times larger than those in Class 1), and 9 for Class 3 elements, and so on. During every time step, while the GPU kernel still launches a thread for each element, only the elements whose class index is a divisor of the current time-marching step will be time-marched. After each time-marching step, boundary data exchange only needs to be performed between the marched elements. As shown in Fig. 4.4, all three classes of elements are marched at time-marching step 1, followed by a march during time-marching steps 2 and 3 solely by Class 1 elements, and then followed by a march during time-marching step 4 by Class 1 and Class 2 elements. This trend continues until time-marching step 9, after which the three classes align again and the whole process repeats.

4.3.5 Parallelization Using Hybrid MPI-CUDA

The above single-GPU acceleration scheme can be easily applied to a GPU cluster environment. These clusters share an infrastructure similar to that of the traditional CPU clusters, except with the addition of one or more GPUs per cluster node. To parallelize the DFDD-ELD algorithm using a hybrid MPI-CUDA scheme, METIS is used to decompose the

entire computation domain into separate regions with a similar number of element-level subdomains, which are then transferred to separate cluster nodes. The cluster nodes first construct the elemental matrices on the host, then time-march the elements using one GPU per cluster node. Note that while the METIS routine is called for the decomposition of the entire computation domain, another level of METIS routine is called in parallel by each cluster node for cache utilization through an element rearrangement, forming a two-level decomposition scheme.

While a single time-marching step across each MPI node is inherently parallel, inter-region communication needs to take place between adjacent time steps to exchange the solution fields on the inter-region boundaries. Keep in mind that although the inter-region communication across MPI nodes is relatively light compared to the CPU marching time, the communication proportion is significantly increased (from less than 1% to more than 10%) in a GPU environment due to the high GPU acceleration of the time-marching updates. To reduce the inter-region transfer time, a GPU kernel is first invoked to collect the derivative of the boundary fields in parallel from the solution vectors. The collected fields are then transferred to the adjacent regions using non-blocking MPI routines through page-locked host memory, whose data transfer rate between the host and the device memory is higher than the non-page-locked host memory. Once a region has received the boundary fields from all of its neighboring regions, it then continues with the next time-marching step on the GPU.

It is worth noting that to fully utilize the LTS technique on multiply nodes, one can potentially control the geometry decomposition routines to leave only the elements of the highest LTS class (the largest elements) on the inter-region boundaries, such that the inter-MPI data exchange due to the update of the marching elements will only need to be performed once for every several time-marching steps. If the geometry is decomposed properly, this unique feature can further speed up the total solution time by reducing the MPI communications. In this work, the LTS technique will only be applied on a single node in the second half of Section 4.5.2 to demonstrate the potential speedup.

4.3.6 Extension to Arbitrarily Large Finite Arrays

The application of DFDD-ELD to an arbitrarily large finite array is similar to that of [81], where the entire finite array configuration can be characterized by nine array unit cells,

consisting of one central cell, four edge cells, and four corner cells as shown in Fig. 4.5(a). The nine primitive unit cells are sufficient to represent the entire array configuration, thereby significantly reducing the construction and the factorization times of the FEM matrices.

The acceleration of the DFDD-ELD algorithm using the unit cell array configuration follows a similar procedure to the original MPI-enabled acceleration scheme. METIS is first used to decompose the entire computation domain into regions of array elements, as shown in Fig. 4.5(b), and the decomposition information is then transferred to separate cluster nodes. One of the most important advantages of the array configuration to the acceleration of the DFDD-ELD algorithm on GPUs lies in its reduction in memory consumption, where only the FEM matrices of the nine unit cells are required to be stored regardless of the actual dimension of the array. With the reduction in storage, a GPU will be able to handle more elements, reducing the number of nodes needed to solve a given problem, and thereby reducing the overall inter-region communication time between the cluster nodes.

4.4 DGTD Formulation and Acceleration Comparison

The implementation strategies discussed in this paper provide a general framework for the parallelization and acceleration of element-level explicit time-marching algorithms. This strategy can be readily applied to the GPU acceleration of the DGTD algorithms with a similar speedup performance because of their similarities to the DFDD-ELD algorithm. Sections 4.4.1 and 4.4.2 formulate the DGTD-Central and DGTD-Upwind algorithms. Section 4.4.3 analyzes the algorithms' process breakdown and their expected acceleration performances.

4.4.1 DGTD-Central

Incorporating Equation (4.1) to Maxwell's equations and applying the Galerkin testing procedure on a single element, we obtain

$$\begin{aligned}&\iiint_{V_e}\left[\varepsilon_\infty\varepsilon_0\vec{N}_i^e\cdot\frac{\partial\vec{E}}{\partial t}+\sigma_e\vec{N}_i^e\cdot\vec{E}+\varepsilon_0\vec{N}_i^e\cdot\vec{G}-(\nabla\times\vec{N}_i^e)\cdot\vec{H}\right]dV\\&=\oiint_{S_e}\vec{N}_i^e\cdot(\hat{n}\times\vec{H})dS-\iiint_{V_e}\vec{N}_i^e\cdot\vec{J}_{\text{imp}}dV\end{aligned}\tag{4.33}$$

where we have applied the vector identity and the divergence theorem, and

$$\vec{G} = \chi_e(t) \otimes \frac{\partial \vec{E}}{\partial t} \tag{4.34}$$

Next, we take the average of the tangential field 淦 $\vec{H} = \frac{1}{2}(n \times \vec{H} + \hat{n} \times \vec{H}^{+})$ to weakly enforce the tangential field continuity, and apply the same vector identity and the divergence theorem in reverse order. After applying the ABC and substituting in the expansion of the electric and magnetic field, we obtain

$$[M_e]\frac{\partial \{e\}}{\partial t} + [B_e]\{e\} + [S_e]\{h\} + [U_e]\{g\} = \{f_e\} + [F_{eh}]\{h^{+} - h\} + [A_{eh}]\{h\} + [A_{ee}]\{e\} \tag{4.35}$$

where the expression for $[M_e]$, $[B_e]$, $[S_e]$, $[U_e]$ and $\{f_e\}$ can be inferred from Equation (4.33), and

$$F_{eh}(i,j) = \frac{1}{2}\iint_{S_e} (\hat{n} \times \vec{N}_i^e) \cdot (\hat{n} \times \hat{n} \times \vec{N}_j^s) dS \tag{4.36}$$

$$A_{eh}(i,j) = -\frac{1}{2}\iint_{S_{ABC}} (\hat{n} \times \vec{N}_i^e) \cdot (\hat{n} \times \hat{n} \times \vec{N}_j^s) dS \tag{4.37}$$

$$A_{ee}(i,j) = -\frac{Y}{2}\iint_{S_{ABC}} (\hat{n} \times \vec{N}_i^e) \cdot (\hat{n} \times \vec{N}_j^s) dS \tag{4.38}$$

The vector $\{g\}$ is again given by $g_j = \mathrm{Re}(a_e d_e \varphi_j)$, with

$$\varphi_j^n = e^{-b_e \Delta t} \varphi_j^{n-1} + \frac{e_j^n - e_j^{n-1}}{\Delta t} \tag{4.39}$$

Equation (4.35) can be further discretized in the time domain using the central difference method into

$$\{e\}^{n+1} = [A_1]\{e\}^n - [A_2]\{h\}^{n+1/2} + [F](\{h^{+}\}^{n+1/2} - \{\mathrm{h}\}^{n+1/2}) + [A_0]^{-1}\{f_e\}^n - [U]\{g\}^n \tag{4.40}$$

where

$$[A_0] = \left[\frac{1}{\Delta t}[M_e] - \frac{1}{2}[A_{ee}] + \frac{1}{2}[B_e]\right] \tag{4.41}$$

and all other matrices involve the multiplication of $[A_0]^{-1}$ with other matrices defined in Equation (4.35). Again, a similar procedure is adopted for the discretization of the magnetic field equation at half time steps, acting as the dual of Equation (4.40), creating a leap-frog scheme for the DGTD-Central algorithm, which is very similar to the DFDD-ELD updating scheme.

4.4.2 DGTD-Upwind

The derivation of the DGTD-Upwind scheme is similar to that of the DGTD-Central scheme, except with a different choice of numerical flux. We take the solution to a 1D Riemann problem [82] to weakly enforce the tangential field continuity using the field $\vec{H} = \vec{H} + (\mathrm{Z}^{+} + \mathrm{Z})^{-1}\left[-\hat{n}\times(\vec{E}^{+} - \vec{E}) + \mathrm{Z}^{+}(\vec{H}^{+} - \vec{H})\right]$, and apply the same vector identity and the divergence theorem in reverse order. After substituting in the expansion of the electric and magnetic field, we obtain

$$[M_e]\frac{\partial\{e\}}{\partial t} + [B_e]\{e\} + [S_e]\{h\} + [U_e]\{g\} = \{f_e\} + [F_{eh}]\{h^{+} - h\} + [F_{ee}]\{\mathrm{e}^{+} - e\} \tag{4.42}$$

where $[M_e]$, $[B_e]$, $[S_e]$, $[U_e]$, $\{f_e\}$ and $\{g\}$ have the same expression as those found in the DGTD-Central scheme, and

$$F_{eh}(i,j) = \iint_{S_e} (\mathrm{Z}^{+} + \mathrm{Z})^{-1}\,\mathrm{Z}^{+}(\hat{n}\times\vec{N}_i^{e})\cdot(n\times\hat{n}\times\vec{N}_j^{s})dS \tag{4.43}$$

$$F_{ee}(i,j) = \iint_{S_e} \mathrm{Z}^{-1}(\hat{n}\times\vec{N}_i^{e})\cdot(\hat{n}\times\vec{N}_j^{s})dS \tag{4.44}$$

Contrary to the leap-frog scheme, Equation (4.42) can be cast into an ordinary differential equation (ODE) form

$$\begin{aligned}\frac{\partial\{e\}}{\partial t} &= F(t,\{e\}) = \{rhsE\} \\ &= [M_e]^{-1}\left(-[B_e]\{e\} - [S_e]\{h\} - [U_e]\{g\} + \{f_e\} + [F_{eh}]\{h^{+} - h\} + [F_{ee}]\{\mathrm{e}^{+} - e\}\right)\end{aligned} \tag{4.45}$$

and solved using a five stage, fourth order low-storage explicit Runge-Kutta (LSERK) [67]

$$\left.\begin{aligned}\{w\}_i &= \alpha_i\{w\}_{i-1} + \Delta t F(t_{i-1},\{e\}^{(i-1)}) \\ \{e\}^{(i)} &= \{e\}^{(i-1)} + \beta_i\{w\}_i\end{aligned}\right\}, \quad i = 1,2,\ldots,5, \tag{4.46}$$

where $\{e\}^{(0)} = \{e\}^{n}$, $\{e\}^{n+1} = \{e\}^{(5)}$, $t_{i-1} = t^{n} + \gamma_i\Delta t$, and the values of α_i, β_i and γ_i are given in [83]. For every solution time step, the LSERK scheme takes five minor steps to solve the ODE with higher order approximation, and hence larger solution time steps can be taken. It is worth noting that the computation of Equation (4.39) for the recursive convolution vector $\{g\}$ for the DGTD-Upwind algorithm carries the Δt term, which used to be the constant solution time step under the Newmark-Beta scheme and is now changing according to the minor $(\gamma_{i+1} - \gamma_i)\Delta t$ steps of the different stages of the LSERK scheme, which becomes

$$\varphi_j^{(i)} = e^{-b_e(\gamma_{i+1}-\gamma_i)\Delta t}\varphi_j^{(i-1)} + \frac{\beta_{i-1}\{w\}_{i-1}}{(\gamma_i - \gamma_{i-1})\Delta t} \tag{4.47}$$

where $\beta_0 = \beta_5$ and $\gamma_6 - \gamma_5 = \gamma_1 - \gamma_0 = 1 - \gamma_5$.

4.4.3 GPU Timing and Acceleration Comparison

Since the fully explicit schemes are memory-intensive, the speed of the time-marching process is limited by how fast the memory is accessed. It is thus preferred to gauge the acceleration efficiency by examine the number of matrix-vector multiplications (MVs) for each time step. A single time marching of the DFDD-ELD, the DGTD-Central and the DGTD-Upwind schemes consists of three MV processes – a volume process, a surface process, and a dispersion process, where each process computes one or more MVs. The detailed GPU implementation of the DFDD-ELD scheme is presented in [56], where the GPU threads and the memory data structure are carefully assigned to perform element-level MVs in parallel with coalesced memory access. Each MV is computed by column, effectively dedicating each GPU thread to the updating of a single field unknown in the solution vector. This approach ensures completely coalesced GPU device global memory access, a key factor to the acceleration capability of the GPUs, and it is readily applicable to the volume and dispersion MVs. Based on the derivations earlier in this section, it is shown that the surface matrices are related to the contribution of surface integrals of the fields from the neighboring elements, where random access is unavoidable when gathering the multiplier vectors, resulting in a drop in GPU performance. Again, the on-chip L1 and L2 caches can be utilized to reduce the bottleneck by decomposing the computation domain into regions of elements with physical proximity using METIS to improve the speedup by 15% for a problem with a hundred thousand elements and over 20% for a problem with one million elements [56]. Here we apply the same parallelization framework for the implementation of the DGTD-Central and DGTD-Upwind schemes, and analyze the relative workload for each time step of the three explicit schemes.

The per-step time-marching of the DFDD-ELD scheme for the update of the electric field involves two volume MVs $[A_1]\{e\}^n$ and $[A_2]\{e\}^{n-1}$, one surface MV $[F](\{h^+\}^{n+1/2} - \{h^+\}^{n-1/2})$ and two dispersion MVs $[T]\{q\}^n$ and $[U]\{g\}^n$, with the same number of MVs for the update of the magnetic field in the same time step. Together, when the medium is lossless and non-dispersive,

each time step takes four volume MVs and two surface MVs; when the medium is electrically lossy but non-dispersive, each time step takes four volume MVs, two surface MVs, and a single dispersion MV $[T_H]\{q_H\}^n$, the dual of $[T]\{q\}^n$; when the medium is electrically dispersive, each time step takes four volume MVs, two surface MVs, and two dispersion MVs $[T_H]\{q_H\}^n$ and $[U]\{g\}^n$.

The per-step time-marching of the DGTD-Central scheme for the update of the electric field involves two volume MVs $[A_1]\{e\}^n$ and $[A_2]\{h\}^{n+1/2}$, one surface MV $[F](\{h^+\}^{n+1/2}-\{\mathrm{h}\}^{n+1/2})$, and one dispersion MV $[U]\{g\}^n$, with the same number of MVs for the update of the magnetic field in the same time step. Together, when the medium is lossless and non-dispersive, each time step takes four volume MVs and two surface MVs; when the medium is electrically lossy but non-dispersive, each time step also takes four volume MVs and two surface MVs; when the medium is electrically dispersive, each time step takes four volume MVs, two surface MVs, and one dispersion MV $[U]\{g\}^n$. Note that unlike DFDD-ELD, the lossy term $[B_e]$ in DGTD-Central is detached from the recursive term. As a result, the DGTD-Central scheme takes as many MVs per step as the DFDD-ELD scheme when the medium is lossless and non-dispersive, while it takes one less MV per step when the medium is electrically lossy and/or dispersive.

The per-step time-marching of the DGTD-Upwind scheme for the update of the electric and magnetic fields is broken down into five minor steps, each of which involves two volume MVs $[B]\{e\}$ and $[S]\{h\}$, two surface MVs $[F_H](\{h^+\}^n-\{\mathrm{h}\}^n)$ and $[F_E](\{e^+\}^n-\{e\}^n)$, and one dispersion MV $[U]\{g\}^n$ for the update of the electric field, where all the matrices have been pre-multiplied with $[M_e]^{-1}$. Note that unlike DFDD-ELD and DGTD-Central, where the lossy term $[B_e]\{e\}$ is embedded in the other volume matrices, DGTD-Upwind has a standalone lossy term where it can be ignored if the element is lossless. The update of the magnetic field in the same minor time step involves an identical number of MVs. Together, when the medium is lossless and non-dispersive, each time step takes fifteen volume MVs and twenty surface MVs; when the medium is electrically lossy but non-dispersive, each time step takes twenty volume MVs and twenty surface MVs; when the medium is electrically dispersive, each time step takes fifteen

volume MVs, twenty surface MVs, and five dispersion MVs $[U]\{g\}^n$. Due to the five-stage minor steps, the per-step DGTD-Upwind is the slowest among the three schemes despite its larger solution time step, which is about twice that of DFDD-ELD and DGTD-Central. The bandwidth comparison between the three schemes will be shown in Sections 4.5.6 and 4.5.7.

4.5 Examples and Speedups

To demonstrate the performance of the schemes proposed for the acceleration of DFDD-ELD on GPUs, various electromagnetics problems are considered below. The XSEDE Keeneland cluster [84] is used as the GPU cluster for the first five examples, whose GPU device is the NVIDIA Tesla M2090, and its Xeon X5560 hexa-core CPU threads are used for performance comparison. Each MPI node consists of a single CPU thread and a single GPU card. Each Tesla M2090 GPU has 16 SMs with 32 SPs and a maximum global memory bandwidth of 177 GB/s. The XSEDE Stampede cluster [85] is used as the GPU cluster for the last two examples, whose GPU device is the NVIDIA Tesla K20 with Compute Capability 3.5, and its Xeon E5-2680 octa-core CPU threads are used for performance comparison. Each MPI node also consists of a single CPU thread and a single GPU card. Each Tesla K20 GPU has 13 SMs with 192 SPs and a maximum global memory bandwidth of 208 GB/s. The timings and the bandwidths are calculated by averaging the timings across all MPI nodes. The timing of each time step is broken down into the four main processes, namely the volume MV, the surface MV, the dispersion MV ($[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$, $[P]\{j_s\}^{n+1/2}$, and $[T]\{q\}^n + [U]\{g\}^n$ for DFDD-ELD, respectively), and the inter-MPI communication time. Although the notations of the four processes only display the matrix-vector multiplications for the electric field updating equation, the timings and the bandwidths include both the per-step electric field and magnetic field updating computations.

4.5.1 Parallel-Plate Transmission Line

The first example is a parallel-plate transmission line as shown in Fig. 4.6. The structure consists of an electric Debye material with $\varepsilon_\infty = 3.64$, $\varepsilon_s = 4.18$, $\tau_e = 11.40\text{ps}$, and $\mu_r = 1.0$, and a conductor loss of $\sigma_e = 6.29\times10^{-3}\text{S/m}$. The inner and outer radii of the coaxial feeding lines are

0.63 mm and 2.0 mm, respectively, with a lossless and non-dispersive material having $\varepsilon_r = 2.1$ and $\mu_r = 1.0$. The number of elements is 41,385, marching at a time step of $\Delta t = 0.2\text{ps}$ for a total of 40,000 time steps. Figure 4.7 shows the scattering parameters of the transmission line by Fourier-transforming the time-domain responses. Table 4.4 gives the per-step CPU and GPU timing comparison using mixed first-order basis functions on various numbers of cluster nodes.

It can be seen from Table 4.4 that a GPU on a single cluster node is capable of accelerating the algorithm by more than 70 times. Note that the speedup first increases, then decreases, with an increasing number of cluster nodes. This is because, as the number of element-level subdomains per GPU decreases, the communication time, including the time for boundary data gathering and receiving, gradually takes over and becomes significant in the total per-step time. Also, note that the communication time differs substantially between the CPUs and GPUs. This is due to the large synchronization time between the CPU nodes. With each MPI node completing a single time-marching step at a different real wall-clock time, the synchronization difference between the CPU timings will be much larger than that of the GPU timings due to their relatively longer marching time. This ratio will increase with a poorer load balancing across the MPI nodes.

4.5.2 Microstrip Patch Array

The second example is a microstrip patch array shown in Fig. 4.8. The structure consists of a magnetic Lorentz material with $\varepsilon_r = 2.67$, $\mu_s = 1.5$, $\mu_\infty = 1.0$, $\omega_0 = 2\pi \times 0.8 \times 10^9 \text{rad/s}$, and $\delta_e = 0.5 \times 10^9 \text{rad/s}$. The inner and outer radii of the coaxial feeding lines are 0.48 mm and 1.5 mm, respectively, with a lossless and non-dispersive material having $\varepsilon_r = 1.86$ and $\mu_r = 1.0$. The number of elements is 136,863, marching at a time step of $\Delta t = 0.12\text{ps}$ for a total of 75,000 time steps when using mixed first-order basis, and a time step of $\Delta t = 0.06\text{ps}$ for a total of 150,000 time steps. Figure 4.9 shows the scattering parameters of the four ports. Table 4.5 and Table 4.6 give the per-step CPU and GPU timing comparison on various numbers of cluster nodes using mixed first-order and full first-order basis functions, respectively. It can be seen from Table 4.5 that with an increasing number of finite elements, the GPU speedup using mixed first-order basis functions increases compared to the previous example. Figure 4.10 breaks down the speedup

into the three substep processes. It is shown that as the problem size increases, the speedup also increases for a fixed number of GPUs. Despite the increase in the overall speedup, the proportion of the communication time still becomes significant with an increasing number of GPU nodes. This is again due to the insufficient number of elements distributed, bringing down the marching/communication ratio. Since the inter-MPI communication holds a bottleneck, the highest CPU/GPU speedup ratio is usually achieved when a minimum number of GPU nodes are used. Other observations can also be drawn from Table 4.5, Table 4.6 and Fig. 4.10. First, a higher speedup is observed in $[P]\{j_s\}^{n+1/2}$, well above the speedup observed in $[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$ and $[T]\{q\}^n + [U]\{g\}^n$. This is due to the better caching and shared memory utilization of the multiplier vectors on GPUs. While random access caching provides a lower latency to the fetching of the neighboring fields through an element rearrangement, the shared memory usage per element increases from $numTetDof$ to $4 \times numTriDof$, both of which effectively increase the global memory bandwidth and hence the overall speedup.

Second, it can be seen that the speedup differs for different basis function orders, as shown in Fig. 4.11, where the speedup for all three processes is lower using the full first-order basis functions. The use of higher-order basis functions is essentially equivalent to larger matrix-vector multiplications for each element. For example, using the mixed first-order basis functions results in 6×6 $[A_1]$ and $[A_2]$ matrices and a 6×24 $[P]$ matrix, while using the full first-order basis functions results in 12×12 $[A_1]$ and $[A_2]$ matrices and a 12×48 $[P]$ matrix, which uses four times the memory of the mixed first-order basis functions. It is shown in the tables that the marching time on GPUs increases by around 3 times due to the increase in the basis function order. However, the marching time on CPU only increases by slightly over 2 times. As the CPU serially performs the matrix-vector multiplication element by element, it is suspected that cache is better utilized on a CPU for certain matrix dimensions since it is well known that a GPU can achieve a high speedup with a higher-dimensional matrix. Despite the decrease in the CPU/GPU speedup, a higher GPU bandwidth (measured in GB/s of global memory access) is achieved using higher-order basis functions, as shown in Fig. 4.12. It can be seen that for a fixed number of GPUs, a higher bandwidth is achieved using full first-order basis functions. This observation is more apparent for $[P]\{j_s\}^{n+1/2}$ and $[T]\{q\}^n + [U]\{g\}^n$, which is due to the increased caching and shared memory utilization of the randomly accessed multiplier vectors. It is expected that

the bandwidth will continue to grow for even higher-order basis functions. Also, it is worth noting that the bandwidth in this work is computed based only on the amount of the finite element data accessed, i.e. the matrices and vectors. It does not take into account the various indexing arrays accessed by the threads. Therefore the actual achieved bandwidth will be higher.

The LTS technique is separately applied on the non-dispersive version of the microstrip patch array example on a single node. The class distribution for the finite elements is illustrated in Fig. 4.13. The 80,000 coax feed finite elements, which are heavily refined to properly model the geometry, are grouped into Class 1 and are painted in red. The 20,000 elements under the patches, which are slightly refined to account for the larger field variation, are grouped into Class 2 and are painted in grey. All the other 50,000 elements, including the rest of the patch and the free-space elements, are grouped into Class 3 and are painted in blue. It is observed that the original total non-LTS solution time is 1,118 seconds while the LTS solution time is 846 seconds, which corresponds to a 1.3 times speedup. It is worth noting that the number of elements in each class distribution largely determines the LTS speedup. If there are proportionally many more elements in the higher classes (larger elements) than in the lower classes (slammer elements), then there will be more elements marching at larger time steps, resulting in a higher LTS speedup, and vice versa.

4.5.3 Antipodal Fermi Antenna

The third example is an antipodal Fermi antenna [86] reproduced in Fig. 4.14, which is printed on a dielectric board with a dimension of $140.2\text{mm} \times 44.7\text{mm} \times 0.86\text{mm}$. The antenna is connected to a coaxial feeding line through a 1.5-mm wide microstrip line. The inner and outer radii of the coaxial feeding line are 0.86 mm and 0.374 mm, respectively, with a lossless and non-dispersive material using $\varepsilon_r = 1.0$ and $\mu_r = 1.0$. The dielectric substrate is also a lossless and non-dispersive material having $\varepsilon_r = 4.4$ and $\mu_r = 1.0$. The number of elements is 1,072,013, marching at a time step of $\Delta t = 0.02\text{ps}$ for a total of 150,000 time steps. The solution uses full first-order basis functions, resulting in 12,864,156 unknowns. Figure 4.15 shows the scattering parameter of the Fermi antenna. Although the point-by-point comparison is not good here because of the discrepancy between the simulation and measurement setups, the levels of the two results agree quite well, which is more important for this case. Table 4.7 gives the per-step CPU

and GPU timing comparison on various numbers of cluster nodes. Figures 4.16 and 4.17 show the speedup breakdown and the bandwidth comparison of $[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$ and $[P]\{j_s\}^{n+1/2}$ calculations, respectively. N/A indicates insufficient memory under the specific MPI configuration.

It can be seen from Table 4.7 that because there is a sufficient number of elements for the Fermi antenna, the GPUs are fully utilized and therefore the level of speedup does not decrease significantly, as found in the previous examples, with an increasing number of GPU nodes. The calculation of $[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$ shows a lower speedup in Fig. 4.16 for the Fermi antenna using 4 and 8 MPI nodes, but this is again suspected to result from a better caching by the CPUs. Note that the Fermi antenna contains 7.8 times more elements than the patch array, which linearly increases the GPU timing for $[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$ as shown in Table 4.6 and Table 4.7. On the other hand, the same increase in the number of elements only increases the CPU timing by around 3 times, effectively lowering the CPU/GPU speedup. Nevertheless, Fig. 4.17 shows that the Fermi antenna case achieves a higher bandwidth for $[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$, and therefore the proposed parallelization scheme still scales well on the GPU cluster.

It is well known that a GPU is capable of accelerating highly parallelizable algorithms with a high speedup, either through the increase in throughput for processor-intensive algorithms or through a higher bandwidth for memory-intensive algorithms. With stable bandwidths under different numbers of MPI nodes, Fig. 4.17 shows that these bandwidths are the highest achievable using the proposed parallelization strategy, and that adding more elements per GPU will not further increase the bandwidth and therefore yield further speedup. This is a limitation for the parallelization of the DFDD-ELD algorithm, and for algorithms with dominant matrix-vector multiplication processes on very fine subdomains. For matrix-vector multiplications, a high bandwidth on GPUs is achieved by the coalesced access of the matrices, as well as the caching of the multiplier vectors, where a higher bandwidth can be achieved for higher-dimensional matrix-vector multiplications. The DFDD-ELD algorithm on element-level matrices and vectors cannot achieve the same bandwidth growth by adding more elements. Instead, bandwidth growth can only be achieved through higher-order basis functions with larger matrix and vector dimensions per element as shown in Section 4.5.2.

4.5.4 Monopole Antenna Array

The fourth set of examples consists of large monopole antenna arrays with varying array dimensions, which are placed on top of infinite ground planes. The height of the monopoles is 10 cm. The inner and outer radii of the coaxial feeding lines are 1 cm and 2.3 cm, respectively, with a lossless and non-dispersive material having $\varepsilon_r = 1.0$ and $\mu_r = 1.0$. The antenna unit cell has 3,344 elements, while the airbox unit cells have approximately 1,000 elements. The solution is obtained by marching at a time step of $\Delta t = 1\text{ps}$ for a total of 25,000 time steps. Table 4.8 gives the per-step CPU and GPU timing comparison for varying numbers of MPI nodes and array dimensions, using full first-order basis functions. Again, N/A indicates insufficient memory under the specific MPI configuration.

Some observations can be made about Table 4.8. First, the DFDD-ELD algorithm with array configuration demonstrates good scalability on both the CPU and the GPU nodes. It can be seen that the total per-step time decreases proportionally with an increasing number of MPI nodes. This shows that a high marching/communication ratio is achieved for problems with large numbers of elements, where the inter-MPI communication time becomes insignificant even under GPU acceleration. Second, like the non-array problems, speedup can be maintained with an increasing number of GPU nodes since the GPUs are fully occupied by the large number of elements per GPU due to the memory savings through the unit cell configuration. However, the significantly increased number of elements per GPU does not result in a meaningful increase in speedup, due to bandwidth limitation as stated in the previous example. Figure 4.18 shows the bandwidth achieved by different dimensions of monopole arrays. It can be seen that for $[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$, the bandwidth for a fixed array dimension varies above and below the threshold bandwidth (~115 GB/s from the previous example). This is due to the mixed effects of using the unit cell configuration, where the possible caching of matrix access increases the bandwidth, while the extra element-to-matrix mapping indices decrease the bandwidth. On the other hand, $[P]\{j_s\}^{n+1/2}$ has a bandwidth consistently below the threshold bandwidth (~ 97 GB/s). This is mainly due to the fact that no cache utilization through element rearrangement has been implemented for the finite array configuration, resulting in a lower bandwidth for the random access of the multiplier vectors. It is expected that the threshold bandwidth will be achieved with the incorporation of element rearrangement.

4.5.5 Vivaldi Antenna Array

The next examples are large Vivaldi antenna arrays with varying array dimensions, which are placed on top of infinite ground planes. The antenna unit cell has 19,137 elements, while each of the surrounding airbox unit cells has approximately 12,800 elements. The solution is calculated by marching at a time step of $\Delta t = 8\text{fs}$ for a total of 150,000 time steps. Table 4.9 gives the per-step CPU and GPU timing comparison for varying numbers of MPI nodes and array dimensions, using full first-order basis functions. Figure 4.19 shows the speedup ratio comparisons between the monopole and Vivaldi arrays of different array dimensions. It can be seen from Fig. 4.19 that the monopole and Vivaldi arrays share a similar trend in speedup ratio for different numbers of MPI nodes, albeit with higher overall speedup for the Vivaldi arrays. This is due to the higher random access latency on the CPU nodes as the unit cell size increases. Figure 4.20 shows the bandwidth achieved by different Vivaldi array dimensions. It can be seen that the bandwidth for $[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$ remains similar to that of the monopole arrays without the spikes, as caching of matrices is no longer possible for problems with larger unit cells. The bandwidth for $[P]\{j_s\}^{n+1/2}$ on the other hand is lower than that of the monopole arrays due to a higher random access latency, which can again be alleviated by cache utilization through element rearrangement. It is expected that a higher speedup can be achieved for problems with larger unit cells, much like the speedups found in the non-array examples.

4.5.6 Three Vias in a Three-Layer PCB

As a comparison between the acceleration performance of the DFDD-ELD, DGTD-Central, and DGTD-Upwind algorithms, a couple of examples are provided. The first is a three-via configuration in a three-layer PCB structure from [83] with the dimensions reproduced in Fig. 4.21. The PCB consists of an electric Debye material with $\varepsilon_\infty = 3.98$, $\varepsilon_s = 4.09$, $\tau_e = 39.50\text{ps}$, and $\mu_r = 1.0$, and an conductor loss of $\sigma_e = 2.58 \times 10^{-3}\text{S/m}$ [87]. The number of elements is 119,724, marching at a time step of $\Delta t = 0.75\text{ps}$ for a total of 26,667 time steps. The computation is performed using mixed second-order basis, generating a total of 2,394,480 unknowns. Figure 4.22 shows the scattering parameters at the two ports of the three-via structure. Figure 4.23 shows the speedup and the bandwidth comparison between the three

algorithms. It can be seen that due to the systemized parallelization framework, all three schemes achieve similar speedup and bandwidth levels despite the differences in the actual per-step time. Comparing the performance of the K20 GPUs with the previous M2090 GPUs, although the speedup is lower due to the different configuration of the two clusters, the three-via example received an increased bandwidth on GPU due to the higher memory bandwidth of the K20 GPUs. The increase in bandwidth is most apparent for the surface MV, where the bandwidth is increased by more than 30% due to the enlarged L2 cache (1.5 MB) and the increased memory clock rate (2.6 GHz for K20, 1.85 GHz for M2090). Table 4.10 gives the per-step CPU and GPU timing comparison for varying numbers of MPI nodes. It is shown that DGTD-Central has the fastest solution time, which is then closely followed by DFDD-ELD due to the extra dispersive MV process. DGTD-Upwind is less than one fifth the speed of the above two algorithms due to the five-stage LSERK process.

4.5.7 Branch-Line Coupler

The last example is a branch-line coupler from [88] with the geometry reproduced in Fig. 4.24. The substrate has a thickness of 0.508 mm using a lossless and non-dispersive material of $\varepsilon_r = 2.1$ and $\mu_r = 1.0$. The number of elements is 447,742, marching at a time step of $\Delta t = 0.1\text{ps}$ for a total of 50,000 time steps for the DFDD-ELD and DGTD-Central scheme and a time step of $\Delta t = 0.04\text{ps}$ for a total of 125,000 time steps for the DGTD-Upwind scheme. Note that for this example we have pushed the time step to be close to the stability constraints, where the time step of the DGTD-Upwind scheme is 0.4 times that of the other two algorithms, and this is due to the stricter stability constraints from the introduction of the penalty terms in the upwind flux, making it more restrictive than the central flux and the DFDD-ELD algorithms. The computation is performed using mixed first-order basis, generating a total of 2,686,452 unknowns, respectively. Figure 4.25 shows the scattering parameters of the four ports. Figure 4.26 shows the speedup and the bandwidth comparison between the three algorithms. Table 4.11 gives the per-step CPU and GPU timing comparison for varying numbers of MPI nodes. Again all three schemes achieve similar speedup and bandwidth levels due to the parallelization framework.

4.6 Figures and Tables

Fig. 4.1: Allocation of threads and blocks for the integration kernels.

Fig. 4.2: Memory allocation for matrix-vector multiplication.

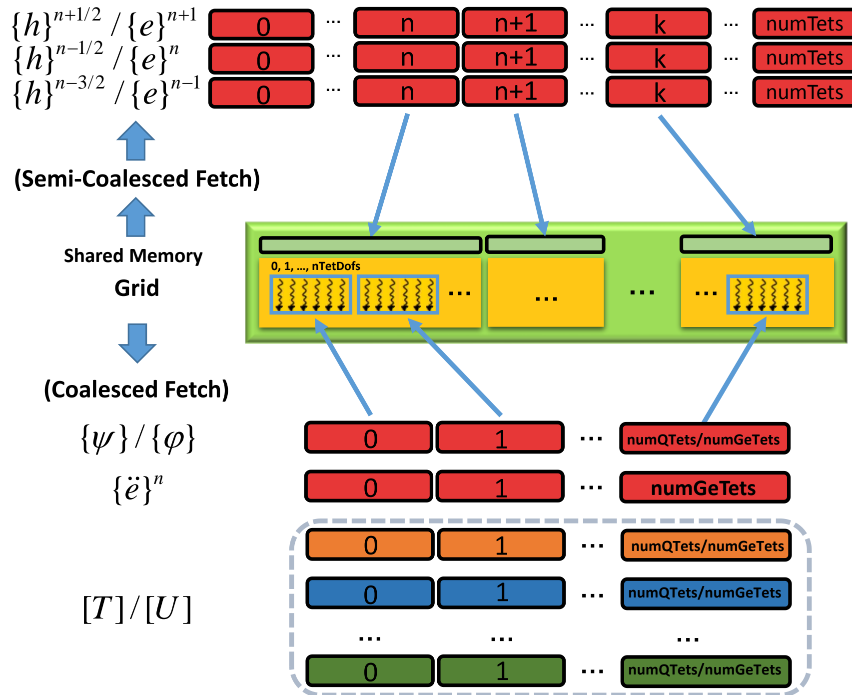


Fig. 4.3: Memory access for the lossy and dispersive update for the electric field.

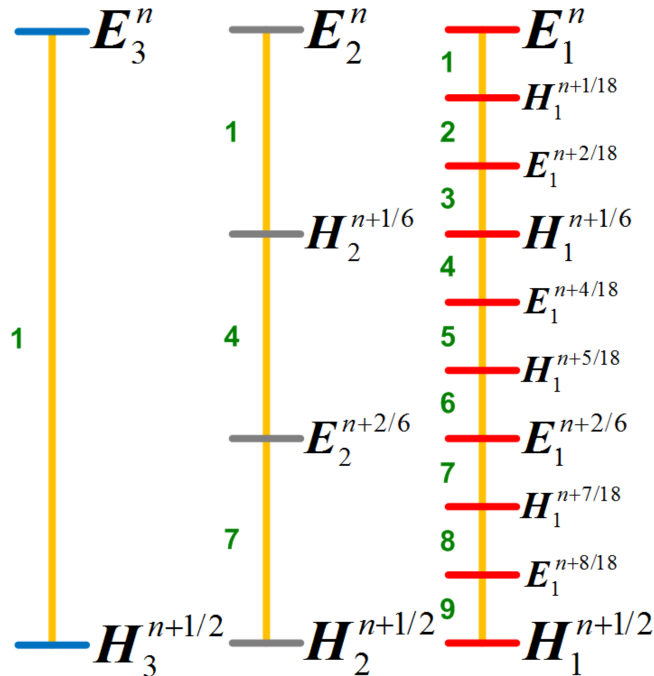


Fig. 4.4: Local time-stepping technique for the DFDD-ELD algorithm using different classes of time steps which are multiples of 3 of each other.

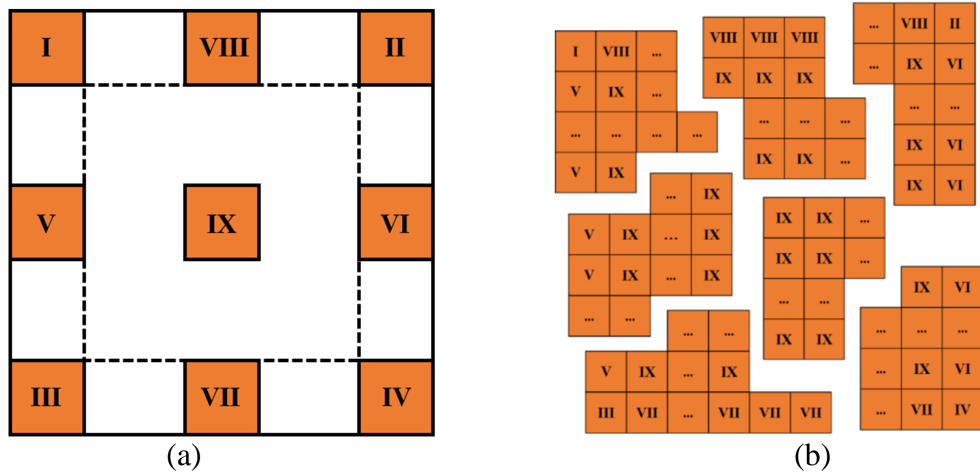


Fig. 4.5: Antenna array configuration. (a) Nine unit cells. (b) Region decomposition.

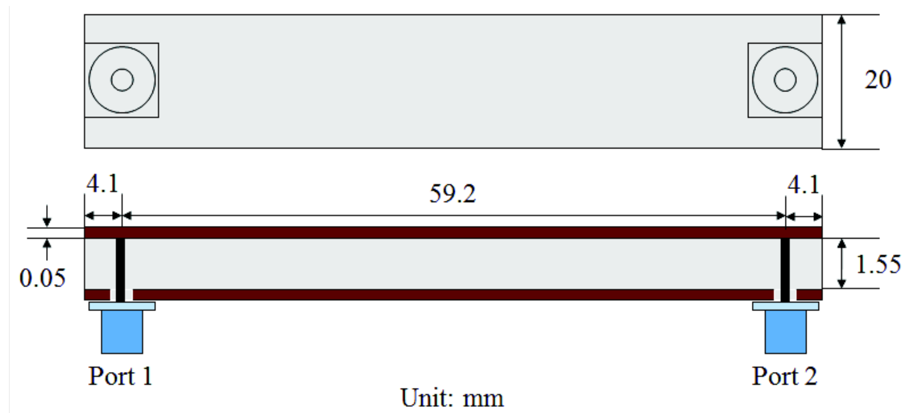


Fig. 4.6: Geometry of the parallel-plate transmission line.

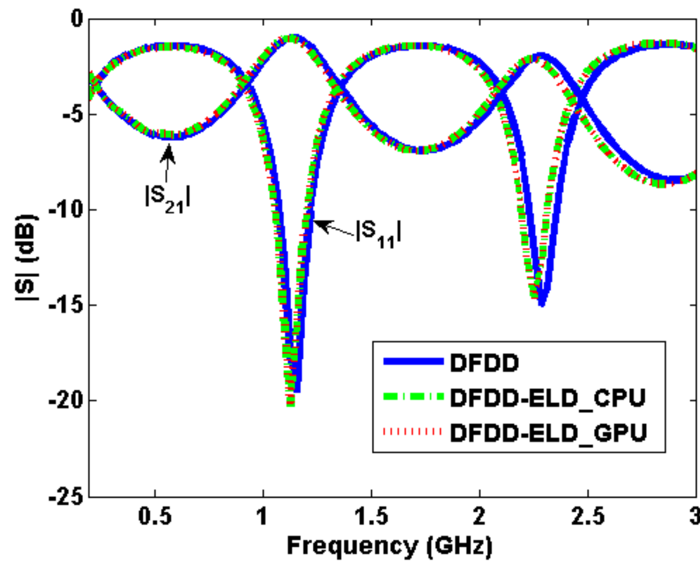


Fig. 4.7: Scattering parameters of the parallel-plate transmission line.

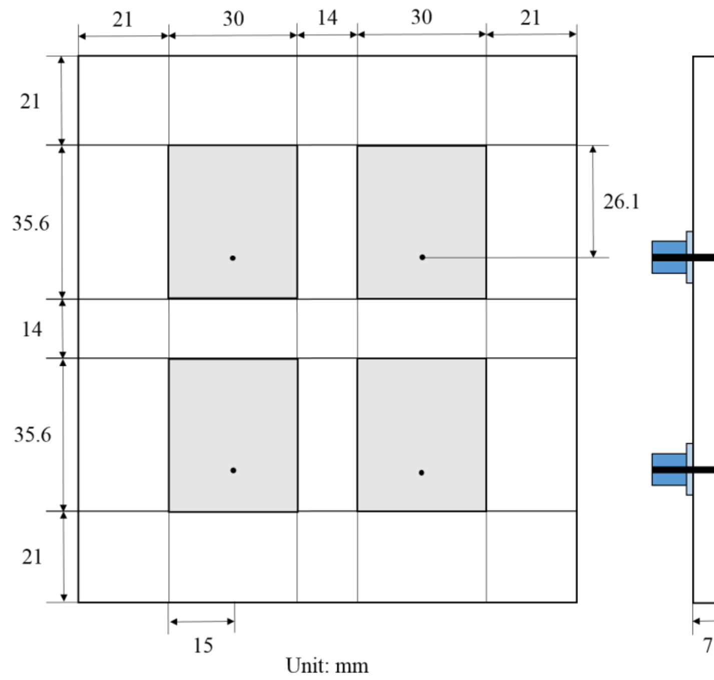


Fig. 4.8: Geometry of the patch array.

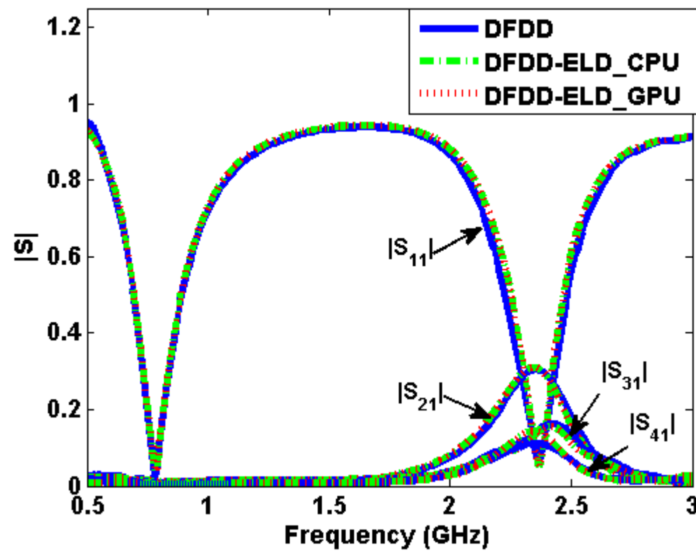


Fig. 4.9: Scattering parameters of the patch array.

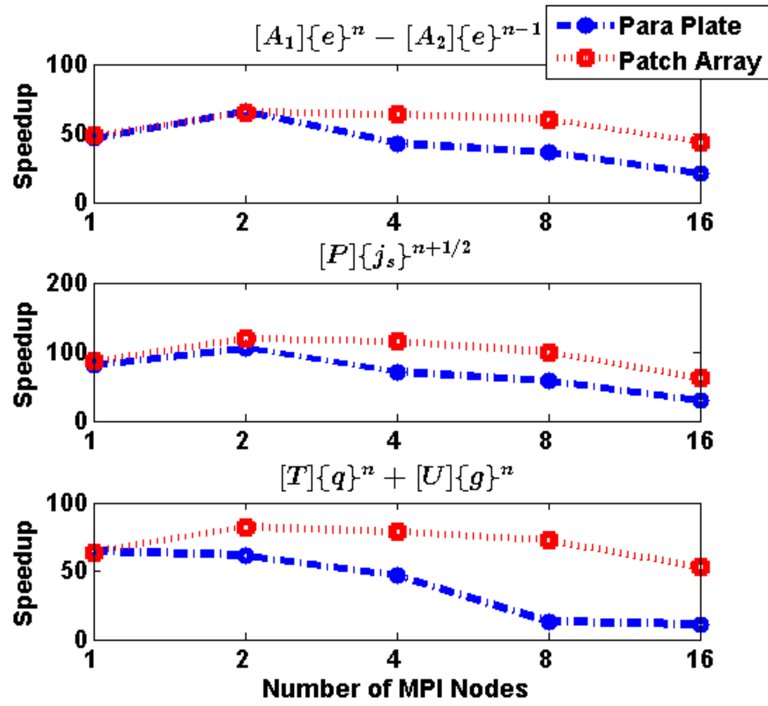


Fig. 4.10: Speedup comparison of the different processes between the transmission line and the patch array using mixed first-order basis functions.

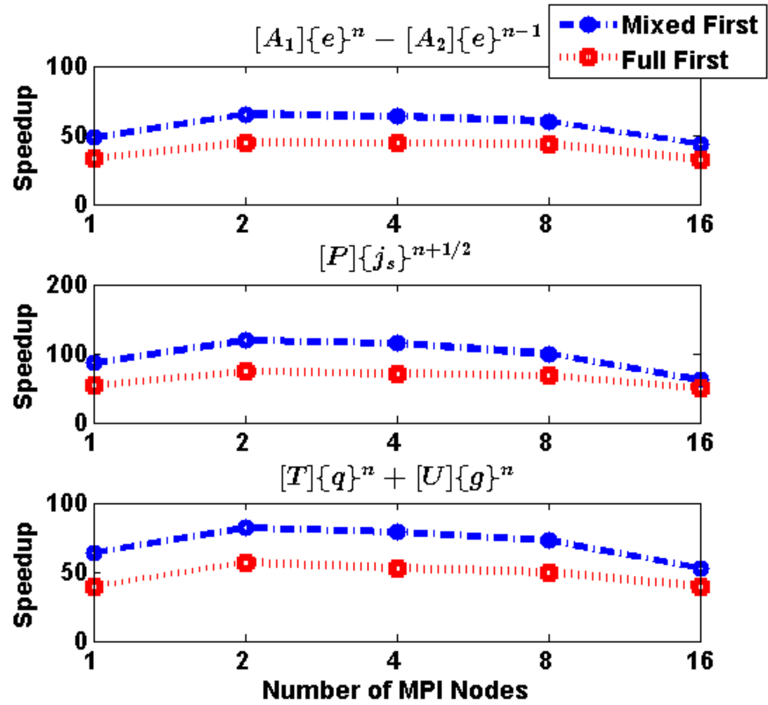


Fig. 4.11: Speedup comparison of the different processes between mixed and full first-order basis functions for the patch array.

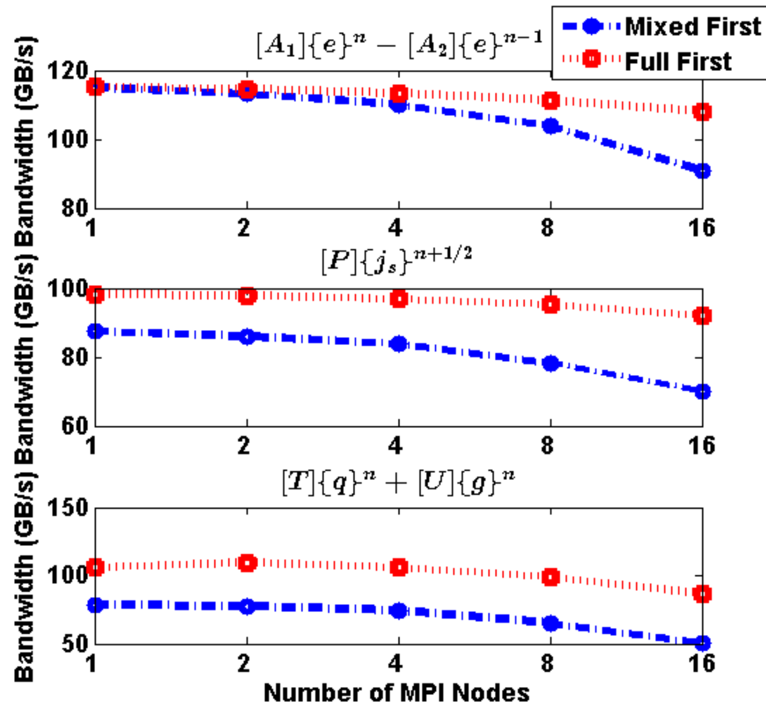


Fig. 4.12: Bandwidth comparison of the different processes between mixed and full first-order basis functions for the patch array.

Fig. 4.13: Local time-stepping class distribution for the patch antenna.

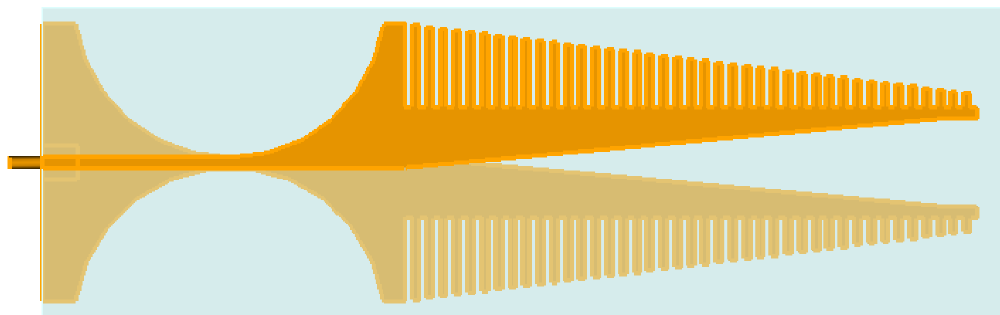


Fig. 4.14: Geometry of the Fermi antenna.

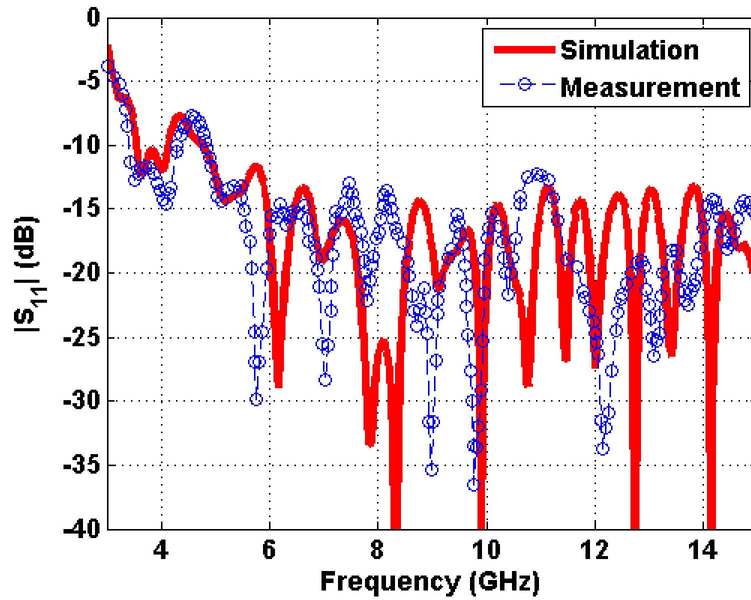


Fig. 4.15: Scattering parameter of the Fermi antenna.

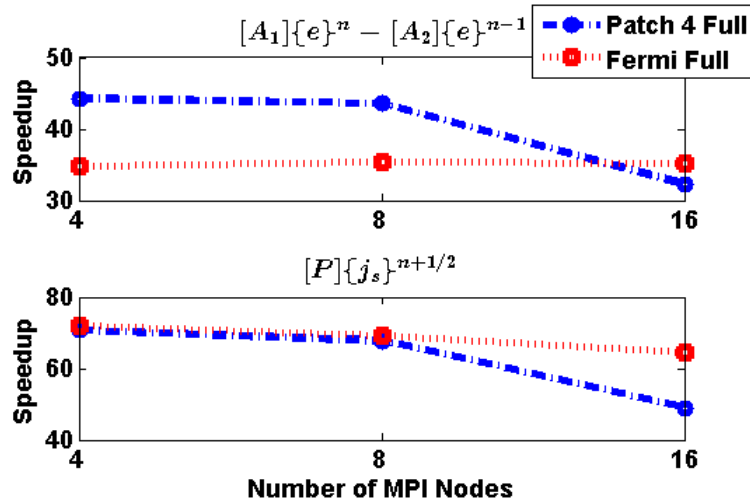


Fig. 4.16: Speedup comparison of the different processes between the patch array and the Fermi antenna using full first-order basis functions.

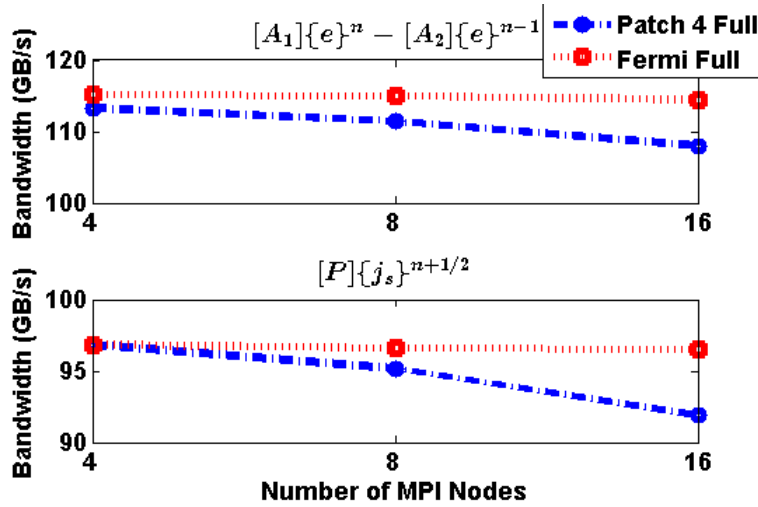


Fig. 4.17: Bandwidth comparison of the different processes between the patch array and the Fermi antenna using full first-order basis functions.

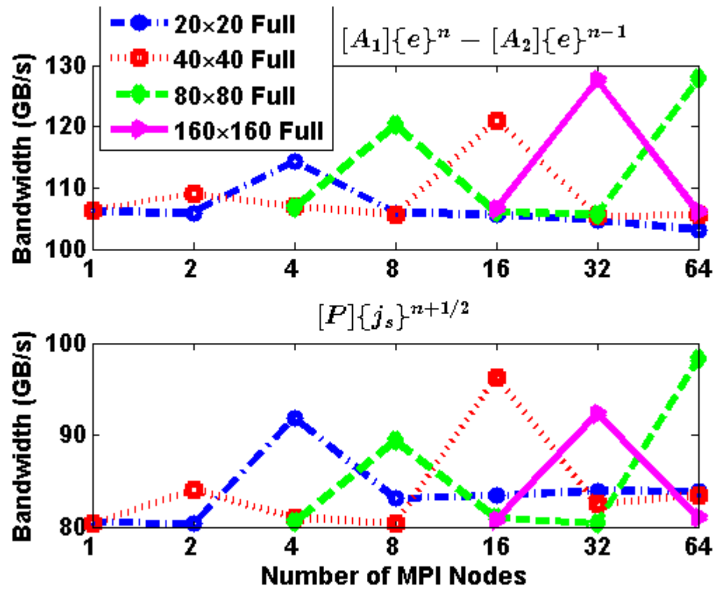


Fig. 4.18: Bandwidth comparison of the different processes between different monopole array dimensions.

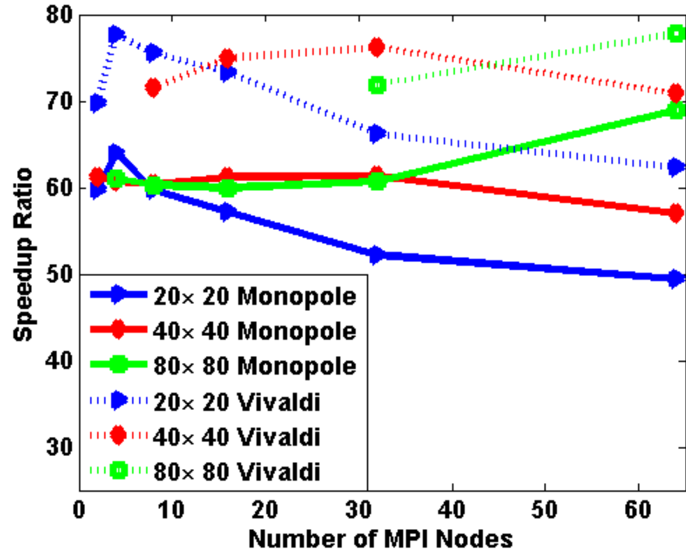


Fig. 4.19: Speedup ratio comparisons between the monopole and Vivaldi arrays.

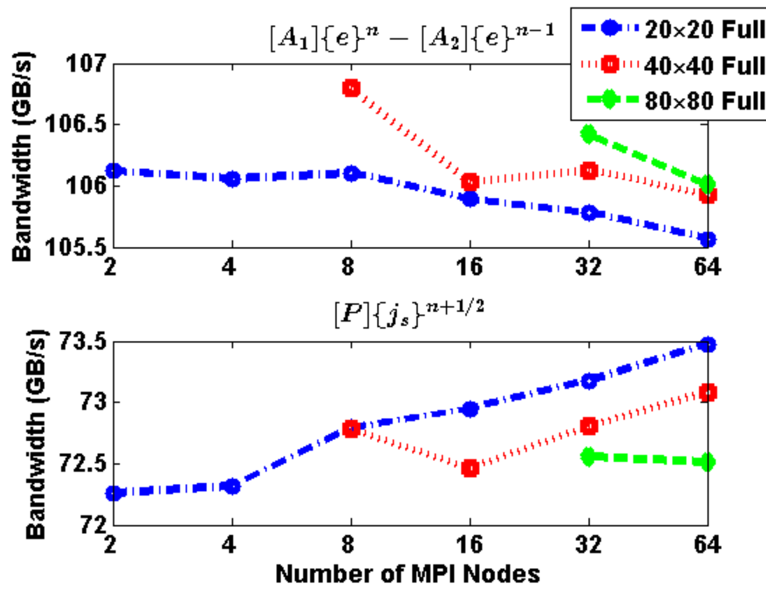


Fig. 4.20: Bandwidth comparison of the different processes between different Vivaldi array dimensions.

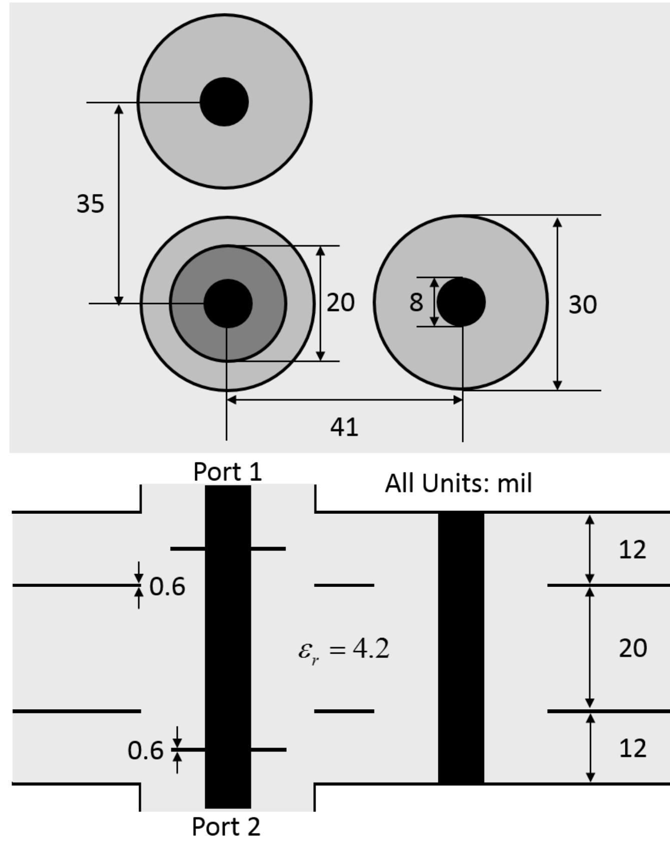


Fig. 4.21: Geometry of the three-via structure.

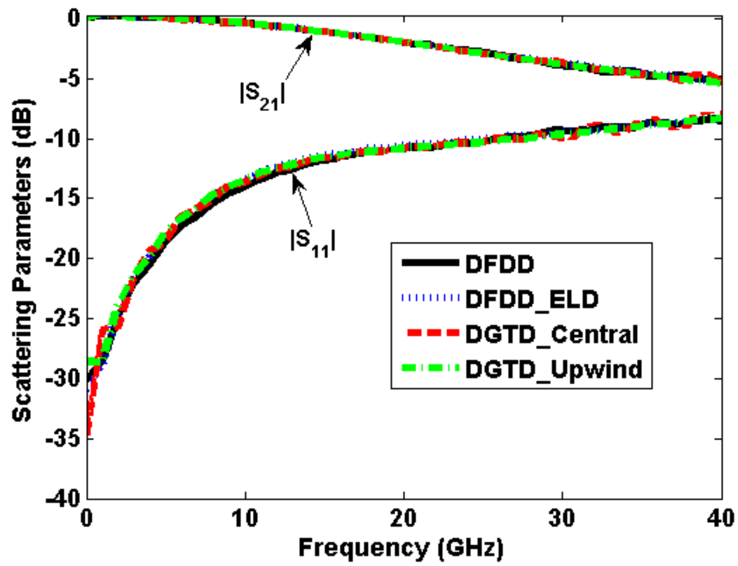
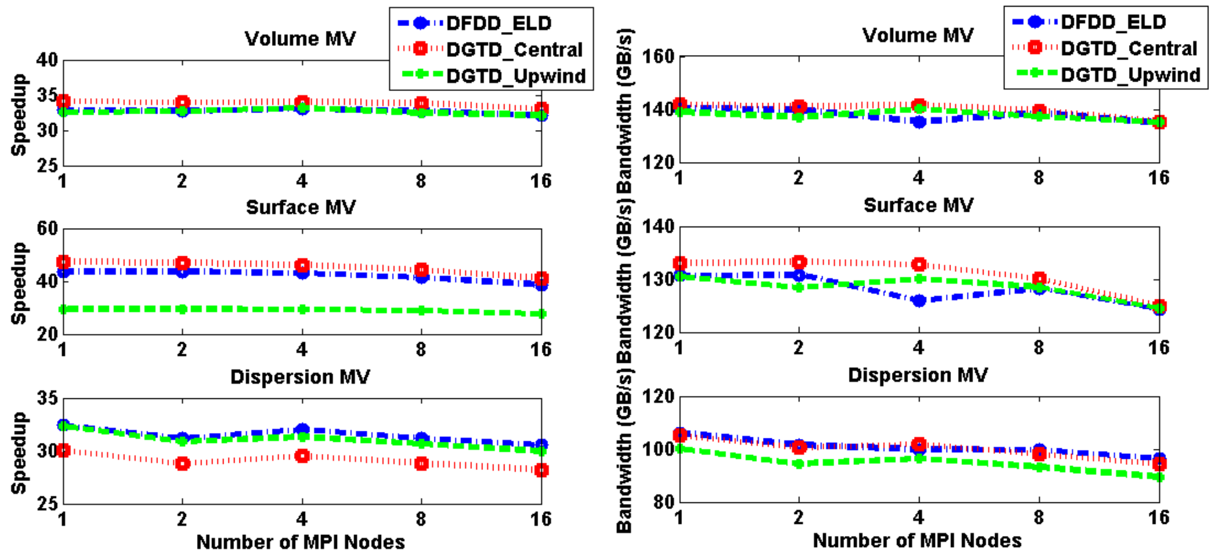


Fig. 4.22: Scattering parameter of the three-via structure.



(a) (b)

Fig. 4.23: (a) Speedup comparison and (b) Bandwidth comparison of the three schemes using full first-order basis functions for the three-via configuration.

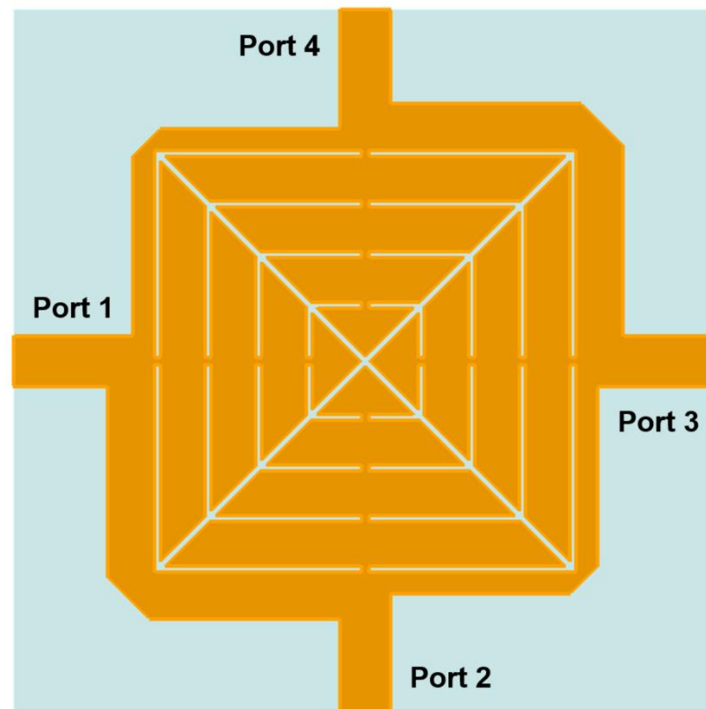


Fig. 4.24: Geometry of the branch-line coupler.

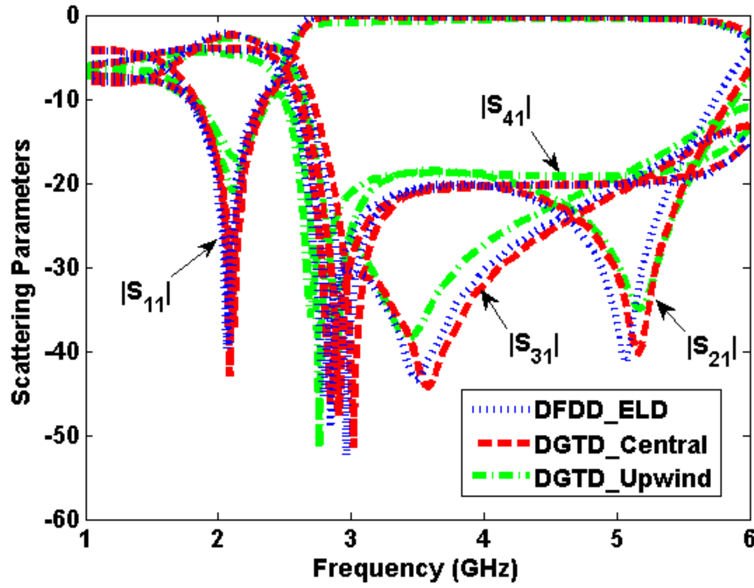


Fig. 4.25: Scattering parameter of the branch-line coupler.

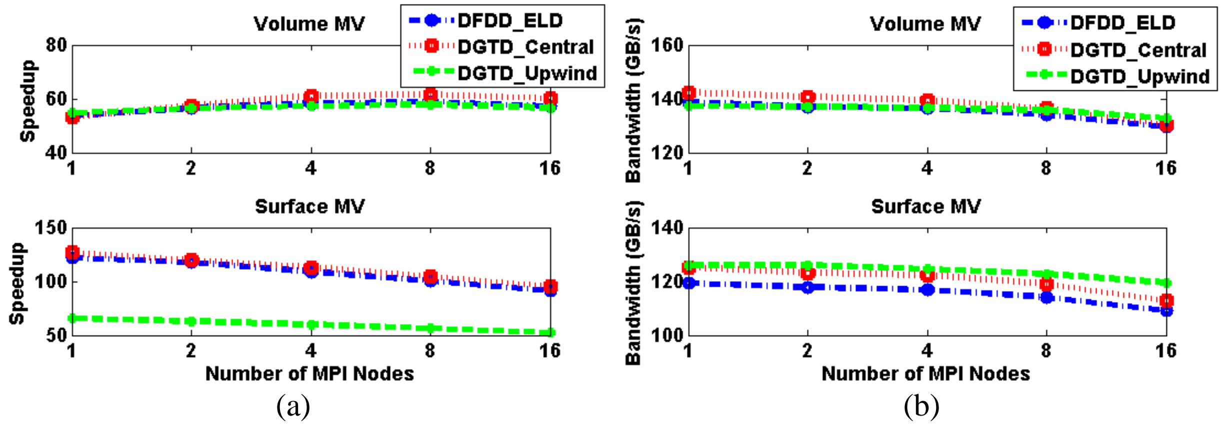


Fig. 4.26: (a) Speedup comparison and (b) Bandwidth comparison of the three schemes using mixed first-order basis functions for the branch-line coupler.

Table 4.1: Single GPU time per time-marching step for a transmission line with 41,385 elements.

Time (ms)	**Number of Threads per Block**		**Speedup Improvement**
	480	**512**	
w/ Rearrangement	1.46	1.61	**9.26%**
w/o Rearrangement	1.55	1.70	**8.80%**
Speedup Improvement	**5.54%**	**5.06%**	**13.85%**

Table 4.2: Single GPU time per time-marching step for a patch array with 136,863 elements.

Time (ms)	**Number of Threads per Block**		**Speedup Improvement**
	480	**512**	
w/ Rearrangement	4.69	5.20	**9.87%**
w/o Rearrangement	5.05	5.56	**9.25%**
Speedup Improvement	**7.11%**	**6.47%**	**15.70%**

Table 4.3: Single GPU time per time-marching step for a Fermi antenna with 1,072,013 elements.

Time (ms)	**Number of Threads per Block**		**Speedup Improvement**
	480	**512**	
w/ Rearrangement	36.39	40.82	**10.85%**
w/o Rearrangement	42.02	46.22	**9.08%**
Speedup Improvement	**13.40%**	**11.68%**	**21.27%**

Table 4.4: Per-step timing comparison for a parallel-plate transmission line using mixed first-order basis functions.

Number of MPI Nodes		**1**	**2**	**4**	**8**	**16**
CPU Time per Step (ms)	$[\boldsymbol{A}_1]\{\boldsymbol{e}\}^n - [\boldsymbol{A}_2]\{\boldsymbol{e}\}^{n-1}$	28.27	21.02	7.71	3.95	1.69
	$[\boldsymbol{P}]\{\boldsymbol{j}_s\}^{n+1/2}$	63.63	44.31	16.27	8.04	2.66
	$[\boldsymbol{T}]\{\boldsymbol{q}\}^n + [\boldsymbol{U}]\{\boldsymbol{g}\}^n$	13.60	8.01	4.64	0.65	0.55
	Total Marching	115.87	79.19	32.22	14.69	5.72
	Communication	0	0.34	6.01	3.35	1.33
	Total Per-Step	115.87	79.53	38.23	18.04	7.05
GPU Time per Step (ms)	$[\boldsymbol{A}_1]\{\boldsymbol{e}\}^n - [\boldsymbol{A}_2]\{\boldsymbol{e}\}^{n-1}$	0.61	0.32	0.18	0.11	0.08
	$[\boldsymbol{P}]\{\boldsymbol{j}_s\}^{n+1/2}$	0.79	0.42	0.23	0.14	0.09
	$[\boldsymbol{T}]\{\boldsymbol{q}\}^n + [\boldsymbol{U}]\{\boldsymbol{g}\}^n$	0.21	0.13	0.10	0.05	0.05
	Total Marching	1.65	0.91	0.56	0.35	0.26
	Communication	0	0.10	0.15	0.17	0.16
	Total Per-Step	1.65	1.02	0.71	0.52	0.42
Speedup		**70.22**	**77.97**	**53.85**	**34.69**	**16.79**

Table 4.5: Per-step timing comparison for a patch array using mixed first-order basis functions.

Number of MPI Nodes		1	2	4	8	16
CPU Time per Step (ms)	$[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$	93.48	64.48	32.32	15.00	6.97
	$[P]\{j_s\}^{n+1/2}$	219.66	155.15	75.56	32.96	14.75
	$[T]\{q\}^n + [U]\{g\}^n$	95.11	60.74	30.73	22.49	2.63
	Total Marching	442.76	298.35	147.96	75.24	27.94
	Communication	0	0.74	5.42	1.97	9.86
	Total Per-Step	442.76	299.08	153.37	77.21	37.80
GPU Time per Step (ms)	$[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$	1.94	0.99	0.51	0.25	0.16
	$[P]\{j_s\}^{n+1/2}$	2.55	1.30	0.66	0.33	0.24
	$[T]\{q\}^n + [U]\{g\}^n$	1.49	0.74	0.39	0.31	0.05
	Total Marching	6.03	3.07	1.61	0.94	0.47
	Communication	0	0.17	0.19	0.13	0.30
	Total Per-Step	6.03	3.24	1.80	1.07	0.77
Speedup		**73.43**	**92.31**	**85.21**	**72.16**	**49.09**

Table 4.6: Per-step timing comparison for a patch array using full first-order basis functions.

Number of MPI Nodes		1	2	4	8	16
CPU Time per Step (ms)	$[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$	217.47	149.32	73.90	34.84	15.18
	$[P]\{j_s\}^{n+1/2}$	413.29	286.66	138.48	63.06	26.96
	$[T]\{q\}^n + [U]\{g\}^n$	147.30	98.64	50.11	34.95	3.57
	Total Marching	812.49	552.59	271.83	137.17	49.35
	Communication	0	0.84	4.18	6.15	16.79
	Total Per-Step	812.49	553.44	276.01	143.32	66.14
GPU Time per Step (ms)	$[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$	6.62	3.33	1.67	0.80	0.47
	$[P]\{j_s\}^{n+1/2}$	7.76	3.90	1.96	0.93	0.55
	$[T]\{q\}^n + [U]\{g\}^n$	3.74	1.74	0.95	0.70	0.09
	Total Marching	18.16	9.02	4.63	2.47	1.16
	Communication	0	0.31	0.28	0.19	0.52
	Total Per-Step	18.16	9.33	4.92	2.66	1.68
Speedup		**44.74**	**59.32**	**56.10**	**53.88**	**39.37**

Table 4.7: Per-step timing comparison for Fermi antenna using full first-order basis functions.

Number of MPI Nodes		2	4	8	16
CPU Time per Step (ms)	$[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$	995.59	456.64	231.64	115.76
	$[P]\{j_s\}^{n+1/2}$	2,738.18	1,129.14	539.85	252.19
	Total Marching	3,921.34	1,686.91	823.66	394.58
	Communication	4.13	224.77	120.74	61.72
	Total Per-Step	3,925.47	1,911.68	944.40	456.30
GPU Time per Step (ms)	$[A_1]\{e\}^n - [A_2]\{e\}^{n-1}$	N/A	14.70	7.37	3.66
	$[P]\{j_s\}^{n+1/2}$	N/A	17.05	8.52	4.24
	Total Marching	N/A	31.80	15.95	7.96
	Communication	N/A	1.46	1.17	0.93
	Total Per-Step	N/A	33.26	17.12	8.89
Speedup		**N/A**	**57.48**	**55.16**	**51.33**

Table 4.8: Per-step timing comparison for the monopole arrays using full first-order basis functions.

Array Size	10x10	20x20	40x40	80x80	160x160	320x320
# of elements	**0.3 M**	**1.2 M**	**5 M**	**20.6 M**	**84.1 M**	**339.4 M**
2 CPU (s)	1.36	5.19	19.96			
2 GPU (ms)	22.94	86.84	326.38	N/A	N/A	N/A
Speedup	59.29	59.77	61.16	N/A	N/A	N/A
4 CPU (s)	0.68	2.63	10.22	40.57		
4 GPU (ms)	11.65	41.11	168.15	664.74	N/A	N/A
Speedup	58.37	63.97	60.78	61.03	N/A	N/A
8 CPU (s)	0.30	1.41	5.30	20.77		
8 GPU (ms)	6.91	23.62	87.83	344.65	N/A	N/A
Speedup	43.42	59.70	60.34	60.26	N/A	N/A
16 CPU (s)	0.22	0.77	2.64	10.42	41.42	
16 GPU (ms)	3.91	13.47	43.13	173.68	674.27	N/A
Speedup	56.26	57.16	61.21	60.00	61.43	N/A
32 CPU (s)	0.13	0.37	1.42	5.31	20.84	
32 GPU (ms)	2.50	7.09	23.15	87.53	322.98	N/A
Speedup	52.00	52.19	61.34	60.67	64.52	N/A
64 CPU (s)	0.07	0.20	0.70	2.71	10.30	39.97
64 GPU (ms)	1.68	4.05	12.29	39.33	168.62	666.56
Speedup	**41.67**	**49.38**	**56.96**	**68.90**	**61.08**	**59.96**

Table 4.9: Per-step timing comparison for the Vivaldi arrays using full first-order basis functions.

Array Size	**10x10**	**20x20**	**40x40**	**80x80**
# of elements	**2.5 M**	**8.7 M**	**32.7 M**	**126.6 M**
2 CPU (s)	12.00	39.58		
2 GPU (ms)	160.50	566.84	N/A	N/A
Speedup	74.77	69.83	N/A	N/A
4 CPU (s)	5.89	22.11		
4 GPU (ms)	80.71	284.67	N/A	N/A
Speedup	72.98	77.67	N/A	N/A
8 CPU (s)	3.01	11.19	38.53	
8 GPU (ms)	43.79	148.18	539.19	N/A
Speedup	68.74	75.52	71.46	N/A
16 CPU (s)	1.63	5.74	20.63	
16 GPU (ms)	23.03	78.30	275.31	N/A
Speedup	70.78	73.31	74.93	N/A
32 CPU (s)	0.98	2.71	10.66	37.80
32 GPU (ms)	13.27	40.94	139.89	525.57
Speedup	73.85	66.19	76.20	71.92
64 CPU (s)	0.60	1.45	5.15	20.60
64 GPU (ms)	8.16	23.28	72.73	264.75
Speedup	**73.53**	**62.29**	**70.81**	**77.81**

Table 4.10: Per-step timing comparison for the three-via configuration using mixed second-order basis functions.

Number of MPI Nodes		**1**	**2**	**4**	**8**	**16**
DFDD-ELD	**CPU Time (ms)**	1,186.61	648.27	375.77	178.86	91.39
	GPU Time (ms)	32.70	18.12	10.61	5.40	2.94
	Speedup	**36.28**	**35.78**	**35.43**	**33.09**	**31.09**
DGTD-Central	**CPU Time (ms)**	1,059.54	573.13	331.46	157.05	80.63
	GPU Time (ms)	27.56	15.21	8.95	4.50	2.54
	Speedup	**38.45**	**37.69**	**37.02**	**34.92**	**31.73**
DGTD-Upwind	**CPU Time (ms)**	5,852.54	3,181.20	1,845.63	878.64	449.78
	GPU Time (ms)	189.84	103.90	60.26	29.71	16.22
	Speedup	**30.83**	**30.62**	**30.63**	**29.57**	**27.72**

Table 4.11: Per-step timing comparison for the branch-line coupler using mixed first-order basis functions

Number of MPI Nodes		1	2	4	8	16
DFDD-ELD	**CPU Time (ms)**	1,056.84	539.11	258.25	130.17	63.35
	GPU Time (ms)	11.45	6.02	3.19	1.84	1.10
	Speedup	**92.28**	**89.60**	**81.04**	**70.89**	**57.61**
DGTD-Central	**CPU Time (ms)**	1038.59	519.15	258.31	130.00	64.71
	GPU Time (ms)	11.03	5.81	3.11	1.77	1.15
	Speedup	**94.15**	**89.33**	**83.17**	**73.58**	**56.43**
DGTD-Upwind	**CPU Time (ms)**	4,984.80	2,494.60	1,244.17	624.56	309.70
	GPU Time (ms)	81.44	41.68	21.94	11.73	6.84
	Speedup	**61.21**	**59.86**	**56.70**	**53.23**	**45.27**

CHAPTER 5
GPU ACCELERATION OF THE NONLINEAR DISCONTINUOUS GALERKIN METHOD

5.1 Introduction

This chapter focuses on the application of the DGTD-Central algorithm to nonlinear electromagnetics with third-order nonlinearity. The Newton-Raphson formulation for the nonlinear algorithm is developed for the convergence of the nonlinear electric field due to the presence of Kerr-type medium. The MPI+GPU framework proposed in Chapter 4 is adapted for the acceleration of the nonlinear DGTD algorithm. The chapter is organized as follows: Section 5.2 formulates the third-order nonlinear DGTD algorithm using the Newton-Raphson iteration method, with performance comparison against the fixed-point method. Section 5.3 adapts the algorithm onto the MPI+GPU framework. Section 5.4 presents examples showing various Kerr-nonlinearity phenomena and documents the GPU speedups for each case.

5.2 Formulation

When only linear medium is considered, the electric permittivity varies linearly with the electric field, and the constitutive relation for the electric field takes the form of

$$\vec{D} = \varepsilon_0 \vec{E} + \vec{P} = \varepsilon_0 \vec{E} + \varepsilon_0 \chi^{(1)} \vec{E} \tag{5.1}$$

where $\vec{P}$ is the linear polarization intensity consisting of only the $\chi^{(1)}$ linear susceptibility of the medium. For a more general medium, Equation (5.1) can be augmented with nonlinear effects in power series [57] as

$$\vec{D} = \varepsilon_0 \vec{E} + \vec{P} = \varepsilon_0 \vec{E} + \varepsilon_0 [\chi^{(1)} + \chi^{(2)} \left\| \vec{E} \right\| + \chi^{(3)} \left\| \vec{E} \right\|^2 + \ldots] \vec{E} \tag{5.2}$$

where the second- and third-order nonlinear susceptibilities are denoted as $\chi^{(2)}$ and $\chi^{(3)}$, respectively. While the second-order nonlinear polarization exists only in materials with inversion symmetry [57], the third-order nonlinear polarization exists in most elements and crystals that are commonly used for optical devices [89]. It is worth noting that the nonlinear susceptibilities are generally tensors in nature. To simplify the derivations, simple non-

dispersive, lossless and symmetric materials are considered, reducing $\chi^{(3)}$ to a scalar term. Focusing on the third-order nonlinear effect, the time-varying relative permittivity for the third-order nonlinear medium can be written as

$$\varepsilon_r = \varepsilon_r(E) = \varepsilon_{r,L} + \varepsilon_{r,\mathrm{NL}} = \varepsilon_{r,\mathrm{L}} + \chi^{(3)}E^2 \tag{5.3}$$

where $\varepsilon_{r,\mathrm{L}}$ and $\varepsilon_{r,\mathrm{NL}}$ are the linear and nonlinear relative permittivity, respectively, and E is the notation for the time-varying electric field magnitude $\left\|\vec{E}(t)\right\|$. The following derivation for the nonlinear lossless and non-dispersive DGTD-Central algorithm pertains to the update of the electric field, since the update equation for the magnetic field has no nonlinear components and thus is completely identical to that of the linear DGTD-Central algorithm. Testing Ampere's law using the Galerkin method, we obtain

$$\iiint_{V_e} \vec{T} \cdot \left[\frac{\partial \vec{D}}{\partial t} - (\nabla \times \vec{H}) \right] dV = -\iiint_{V_e} \vec{T} \cdot \vec{J}_{\mathrm{imp}} dV \tag{5.4}$$

where the time-varying permittivity is embedded in the first term of Equation (5.4). Substituting in the expansion of the fields and applying the central flux, the equation after taking the time derivative on $\vec{D}$ becomes

$$\begin{aligned} &\iiint_{V_e} \left[\varepsilon_0 \frac{\partial \varepsilon_r}{\partial t} \vec{N}_i^e \cdot \vec{N}_j^e \right] dV \{e\} + \iiint_{V_e} \left[\varepsilon_0 \varepsilon_r \vec{N}_i^e \cdot \vec{N}_j^e \right] dV \frac{\partial \{e\}}{\partial t} + [S_e]\{h\} \\ &= \{f_e\} + [F_{eh}]\{h^+ - h\} \end{aligned} \tag{5.5}$$

where the expression for the various matrices and vectors can be found in Section 4.4.1, and the matrices for the boundary condition are omitted since we assume that the nonlinear region is always confined in the middle of the computation domain surrounded by the linear region. Notice that besides the omitted loss and dispersion terms, Equation (5.5) differs from Equation (4.35) in that since the time-varying permittivity is embedded in the original mass matrix $[M_e]$, the volume integration pertaining to the electric field is now split into two terms by the product rule, where for the nonlinear medium, both the relative permittivity and the electric field are functions of time. Discretizing Equation (5.5) in the time domain using central difference gives

$$\iiint_{V_e}\left[\varepsilon_0 \frac{\varepsilon_r^{n+1}-\varepsilon_r^n}{\Delta t}\vec{N}_i^e\cdot\vec{N}_j^e\right]dV\left(\frac{\{e\}^{n+1}+\{e\}^n}{2}\right)$$
$$+\iiint_{V_e}\left[\varepsilon_0 \frac{\varepsilon_r^{n+1}+\varepsilon_r^n}{2}\vec{N}_i^e\cdot\vec{N}_j^e\right]dV\left(\frac{\{e\}^{n+1}-\{e\}^n}{\Delta t}\right)=\{b\}^{n+1/2} \tag{5.6}$$

where ε_r^{n+1} is the field-dependent nonlinear permittivity at the future time step, ε_r^n is the converged permittivity at the current time step, and

$$\{b\}^{n+1/2}=-[S_e]\{h\}^{n+1/2}+\{f_e\}^{n+1/2}+[F_{eh}](\{h^+\}^{n+1/2}-\{h\}^{n+1/2}) \tag{5.7}$$

After rearranging the terms, Equation (5.6) can be casted into a field-marching form of

$$[M_e]^{n+1}\{e\}^{n+1}-[M_e]^n\{e\}^n=\{b\}^{n+1/2} \tag{5.8}$$

where

$$[M_e]^{n+1}=\frac{\varepsilon_0}{\Delta t}\iiint_{V_e}\left[\varepsilon_r^{n+1}\vec{N}_i^e\cdot\vec{N}_j^e\right]dV \tag{5.9}$$

and

$$[M_e]^n=\frac{\varepsilon_0}{\Delta t}\iiint_{V_e}\left[\varepsilon_r^n\vec{N}_i^e\cdot\vec{N}_j^e\right]dV \tag{5.10}$$

Due to the variation of the field magnitude at each time step, $\varepsilon_r(E)$ of each element changes with time, and therefore the mass matrix $[M_e]^{n+1}$ needs to be reassembled at every time step. Notice that we have recovered the original expression for $[M_e]$ as in the linear DGTD algorithm, albeit with a field- and time-dependent permittivity. The dependency of $\{e\}^{n+1}$ in $[M_e]^{n+1}$ renders Equation (5.8) a nonlinear equation. In the following derivation, the Newton-Raphson method will be adapted from [90] to solve for Equation (5.8) at every time marching step.

5.2.1 Nonlinear Formulation Using the Newton-Raphson Method

The Newton-Raphson method for nonlinear equations solves the problem

$$\vec{f}(\{x\})=0 \tag{5.11}$$

where $\vec{f}$ is a nonlinear function with $\{x\}$ being the unknowns. To find a converged solution, at any iteration step we first begin with an initial guess $\{x\}_0$ and plug it into the function. A closer solution to the real solution can be obtained using the equation

$$\{x\}_{k+1} = \{x\}_k - \frac{\vec{f}(\{x\}_k)}{\bar{J}(\{x\}_k)} \tag{5.12}$$

where $\bar{J}$ is the Jacobian matrix of $\vec{f}$ and is given as

$$J_{ij} = \frac{\partial f_i(\{x\})}{\partial x_j} \tag{5.13}$$

and the iteration stops when the residual, $\{r\}_k = \vec{f}(\{x\}_k)$, reduces below a predefined threshold. The Newton-Raphson is a preferable iteration method for solving nonlinear equations since it is capable of achieving quadratic convergence, where the negative exponent of the residual doubles after every iteration step. The Newton-Raphson method can be explained from a graphical point of view, as shown in Fig. 5.1, where the *x*-axis is the value of the unknown x, and the *y*-axis is the value of the residual function $r(x)$. The solution to the unknown lies at the intersection between $r(x)$ and the *x*-axis. During each iteration step, the Jacobian matrix marches the solution towards the direction of the derivative of the residual function, which is equivalent to drawing a tangent line and updates the solution to where the tangent line intersects with the *x*-axis. As long as there is a good initial guess and a unique solution for all values of $r(x)$ exists, the iteration will effectively converge to the real solution.

To solve for the field unknowns in Equation (5.8) using the Newton-Raphson method, we first cast the equation into the residual form

$$\{r(\{e\}^{n+1})\}^{n+1} = [M_e]^{n+1}\{e\}^{n+1} - [M_e]^n\{e\}^n - \{b\}^{n+1/2} \tag{5.14}$$

and expand the electric field using

$$\vec{E} = \frac{1}{2}\sum_{j=1}^{N} \vec{N}_j^e \left(e_j^{n+1} + e_j^n\right) \tag{5.15}$$

Note that the expansion the electric field using the central difference approach is essential to the quadratic convergence of the Newton-Raphson method. The first step to the explicit expression for the Jacobian matrix can be derived by using the product rule as

$$J_{ij} = \frac{\partial r_i}{\partial e_j^{n+1}} = M_{e,ij}^{n+1} + \sum_k \frac{\partial M_{e,ik}^{n+1}}{\partial e_j^{n+1}} \left(e_k^{n+1}\right) \tag{5.16}$$

where the partial derivative of the mass matrix is expressed as

$$\frac{\partial M_{e,ik}^{n+1}}{\partial e_j^{n+1}} = \frac{\varepsilon_0}{\Delta t} \iiint_{V_e} \frac{\partial \varepsilon_r}{\partial e_j^{n+1}} \vec{N}_i^e \cdot \vec{N}_k^e dV \tag{5.17}$$

The derivative of ε_r with respect to the basis expansion field unknowns can be computed using the chain rule as

$$\frac{\partial \varepsilon_r(E)}{\partial e_j^{n+1}} = \frac{\partial \varepsilon_r(E)}{\partial E}\frac{\partial E}{\partial e_j^{n+1}} \tag{5.18}$$

where

$$\frac{\partial E}{\partial e_j^{n+1}} = \frac{1}{2}\vec{N}_j^e \cdot \frac{\vec{E}}{E} \tag{5.19}$$

and

$$\frac{\partial \varepsilon_r(E)}{\partial E} = \frac{\partial}{\partial E}(\varepsilon_{r,\mathrm{L}} + \chi^{(3)}E^2) = 2\chi^{(3)}E \tag{5.20}$$

Combining Equations (5.16)-(5.20) into Equation (5.16), the explicit expression for the Jacobian matrix can be expressed as

$$\begin{aligned} J_{e,ij}^{n+1} &= M_{e,ij}^{n+1} + \frac{1}{\Delta t}\sum_k \left\{ \varepsilon_0 \iiint_{V_e} \left[2\chi^{(3)}E\right]\left[\frac{1}{2}\vec{N}_j^e \cdot \frac{\vec{E}}{E}\right]\left[\vec{N}_i^e \cdot \vec{N}_k^e\right] dV \left(e_k^{n+1}\right)\right\} \\ &= M_{e,ij}^{n+1} + \frac{\varepsilon_0}{\Delta t}\chi^{(3)} \iiint_{V_e} \left[\vec{N}_i^e \cdot \sum_k \vec{N}_k^e e_k^{n+1}\right]\left[\vec{N}_j^e \cdot \vec{E}\right] dV \end{aligned} \tag{5.21}$$

The second term on the right-hand side of Equation (5.21) pertains to $[\Delta M_e]$, the derivative of the mass matrix, which needs to be updated at every nonlinear iteration step using the updated $\{e_k^{n+1}\}$ from the previous iteration step.

At each time marching step n of using the Newton-Raphson method on Equation (5.8), the following procedures are performed:

1. Compute $\{b\}^{n+1/2}$. Set the initial guess for the electric field solution to $\{e\}_0^{n+1} = \{e\}^n$ and the mass matrix to $[M_e]_0^{n+1} = [M_e]^n$. Use the terms to compute the initial residual $\{r\}_0^{n+1}$ as in Equation (5.14).
2. At the kth iteration step, assemble $[\Delta M_e]_{k-1}^{n+1}$ using $\{e\}_{k-1}^{n+1}$ then update the Jacobian matrix using Equation (5.21).
3. Invert the Jacobian matrix and update the field solution $\{e\}_k^{n+1}$ using $\{e\}_{k-1}^{n+1}$ and $\{r\}_{k-1}^{n+1}$ in Equation (5.12).

4. Use the updated field solution to update the nonlinear permittivity $\varepsilon_r(E)$ and then in turn update the mass matrix $[M_e]_k^{n+1}$ using Equation (5.9).
5. Update the residual $\{r\}_k^{n+1}$ as in Equation (5.14) using $[M_e]_k^{n+1}$ and $\{e\}_k^{n+1}$. If the norm of the residual is smaller than the predefined threshold, then the nonlinear iteration has converged, and the equation can be marched to time step n+1. Otherwise, go back to Step 2 for the (k+1)th iteration step.

5.2.2 Comparison between the Newton-Raphson and the Fixed-Point Methods

In contrast to the Newton-Raphson method, every step in the fixed-point method is fast and straightforward, albeit the method comes with only a linear convergence. Comparing to the Newton-Raphson method, where the field solution at each iteration step is updated using the derivative to the residual function $r(x)$, the fixed-point method casts the system into the form of $g(x) = x$ and directly uses $g(x)$ as the updated solution, where the convergence is slow and is not easily guaranteed. At each time marching step n of using the fixed-point method on Equation (5.8), the following procedures are performed:

1. Compute $\{b\}^{n+1/2}$. Set the initial guess for the electric field solution to $\{e\}_0^{n+1} = \{e\}^n$ and the mass matrix to $[M_e]_0^{n+1} = [M_e]^n$.
2. At the kth iteration step, invert the mass matrix $[M_e]_{k-1}^{n+1}$ and update the field solution $\{e\}_k^{n+1}$ using $\{e\}_{k-1}^{n+1}$ by simple matrix-vector multiplication.
3. Use the updated field solution to update the nonlinear permittivity $\varepsilon_r(E)$ and then in turn update the mass matrix $[M_e]_k^{n+1}$ using Equation (5.9).
4. Update the residual $\{r\}_k^{n}$ as in Equation (5.14) using $[M_e]_k^{n+1}$ and $\{e\}_k^{n+1}$. If the norm of the residual is smaller than the predefined threshold, then the nonlinear iteration has converged, and the equation can be marched to time step n+1. Otherwise, go back to Step 2 for the (k+1)th iteration step.

Figure 5.2 shows the convergence comparison between the Newton-Raphson and the fixed-point method for the marching of the electric field during a fictitious time marching step in an arbitrary nonlinear element. While the fixed-point method took over 100 iteration steps to

converge to slightly below our predefined threshold, 1e-10, the Newton-Raphson method took only 4 iteration steps to converge to a residual around 1e-16. Notice that the residual drops by a double order of magnitude with each Newton-Raphson iteration step, and this validates the Newton-Raphson formulation in this section. It is worth noting that while in general the Newton-Raphson method is capable of converging nonlinear function within a few steps, each fixed-point iteration process is very light, which is computationally preferable for problems with moderate nonlinearity. However, if the problem is heavily nonlinear, the fixed-point method might require too many iteration steps if it is even capable of converging the solution at all. In this case the Newton-Raphson method should be employed instead.

5.3 GPU Implementation

The GPU implementation for the nonlinear DGTD is an extension to the linear DGTD framework, employing the same coalesced memory accessing pattern and thread/block allocation. During each time marching step, the magnetic field update process for the nonlinear DGTD is completely identical to the original DGTD. In the electric field update process, the procedures also remain the same for all the terms that are not related to $\{e\}^{n+1}$. The adaptation effort therefore focuses on the parallelization for the additional electric field update process, specifically, the parallelization of the $\{e\}^{n+1}$-related nonlinear volume matrix assembly, and the inversion of this volume matrix. For the fixed-point method, the volume matrix is simply the nonlinear mass matrix $[M_e]^{n+1}$; for the Newton-Raphson method, it is the Jacobian matrix $[J_e]^{n+1} = [M_e]^{n+1} + [\Delta M_e]^{n+1}$ as described in Equation (5.21).

To assemble the nonlinear volume matrix, note that each volume matrix entry is numerically integrated through quadrature, where the contribution from each weighted quadrature point is summed. Due to the presence of nonlinearity, ε_r on each quadrature point changes during each iteration step, while the other constituting terms in Equation (5.9) remain identical. To parallelize the assembly of the volume matrix, the constituting matrices at each quadrature point is pre-calculated and stored, and are then summed together at each iteration step by first multiplying with the updated ε_r. For the $[\Delta M_e]$ term in the Jacobian matrix, the constituting matrices need to be reassembled due to the changes in the field solution. In this case

the basis functions on each quadrature are stored and accessed for the matrix construction in a coalesced fashion. The proposed parallelization strategy and the memory access pattern is shown in Fig. 5.3. The above processes are completely parallelizable. However, the potential drawback of this approach is that as the number of quadrature points of a problem increases due to the use of higher order basis functions, the memory requirement becomes more stringent.

To invert the nonlinear volume matrix, we parallelized the standard non-pivoting element-level Gaussian elimination on the GPU. Although the elimination is only semi-parallelizable, the batch processing of the elimination process for the nonlinear elements somewhat provides a decent speedup. Note that it is well known that the FEM mass matrix has a small condition number, and therefore it can be easily inverted using the standard Gaussian elimination without partial pivoting. This is beneficial for the GPU acceleration since the partial pivoting process involves many conditional statements and branches, which are undesirable for the parallelization on GPU.

5.4 Examples and Speedups

To demonstrate the validity of the proposed nonlinear DGTD algorithm and its parallelization efficiency on the GPU, various electromagnetics problems demonstrating the Kerr-type phenomenon are considered below. The XSEDE Stampede cluster with NVIDIA's NVIDIA Tesla M2090 and CUDA v.5.0 is used as the GPU cluster for the examples, with all the timing approaches identical to those found in Chapter 4.

5.4.1 Coaxial Cable Demonstrating the Self-Phase Modulation and the Third-Harmonic Generation

The first example is a coaxial cable shown in Fig. 5.4 with an inner and outer radius of 1 mm and 2 mm, respectively, and a length of 40 mm. Small sections of linear medium are placed near the two ports for the linear excitation and absorption of the fields, while the rest of the coaxial cable is filled with either a linear or nonlinear medium, using a linear permittivity of $\varepsilon_{r,L} = 1.0$ and a third-order nonlinearity coefficient of $\chi^{(3)} = 4\text{e-}8$. The input signal is a modulated Gaussian pulse with a center frequency of 20 GHz. The number of elements is 110,715, marching at a time step of $\Delta t = 0.075\text{ps}$ for a total of 10,000 time steps for both the

linear and nonlinear cases. Mixed first-order basis functions are used for the solution. The incident TEM wave has an electric field strength profile of $\hat{r}/[\ln(b/a)r]$, and the time domain response for the two cases is shown in Fig. 5.5. The specific electric field excitation generates a weak nonlinearity in the medium, which generates a maximum instantaneous relative permittivity of $\varepsilon_r = 1.055$, or a 5.5% change to the linear relative permittivity. It is shown that for the linear case, the shape of the output signal is identical to the input, while the nonlinear medium generates shock waves (self-steepening effect) at the output [57], [91]. The self-steepening effect can be explained by pulse propagation in medium with varying permittivity. As a pulse propagates down the nonlinear medium at any instance of time, the leading edge of the pulse modifies the nonlinear medium and increases the relative permittivity. Since a wave travels slower in a medium with a higher permittivity, the leading edge portion of the pulse slows down and stacks on the top of the trailing edge, producing a steepening trailing edge. The self-steepening effect intensifies with the presence of a higher nonlinearity, either through a higher nonlinearity coefficient or a larger electric field.

The frequency domain response for the output scattering parameter is shown in Fig. 5.6. For the linear case, we have retained the frequency profile of the original Gaussian pulse centered at 20 GHz. For the nonlinear case, the third-harmonic effect generates harmonics at odd multiples of the original 20 GHz signal at 60 GHz, 100 GHz, 140 GHz, and so on. In addition, the self-phase modulation effect broadens the input bandwidth, where the leading and the trailing edge shift to lower and higher frequencies, respectively [57]. The problem is validated using COMSOL, and the results show good agreement. Tables 5.1 and 5.2 give the average per-step CPU and GPU timing for the solution using the fixed-point and the Newton-Raphson method, respectively. The timings are made using mixed first-order basis functions on various numbers of cluster nodes. Since the nonlinearity strength during the solution varies with both space and time, not all elements go through the same iterations of nonlinear convergence at any moment, and therefore it is difficult to compute the relevant bandwidths. Nonetheless, some observations can be made. While strong scaling is still preserved, that is, the total per-step time halves when doubling the number of processing nodes, the GPU speedup for the nonlinear DGTD parallelization is lower than the original element-level finite element parallelization framework proposed in this dissertation. This is in large due to the uneven nonlinearity encountered by the elements, and the semi-serial nature of the Gaussian elimination process.

The effect is that some of the threads in a warp will be stalled and wait for the others to complete their calculations before the warp can carry out the next instruction as a whole. This thread idleness effectively lowers the number of FLOPS as well as the overall bandwidth. The bottleneck is especially apparent for the fixed-point method, since the bulk of its nonlinear iteration process focuses on the Gaussian elimination, and a significantly larger communication time due to the different level of instantaneous nonlinearity present on each decomposed MPI region at any instance. On the other hand, since there are many more processes taken by the Newton-Raphson method besides the Gaussian elimination, the Newton-Raphson method effectively has higher GPU speedups.

5.4.2 Photonic Amplifier Demonstrating the Frequency Mixing Effect

The second example is a silicon photonic amplifier [92] shown in Fig. 5.7 demonstrating the four-wave mixing effect [57]. The four-wave mixing effect adds and subtracts three input frequencies f_1, f_2, and f_3 at various combinations and produces a fourth frequency f_4 for each of the combinations. If only two input frequencies are present, the mixing produces combinations of the two excitation frequencies and a duplicating frequency, such as $2f_{\text{pump}} - f_{\text{signal}}$, $2f_{\text{signal}} - f_{\text{pump}}$, and so on. In this problem, a signal excitation at f_{signal} and a pump excitation at f_{pump} are given. With $f_{\text{pump}} = f_1 = f_2$, the mixing produces an "idler" centered at $f_{\text{idler}} = 2f_{\text{pump}} - f_{\text{signal}} = 190.2\text{THz}$. The mechanism behind the amplification on the signal excitation is that while propagating in the nonlinear medium, the pump nonlinearly mixes with the signal to first produce a small idler, and the idler grows as the waves travels further down the nonlinear medium. While growing, the idler will in turn nonlinearly mix with the pump to produce a $2f_{\text{pump}} - f_{\text{idler}}$, which lands right back at the frequency band of the signal. Therefore, the idler and the signal amplify each other through the mixing effect with the pump.

The cross section of the photonic amplifier consists of a nonlinear silicon core with a dimension of $550\text{nm} \times 300\text{nm}$, and a SiO_2 cladding with a dimension of $2.550\mu\text{m} \times 2.3\mu\text{m}$. The length of the silicon structure is $40\mu\text{m}$, which is scaled from the original problem to reduce the computation time, while the nonlinearity is also adjusted accordingly. The analytical input excitations are shown in Fig. 5.8, where there is a pump with $f_{\text{pump}} = 193.4\text{THz}$ and a signal with

$f_{\mathrm{signal}} = 196.6\mathrm{THz}$, and the magnitude of the pump is 100 times that of the signal. The excitation is fed through a PEC waveguide connecting to the silicon core. The linear relative permittivity for the silicon and the oxide are $\varepsilon_{r,\mathrm{Si},L} = 12.1104$ and $\varepsilon_{r,\mathrm{SiO2}} = 2.1025$, respectively, with a third-order nonlinearity coefficient of $\chi^{(3)} = 0.02$ in the silicon core. The number of elements is 2,751,462, marching at a time step of $\Delta t = 0.025\mathrm{fs}$ for a total of 840,000 time steps, where mixed first-order basis functions are used for the solution. Probes at the input and output of the silicon waveguide are used to measure the magnitude of the fields before and after the amplification.

When only the signal excitation is injected into the photonic waveguide, fields at the output remain identical to the original excitation at the input, since the signal strength is not strong enough to invoke a significant nonlinearity in the waveguide. This is shown in Fig. 5.9, where the signal excitation is normalized against the amplitude of the pump. When both the signal and the pump are injected into the photonic waveguide, the strong pump excitation invokes a significant nonlinearity in the waveguide and triggers the frequency mixing effect, as shown in Fig. 5.10. For the field at the input in Fig. 5.10(a), the pump and signal are observed, though with an enlarged bandwidth for the pump. This spectral broadening is produced by the reflection of the self-phase modulated bandwidth of the pump broadened after passing through the nonlinear material. On the other hand, since the signal excitation is relatively weak, no broadening can be observed.

For the field at the output in Fig. 5.10(b), the three peaks from left to right are the idler, the pump, and the signal. A couple of observations can be made. First, the self-phase modulation the pump experiences in the nonlinear waveguide becomes apparent, where its bandwidth is broadened. In addition, since some of its power is mixed into the signal and the idler according to the four-wave mixing theory, the power at 193.4THz is reduced, leaving a slight dip in the center of the broadened pump band. Second, the mixing of the signal and the pump generates an idler at 190.2 THz, which agrees with the theoretical prediction. Notice that the bandwidth of the signal at the output also broadens. This is due to its mixing with the broadened pump bandwidth, which first generates a wider idler bandwidth and then broadens the signal band itself. The peak on/off signal gain is 5.18dB, which agrees with the measurement result in [92] and the CST simulation.

For the solution time, the average GPU time per marching step is 20.64ms using 16 GPU nodes, which results in a total solution time of 4.8 hours. The CPU solution time is expected to be much longer than the GPU and is thus omitted. Nonetheless, according to the timing analysis from the previous example, it is shown that the acceleration trend for the linear DGTD algorithm can be applied towards the nonlinear DGTD algorithm as well. Therefore, it is expected that this large photonic amplifier problem will result in a higher GPU speedup than the previous example.

5.4.3 Propagation in Bulk Medium Demonstrating the Self-Focusing Effect

The last example demonstrates the self-focusing effect through beam-shaped field propagation in a bulk medium. A beam-shaped field, such as lasers, has a Gaussian profile for its transverse field intensity. When the beam propagates through a nonlinear bulk medium, the field will induce a variation in the medium refractive index (a function of permittivity) with a transverse Gaussian profile, where the refractive index at the medium overlapping the beam axis becomes larger than the one at the rim of the beam, effectively creating an induced waveguide in the bulk medium. The induced waveguide helps countering natural diffraction, and mitigates the temporal and spatial broadening of the beam as it propagates in the medium. If the mode of the beam matches the induced waveguide mode, solitons of fixed spatial magnitude will be created [59].

The geometry of the problem is shown in Fig. 5.11, where the 1mm×1mm×3mm cyan bulk medium waveguide is the nonlinear bulk medium and the grey cladding is the linear region used for the absorbing boundary condition. The linear relative permittivity for both regions is $\varepsilon_r = 1.0$, with a third-order nonlinearity coefficient of $\chi^{(3)} = 8$. The excitation is a tapered TEM sine wave at 300GHz, and is fed through a square PEC waveguide segment with a dimension that is half the excitation wavelength. The PEC waveguide segment creates an opening to the bulk medium and shapes the transverse intensity of the outgoing wave with a Gaussian profile. The number of elements is 664,039, marching at a time step of $\Delta t = 0.01\text{ps}$ for a total of 5,000 time steps, where mixed first-order basis functions are used for the solution.

Figure 5.12 shows the field profiles in the bulk medium at various times for both linear and nonlinear media. The specific electric field excitation generates a strong nonlinearity in the medium, which generates a maximum instantaneous relative permittivity of $\varepsilon_r = 8.27$, or a 727%

change to the linear relative permittivity. It can be seen in each plot that due to nonlinearity, the excitation experiences pulse compression which shortens the duration of each pulse. This effect is due to self-phase modulation. As the excitation propagates down the bulk medium, the waves are naturally diffracted in the linear medium, where the magnitude of the waves decreases significantly after a couple of wavelengths. In the nonlinear medium, on the other hand, the intensity of the field modifies the surrounding medium into a self-induced waveguide, counteracting natural diffraction and preserving the magnitude of the propagating wave for a longer distance into the medium, as shown in the figures.

Table 5.3 shows the GPU average per-step timing for the solution using the fixed-point and the Newton-Raphson methods. Due to the large nonlinearity of the problem, the average fixed-point iteration counts per time step for the solution greatly outnumbers the number of iterations needed for the Newton-Raphson method, and therefore the total solution time using the Newton-Raphson method is shorter than using the fixed-point method. In addition, while different MPI-decomposed regions experience different levels of nonlinearity due to the propagation of the field, some MPI nodes must idle and wait for the others to iteratively converge before moving onto the next time step together. This idling time is taking into account in the average communication time, where it is much higher for the fixed-point method due to the large number of iteration differences between different regions at any particular moment.

5.5 Figures and Tables

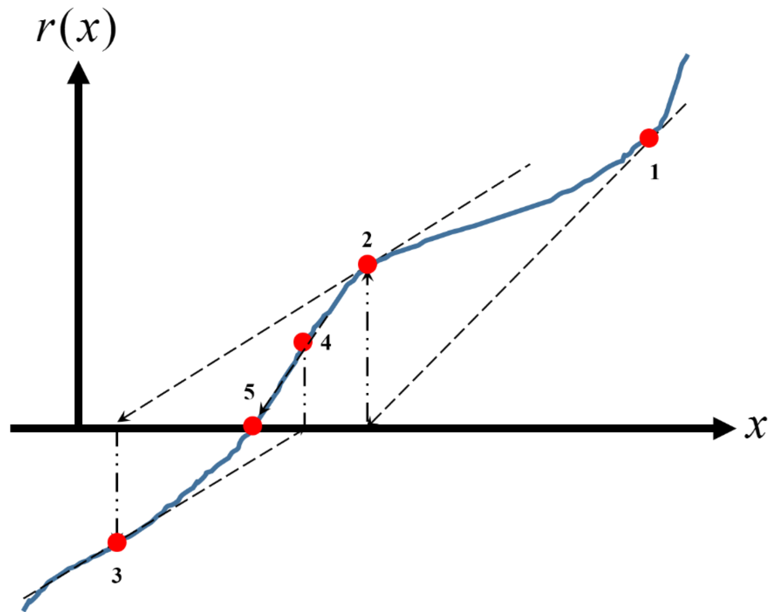


Fig. 5.1: Newton-Raphson iteration steps illustrated in 1-D plot.

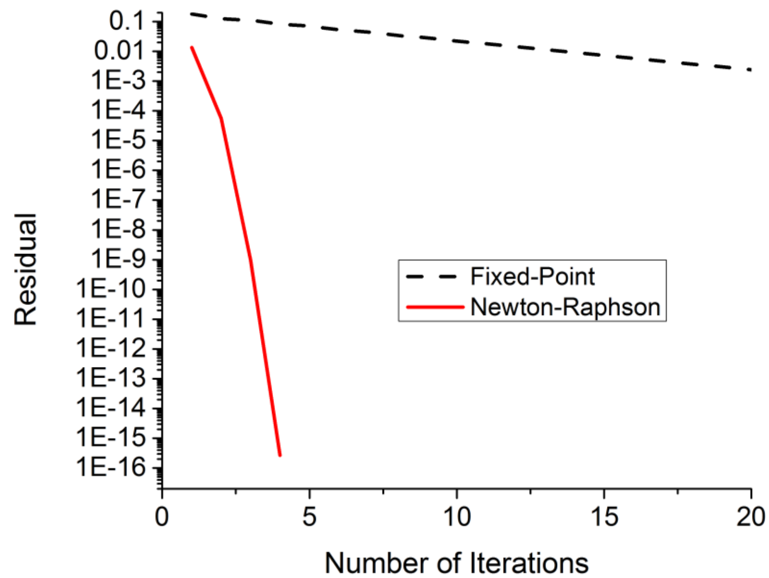


Fig. 5.2: Number of iterations taken by the fixed-point and the Newton-Raphson methods.

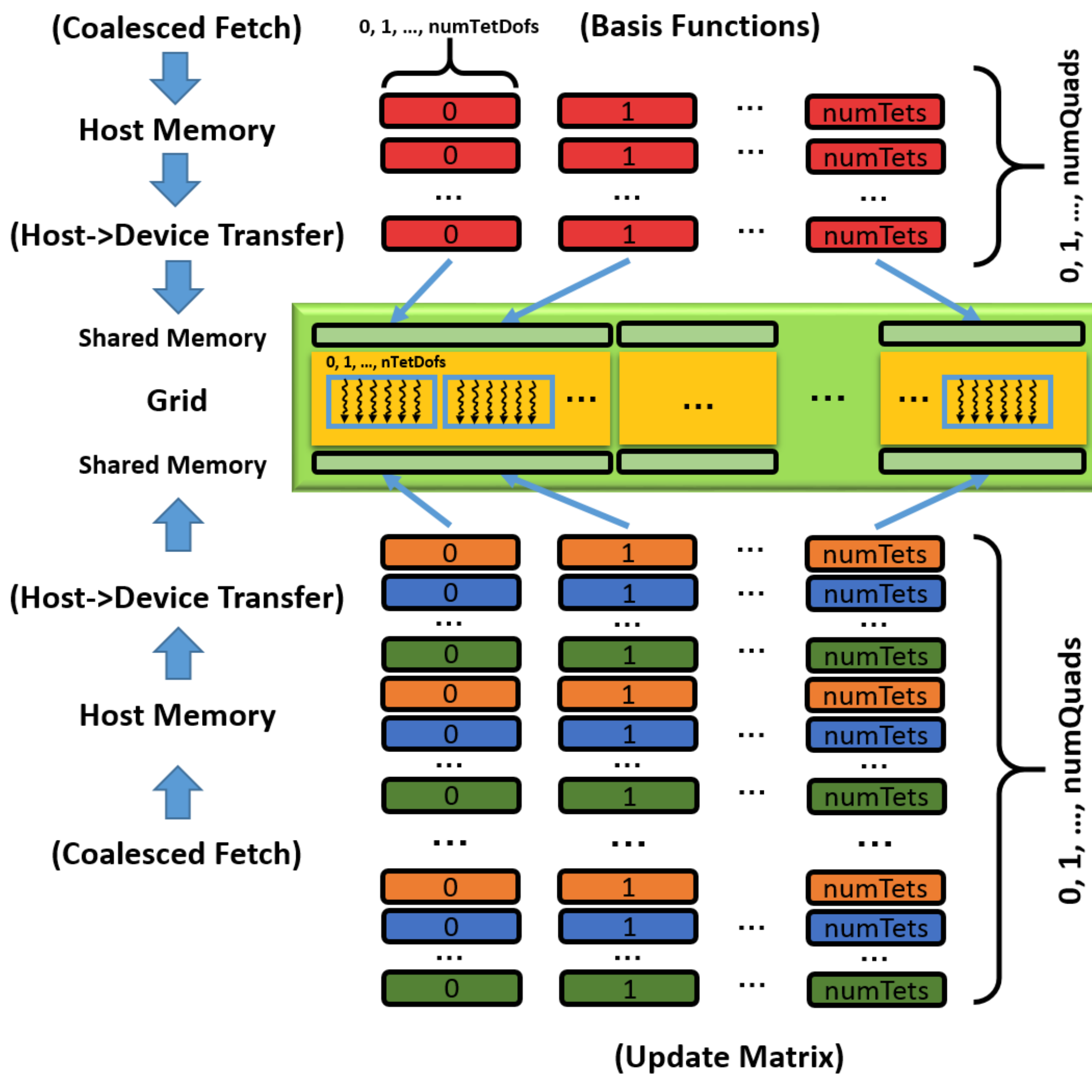


Fig. 5.3: Parallelization strategy and memory access pattern for the assembly of the nonlinear volume matrices.

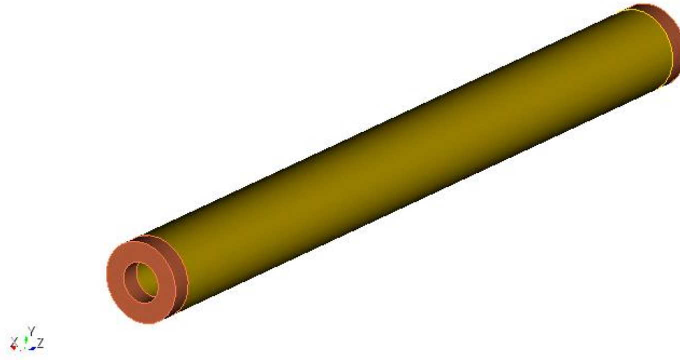


Fig. 5.4: Geometry of the coaxial cable.

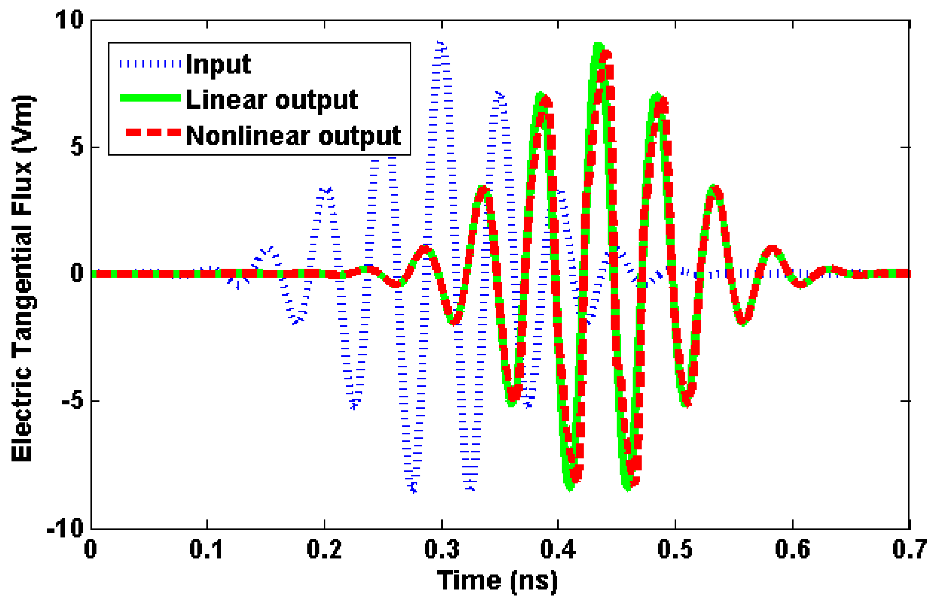


Fig. 5.5: Time domain response of the electric flux at the input and the output ports for both the linear and the nonlinear medium.

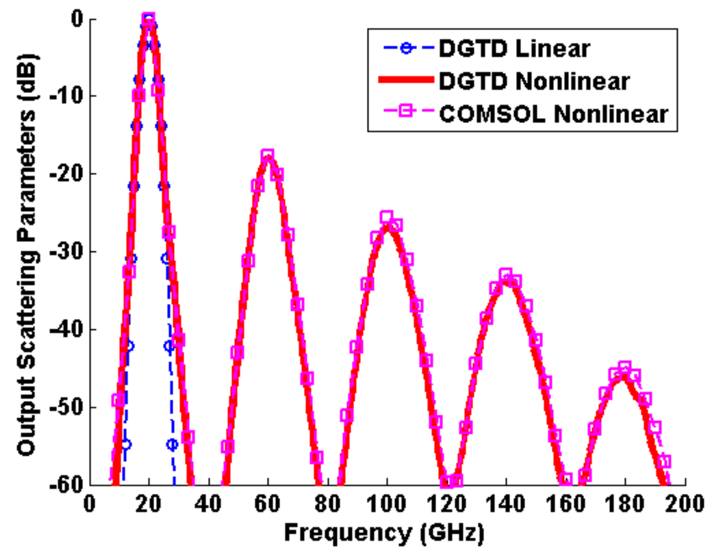


Fig. 5.6: Frequency domain response of the scattering parameter at the output port for both the linear and the nonlinear medium.

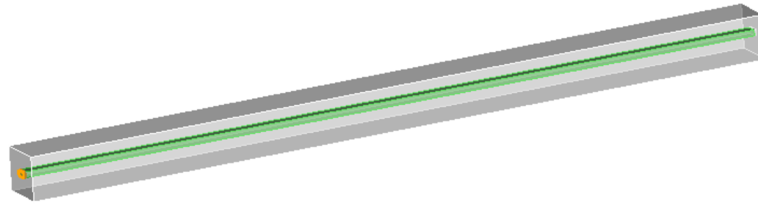


Fig. 5.7: Geometry of the photonic amplifier.

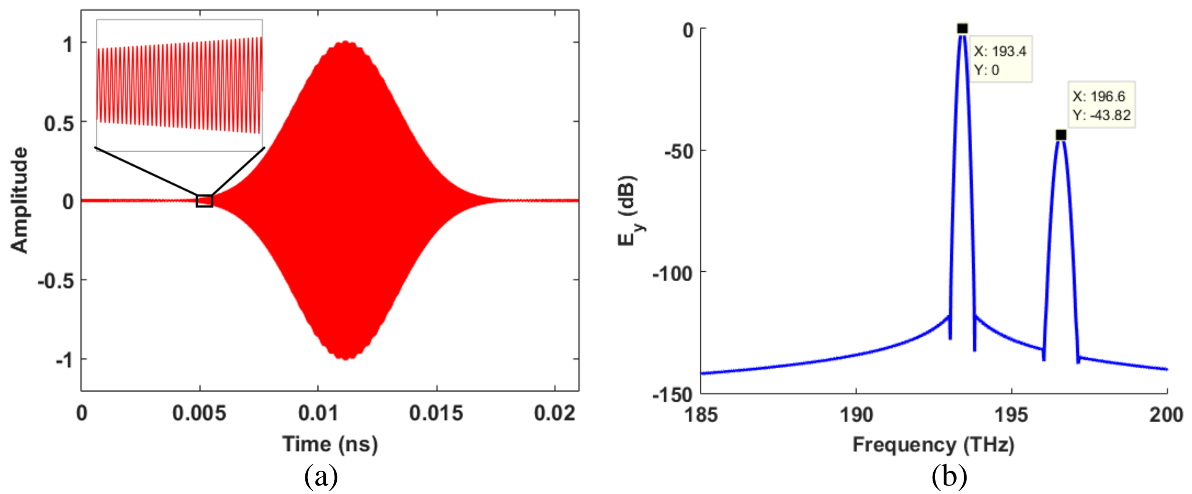


Fig. 5.8: Excitation for the photonic amplifier in the a) time domain and b) frequency domain. The pump frequency is centered at 193.4 THz and the signal frequency is centered at 196.6 THz.

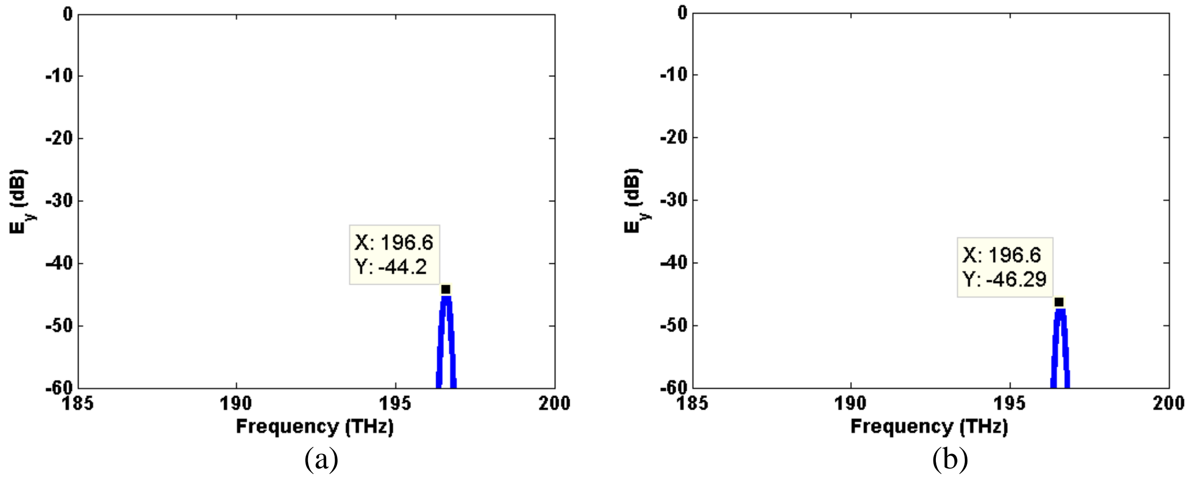


Fig. 5.9: The probed electric field at the a) input and b) output of the waveguide for excitation with signal only.

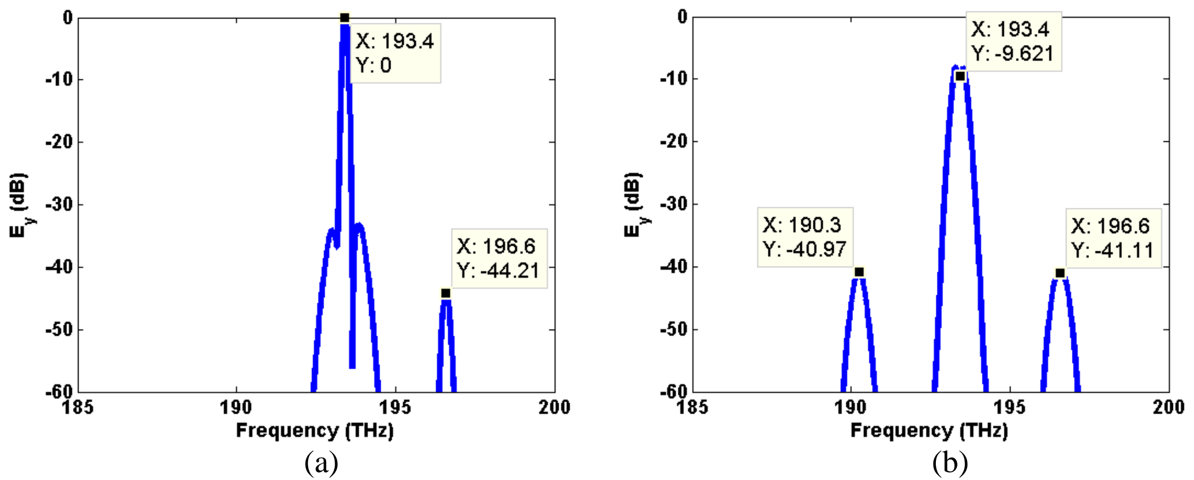


Fig. 5.10: The probed electric field at the a) input and b) output of the waveguide for excitation with signal and pump.

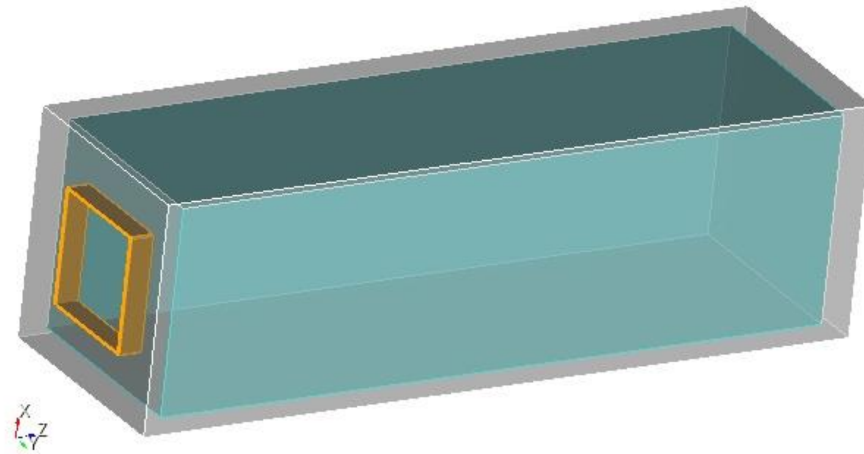


Fig. 5.11: Geometry of the bulk medium waveguide.

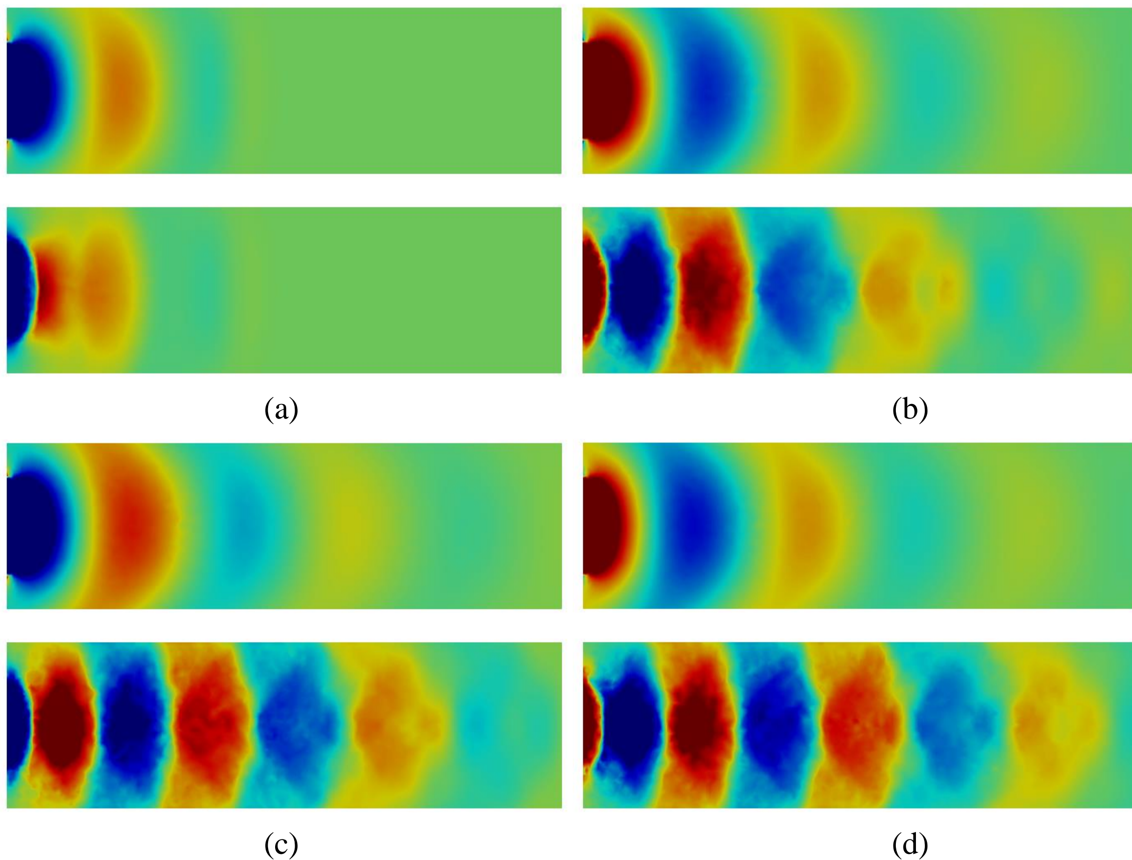


Fig. 5.12: Field profile in the bulk medium at a) 5 ns, b) 20 ns, c) 35 ns, and d) 50 ns. The upper graph in each figure shows the field profile traveling in a linear medium, while the lower graph shows the field profile traveling in the nonlinear medium.

Table 5.1: Average per-step timing comparison for the nonlinear coaxial cable with fixed-point iteration.

Number of MPI Nodes		1	2	4	8
CPU Time per Step (ms)	**Volume Processes**	619.85	309.99	155.18	78.26
	Surface MV	169.45	83.20	40.71	18.37
	Total Marching	798.15	397.41	197.84	97.55
	Communication	0	33.63	30.65	35.96
	Total Per-Step	798.15	431.04	228.49	133.51
GPU Time per Step (ms)	**Volume Processes**	32.48	16.33	8.23	4.19
	Surface MV	1.69	0.88	0.47	0.26
	Total Marching	34.23	17.26	8.76	4.51
	Communication	0	2.27	1.52	2.63
	Total Per-Step	34.23	19.53	10.28	7.14
Speedup		**23.32**	**22.07**	**22.23**	**18.71**

Table 5.2: Average per-step timing comparison for the nonlinear coaxial cable with Newton-Raphson iteration.

Number of MPI Nodes		1	2	4	8
CPU Time per Step (ms)	**Volume Processes**	1,305.10	654.65	327.22	163.95
	Surface MV	168.03	82.72	40.53	18.28
	Total Marching	1,482.00	741.61	369.71	183.15
	Communication	0	35.51	33.74	38.79
	Total Per-Step	1,482.00	777.12	403.45	221.94
GPU Time per Step (ms)	**Volume Processes**	45.46	22.80	11.47	5.83
	Surface MV	1.69	0.87	0.47	0.26
	Total Marching	47.21	23.73	12.00	6.14
	Communication	0	2.94	1.53	4.57
	Total Per-Step	47.21	26.67	13.52	10.71
Speedup		**31.39**	**29.14**	**29.83**	**20.72**

Table 5.3: Average GPU per-step timing comparison for the bulk medium propagation with fixed-point and Newton-Raphson iterations.

Number of MPI Nodes		1	2	4	8
Fixed-Point Time per Step (ms)	**Volume Processes**	569.93	287.23	142.95	72.41
	Surface MV	10.01	5.04	2.55	1.31
	Total Marching	580.11	292.34	145.57	73.81
	Communication	0	112.80	97.24	59.59
	Total Per-Step	**580.11**	**405.14**	**242.81**	**133.4**
Newton-Raphson Time per Step (ms)	**Volume Processes**	354.91	177.85	89.05	44.60
	Surface MV	10.01	5.04	2.55	1.31
	Total Marching	365.00	182.97	91.66	45.98
	Communication	0	15.54	10.60	8.52
	Total Per-Step	**365.00**	**198.51**	**102.26**	**54.50**

CHAPTER 6
CONCLUSION AND FUTURE RESEARCH

This dissertation focuses on the investigation of the GPU architecture and the underlying parallelization capability and applies the findings to the acceleration of the finite element analysis of electromagnetic problems. The traditional finite element method in the frequency domain is first investigated, followed by the acceleration of the domain-decomposition-enabled finite element algorithm in the time domain. The proposed GPU parallelization framework is then adapted onto the formulated third-order nonlinear DGTD algorithm. This chapter concludes the work and proposes possible future research directions.

In this work, we first described the bottlenecks in the GPU parallelization of the finite element algorithm for electromagnetic analysis and proposed possible parallelization schemes on a single GPU and multiple GPUs. Through the examples, we showed that for a single GPU, parallelizing the construction of the elemental matrices is straightforward, and the assembly-by-DOF approach on GPU is able to achieve speedup over a CPU. When higher-order finite elements are used, there will be more calculations per unknown, and GPU acceleration will be more pronounced. For the solution phase, we showed that by using existing CUDA libraries for an iterative solver and by incorporating an effective preconditioner, we can successfully reduce the total computation time using a GPU. In particular, for time-domain finite element analyses, the GPU-based BiCGStab coupled with CPU-based ILU preconditioner yields a significant speedup over the CPU-based parallel direct solver, such as Pardiso in MKL, and this speedup increases as the problem size increases. For frequency-domain analyses, the GPU-based BiCGStab coupled with a GPU-based parallel preconditioner, such as the SSOR-AI preconditioner, can also achieve a speedup by nearly an order of magnitude over the CPU implementation of the same algorithm. We demonstrated the GPU acceleration performance through a variety of EMC, antenna, and scattering examples using both the time-domain and frequency-domain finite element methods. Our preliminary investigation on multi-GPU computation showed that more advanced approaches, such as domain decomposition algorithms, are needed in order to achieve a significant speedup for the finite element computation of large-scale electromagnetic analyses.

The GPU acceleration of the DFDD-ELD algorithm incorporated with the capability of modeling general lossy and dispersive materials is carried out in this work. The element-level decomposition is highly parallelizable and thus produces a higher speedup than the aforementioned traditional FEM. The DFDD-ELD algorithm is implemented under a GPU cluster environment using hybrid MPI-CUDA, where the memory layout and thread allocation are carefully assigned to perform the time-marching of the solution in a fully coalesced fashion. As the DFDD-ELD algorithm is memory-intensive, bandwidth is used as a good indicator of GPU utilization. It is shown from the examples that large problems can be accelerated with a high speedup on a GPU cluster. However, while a large number of elements is needed to preserve the bandwidth and hence the overall speedup, there exists a bandwidth threshold for the DFDD-ELD algorithm due to the nature of its very-small subdomain matrices. Nevertheless, a higher bandwidth can still be achieved by using higher-order basis functions, which enlarge the dimension of the subdomain matrices and attain a higher shared memory utilization for the multiplier vectors. The same conclusions can be drawn for the array configuration of the DFDD-ELD algorithm as well, where the memory savings through the usage of unit cells enable more elements to be processed per GPU node and hence save on computation resources. The incorporation of the local time-stepping sizes can further reduce the overall computation time through marching different batches of similarly-sized elements on different time step sizes, then synchronizing the solutions at some fixed time intervals. Based on the proposed parallelization framework, the acceleration of the DGTD-Central and DGTD-Upwind algorithms can be easily realized. It is shown through the examples that comparable speedup and bandwidth can be achieved for all three algorithms.

The proposed parallelization framework could be further utilized in the application of the explicit element-level algorithms to nonlinear electromagnetic problems. The modeling of third-order Kerr-type nonlinearity is presented in this dissertation. The nonlinear DGTD algorithm using both the fixed-point and the Newton-Raphson iteration methods for the convergence of instantaneous third-order Kerr-type nonlinearity is developed, and various nonlinear phenomena are captured and presented through the examples. It is shown that the proposed GPU parallelization framework is well adapted onto the nonlinear DGTD algorithm, where the various speedup analysis and findings for the linear algorithm can also be reached for the nonlinear algorithm, albeit generally with a lower GPU speedup due to the semi-parallelizable nature of the

Gaussian elimination process used in the nonlinear iterative convergence process. When the nonlinearity in a problem is weak, the solution using the fixed-point method is faster than the Newton-Raphson method since each fixed-point iteration process is relatively simple, whereas when the nonlinearity in a problem is strong, the solution using the Newton-Raphson method is faster instead due to the low iteration count resulting from its quadratic convergence.

Two general future research directions can be pursued. First, further optimization can be made on the mapping of the element-level finite element algorithms onto the GPU environment. The proposed scheme has taken the basic architecture of the GPUs into account when mapping the algorithms to achieve coalesced memory access, and efforts have been made to mitigate the random access bottleneck. However, further detailed tweaking of the shared memory banks and the various constant memory caches could prevent conflicting requests from the threads and improve accessing efficiency for these fast memories. Many additional existing GPU features, such as asynchronous concurrent execution between the host and the device, concurrent kernel execution, and dynamic parallelism, can also be incorporated to reduce thread idleness for both the CPU hosts and the GPU devices. Furthermore, with the ongoing hardware advancements on the GPUs, emerging GPU technologies could potentially be utilized to better manage the warp scheduling and thread executions.

The second possible future research direction is the advancement for the nonlinear DGTD algorithm. The proposed algorithm works well for problems with instantaneous nonlinearity, that is, when the time duration of the wave pulse in a problem is longer than the material nonlinear response time. However, when the frequency of the problem further increases, one can no longer approximate the response from the material as instantaneous nonlinearity, and new algorithms incorporating the dispersion and lossy effects need to be developed. These effects include the simple linear Debye and Lorentz dispersion, and the nonlinear dispersion phenomenon as the Raman effect due to the vibrations of the optical phonons, and the Brillouin effect due to the acoustical phonons [57]. These dispersion effects involve convolution in the time domain, which can be solved using auxiliary differential equations [62]. The incorporation of time-varying permittivity in full tensor form can also be formulated to study other nonlinear phenomena, such as the second-order nonlinearity.

REFERENCES

[1] D. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Waltham, MA: Morgan Kaufmann, 2013.

[2] M. Flynn, “Some computer organizations and their effectiveness,” *IEEE Trans. Comput.*, vol. C-21, no. 9, pp. 948-960, Sept. 1972.

[3] M. Macedonia, “The GPU enters computing’s mainstream,” *Computer*, vol. 36, no. 10, pp. 106-108, 2003.

[4] S. E. Krakiwsky, L. E. Turner and M. M. Okoniewski, “Graphics processor unit (GPU) acceleration of finite-difference time-domain (FDTD) algorithm,” *Proc. Int. Symp. Circuits and Systems*, vol. 5, pp. 265-268, 2004.

[5] M. J. Inman and A. Z. Elsherbeni, “Programming video cards for computational electromagnetics applications,” *IEEE Antennas Propag. Mag.*, vol. 47, no. 6, pp. 71–78, Dec. 2005.

[6] K. Liu, X.-B Wang, Y. Zhang and C. Liao, “Acceleration of time-domain finite element method (TD-FEM) using graphics processor units (GPU),” in *2006 7th International Symposium on Antennas, Propagation & EM Theory*, 2006.

[7] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Boston, MA: Addison-Wesley Professional, 2003.

[8] K. Fatahalian and M. Houston, “A closer look at GPUs,” *Communications of the ACM*, vol. 51, no. 10, pp. 50-57, Oct. 2008.

[9] S. Peng and Z. Nie, “Acceleration of the method of moments calculations by using graphics processing units,” *IEEE Trans. Antennas Propag.*, vol. 56, no. 7, pp. 2130-2133, Jul. 2008.

[10] T. Topa, A. Karwowski, and A. Noga, “Using GPU with CUDA to accelerate MoM-based electromagnetic simulation of wire-grid models,” *IEEE Trans. Antennas Propag. Lett.*, vol. 10, pp. 342-345, Apr. 2011.

[11] D. De Donno, A. Esposito, G. Monti, and L. Tarricone, “Parallel efficient method of moments exploiting graphics processing units,” *Microw. Opt. Techn. Lett.*, vol. 52, no. 11, pp. 2568-2572, Nov. 2010.

[12] S. Li, B. Livshitz, and V. Lomakin, “Fast evaluation of Helmholtz potential on graphics processing units (GPUs),” *J. Comput. Phys.*, vol. 229, no. 22, pp. 8463-8483, Nov. 2010.

[13] E. Lezar and D. B. Davidson, “GPU-accelerated method of moments by example: Monostatic scattering,” *IEEE Antennas Propag Mag.*, vol. 52, no. 6, pp. 120-135, Dec. 2010.

[14] S. Li, R. Chang, A. Boag, and V. Lomakin, “Fast electromagnetic integral-equation solvers on graphics processing units,” *IEEE Antennas Propag Mag.*, vol. 54, no. 5, pp. 71-87, Oct. 2012.

[15] P. Sypek, A. Dziekonski, and M. Mrozowski, “How to render FDTD computations more effective using a graphics accelerator,” *IEEE Trans. Magn.*, vol. 45, no. 3, pp. 1324-1327, Mar. 2009.

[16] V. Demir, “A stacking scheme to improve the efficiency of finite-difference time-domain solutions on graphics processing units,” *Applied Computational Electromagnetics Society Journal*, vol. 25, no. 4, pp. 323-330, Apr. 2010.

[17] V. Demir and A. Z. Elsherbeni, "Compute unified device architecture (CUDA) based finite-difference time-domain (FDTD) implementation," *Appl. Comput. Electrom.*, vol. 25, no. 4, pp. 303-314, Apr. 2010.

[18] K. Xu, Z. Fan, D.-Z. Ding, and R.-S. Chen, "GPU accelerated unconditionally stable Crank-Nicolson FDTD method for the analysis of three-dimensional microwave circuits," *Prog. Electromagn. Res.*, vol. 102, pp. 381-395, 2010.

[19] D. De Donno, A. Esposito, L. Tarricone, and L. Catarinucci, "Introduction to GPU computing and CUDA programming: A case study on FDTD," *IEEE Antennas Propag. Mag.*, vol. 52, no. 3, pp. 116-122, Jun. 2010.

[20] K. H. Lee, I. Ahmed, R. S. M. Goh, E. H. Khoo, E. P. Li, and T. G. G. Hung, "Implementation of the FDTD method based on Lorentz-Drude dispersive model on GPU for plasmonics applications," *Prog. Electromagn. Res.*, vol. 116, pp. 441-456, Jul. 2011.

[21] W. Yu, X. Yang, Y. Liu, R. Mittra, D.-C. Chang, C.-H. Liao, M. Akira, W. Li, and L. Zhao, "New development of parallel conformal FDTD method in computational electromagnetics engineering," *IEEE Antennas Propag. Mag.*, vol. 53, no. 3, pp. 15-41, Jun. 2011.

[22] M. Livesey, J. F. Stack, F. Costen, T. Nanri, N. Nakashima, and S. Fujino, "Development of a CUDA implementation of the 3D FDTD method," *IEEE Antennas Propag. Mag.*, vol. 54, no. 5, pp. 186-195, Oct. 2012.

[23] J.-M. Jin and D. J. Riley, *Finite Element Analysis of Antennas and Arrays.* Hoboken, NJ: Wiley, 2008.

[24] N. Goedel, N. Nunn, T. Warbutron, and M. Clemens, "Scalability of higher-order discontinuous Galerkin FEM computations for solving electromagnetic wave propagation problems on GPU clusters," *IEEE Trans. Magn.*, vol. 46, no. 8, pp. 3469-3472, Aug. 2010.

[25] N. Goedel, N. Nunn, T. Warbutron, and M. Clemens, "Accelerating multi GPU based discontinuous Galerkin FEM computations for electromagnetic radio-frequency problems," *Appl. Comput. Electrom.*, vol. 25, no. 4, pp. 331-338, Apr. 2010.

[26] C. Potratz, H.-W. Glock, and U. Van Rienen, "Time-domain field and scattering parameter computation in waveguide structures by GPU-accelerated discontinuous-Galerkin method," *IEEE Trans. Microw. Theory Techn.*, vol. 59, no. 11, pp. 2788-2797, Nov. 2011.

[27] S. Dosopoulos and J.-F. Lee, "Discontinuous Galerkin time domain for Maxwell's equations on GPUs," in URSI International Symposium on Electromagnetic Theory (EMTS), pp. 989-991, Aug. 2010.

[28] C. Cecka, A. J. Lew, and E. Darve, "Assembly of finite element methods on graphics processors," *Int. J. Numer. Meth. Eng.*, vol. 85, no. 5, pp. 640–669, Feb. 2011.

[29] G. Markall, A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin, "Finite element assembly strategies on multi-core and many-core architectures," *Int. J. Numer. Meth. Fl.*, vol. 71, no. 1, pp. 80-97, Jan. 2013.

[30] A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski, "Finite element matrix generation on a GPU," *Prog. Electromagn. Res.*, vol. 128, pp. 249-265, 2012.

[31] D. Goeddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek, "Using GPUs to improve multigrid solver performance on a cluster," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 1, pp. 36-55, Nov. 2008.

[32] S. Georgescu and H. Okuda, "Conjugate gradients on multiple GPUs," *Int. J. Numer. Meth. Fl.*, vol. 64, no. 10-12, pp. 1254-1273, Dec. 2010.
[33] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *J. Supercomput.*, vol. 63, no. 2, pp. 443-466, Feb. 2013.
[34] R. Couturier and S. Domas, "Sparse systems solving on GPUs with GMRES," *J. Supercomput.*, vol. 59, no. 3, pp. 1504-1516, Mar. 2012.
[35] M. Naumov, "Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS," white paper, NVIDIA, Jun. 2011.
[36] H. Knibe, C. W. Oosterlee, and C. Vuik, "GPU implementation of a Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method," *J. Comput. Appl. Math.*, vol. 236, no. 3, pp. 218-293, Sept. 2011.
[37] S. Xu, W. Xue, K. Wang, and H. X. Lin, "Generating approximate inverse preconditioners for sparse matrices using CUDA and GPGPU," *J. Algorithms and Comput. Tech.*, vol. 5, no. 3, pp. 475-500, Aug. 2011.
[38] K. Xu, D. Z. Ding, Z. H. Fan, and R. S. Chen, "FSAI preconditioned CG algorithm combined with GPU technique for the finite element analysis of electromagnetic scattering problems," *Appl. Numer. Math.*, vol. 30, no. 2-3, pp. 305-340, Jun. 1999.
[39] R. Helfenstein and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," *J. Comput. Appl. Math.*, vol. 236, no. 15, pp. 3584-3590, Sept. 2012.
[40] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM, 2003.
[41] M. Benzi and M. Tuma, "A Comparative study of sparse approximate inverse preconditioners," *Finite Elem. Anal. Des.*, vol. 47, no. 4, pp. 387-393, Apr. 2011.
[42] H.-T. Meng, B.-L. Nie, S. Wong, C. Macon, and J.-M. Jin, "GPU accelerated finite element computation for electromagnetic analysis," *IEEE Antennas Propag. Mag.*, vol. 56, no. 2, pp. 39-62, Apr. 2014.
[43] Y. J. Li and J. M. Jin, "A vector dual-primal finite element tearing and interconnecting method for solving 3-D large-scale electromagnetic problems," *IEEE Trans. Antennas Propag.*, vol. 54, no. 10, pp. 3000–3009, Oct. 2006.
[44] Y. J. Li and J. M. Jin, "A new dual-primal domain decomposition approach for finite element simulation of 3-D large-scale electromagnetic problems," *IEEE Trans. Antennas Propag.*, vol. 55, no. 10, pp. 2803–2810, Oct. 2007.
[45] Y. J. Li and J. M. Jin, "Parallel implementation of the FETI-DPEM algorithm for general 3-D EM simulations," *J. Comput. Phys.*, vol. 228, no. 9, pp. 3255–3267, Feb. 2009.
[46] M.-F. Xue and J. M. Jin, "Nonconformal FETI-DP methods for large-scale electromagnetic simulation," *IEEE Trans. Antennas Propag.*, vol. 60, no. 9, pp. 4291–4305, Sept. 2012.
[47] A. Kloeckner, T. Warburton, J. Bridge, and J. S. Hesthaven, "Nodal discontinuous Galerkin methods on graphics processors," *J. Comput. Phys.*, vol. 228, no. 21, pp. 7863-7882, Nov. 2009.
[48] N. Goedel, S. Schomann, T. Warburton, and M. Clemens, "GPU accelerated Adams-Bashforth multirate discontinuous Galerkin simulation of high frequency electromagnetic fields," *IEEE Trans. Magn.*, vol. 46, no. 8, pp. 2735-2738, Aug. 2010.
[49] S. Schomann, N. Gödel, T. Warburton, and M. Clemens, "Local time-stepping techniques using Taylor expansion for modeling wave propagation with discontinuous Galerkin FEM," *IEEE Trans. Magn.*, vol. 46, no. 8, pp. 3504-3507, 2010.

[50] C. Potratz, H.-W. Glock, and U. Van Rienen, “Time-domain field and scattering parameter computation in waveguide structures by GPU-accelerated discontinuous-Galerkin method,” *IEEE Trans. Microw. Theory Techn.*, vol. 59, no. 11, pp. 2788-2797, Nov. 2011.
[51] S. Dosopoulos and J.-F. Lee, “Discontinuous Galerkin time domain for Maxwell's equations on GPUs,” 2010 URSI International Symposium on Electromagnetic Theory (EMTS), 2010, pp. 989-991.
[52] Z. Lou and J. M. Jin, “A novel dual-field time-domain finite-element domain decomposition method for computational electromagnetics,” *IEEE Trans. Antennas Propag.*, vol. 54, no. 6, pp. 1850–1862, Jun. 2006.
[53] Z. Lou and J. M. Jin, “A new explicit time-domain finite-element method based on element-level decomposition,” *IEEE Trans. Antennas Propag.*, vol. 54, no. 10, pp. 2990-2999, Oct. 2006.
[54] X. Li and J. M. Jin, “A comparative study of three finite element–based explicit numerical schemes for solving Maxwell’s equations,” *IEEE Trans. Antennas Propag.*, vol. 60, no. 3, pp. 1450-1457, Mar. 2012.
[55] X. Li and J. M. Jin, “Modeling of doubly lossy and dispersive media with the time-domain finite-element dual-field domain-decomposition algorithm,” *Int. J. Numer. Model. El., vol.* 26, no. 1, pp. 28-40, Jan. 2013.
[56] H.-T. Meng and J.-M. Jin, “Acceleration of the dual-field domain decomposition algorithm using MPI-CUDA on large-scale computing systems,” *IEEE Trans. Antennas Propag.*, vol. 62, no. 9, pp. 1-10, Sept. 2014.
[57] R. W. Boyd, *Nonlinear Optics*. Burlington, MA: Academic Press, 2008.
[58] B. Saleh and M. Tech, *Fundamentals of Photonics*. New York, NY: Wiley, 2013.
[59] M. Segev, “Optical spatial solitons,” *J. Opt. Quantum Electron.,* vol. 30, no. 7-10, pp. 503-533, Oct. 2008.
[60] G. P. Agrawal, *Nonlinear Fiber Optics.* New York: Academic, 1989.
[61] R. M. Joseph and A. Taflove. “FDTD Maxwell's equations models for nonlinear electrodynamics and optics,” *IEEE Trans. Antennas Propag.*, vol. 45, pp. 364–374, Mar. 1997.
[62] M. Fujii, M. Tahara, I. Sakagami, W. Freude, and P. Russer, “High-order FDTD and auxiliary differential equation formulation of optical pulse propagation in 2-D Kerr and Raman nonlinear dispersive media,” *IEEE J. Quantum Electron.*, vol. 40, no. 2, pp. 175–182, Feb. 2004.
[63] C. M. Reinke, A. Jafarpour, B. Momeni, M. Soltani, S. Khorasani, A. Adibi, and R. K. Lee, “Nonlinear finite-difference time-domain method for the simulation of anisotropic, $\chi^{(2)}$, and $\chi^{(3)}$ optical effects,” *J. Lightw. Technol.*, vol. 24, no. 1, pp. 624–634, Jan. 2006.
[64] A. Naqavi, M. Miri, K. Mehrany, and S. Khorasani, “Extension of unified formulation for the FDTD simulation of nonlinear dispersive media,” *IEEE Photon. Technol. Lett.*, vol. 22, no. 16, pp. 1214-1216, Feb. 2010.
[65] S. Polstyanko and J.-F. Lee, “Nonlinear hybrid vector finite element method for modeling wave propagation in nonlinear media,” *Radio Sci.*, vol. 31, no. 4, pp. 913-922, Jul.-Aug. 1996.
[66] E. Valentinuzzi, “Dispersive properties of Kerr-like nonlinear optical structures,” *J. Lightwave Technol.*, vol. 16, no. 1, pp. 152-155, Jan. 1998.

[67] M. L. Brandao and H. E. H Figueroa, "Marching finite-element schemes for nonlinear optical propagation," *Physics and Engineering of Millimeter and Sub-Millimeter Waves, 2001. The Fourth International Kharkov Symposium on*, vol. 1, pp. 154-156, 2001.

[68] A. Suryanto, M. E. van Groesen, and H. J. W. M. Hoekstra, "A finite element scheme to study the nonlinear optical response of a finite grating without and with defect," *Opt. Quant. Elec.*, vol. 35, no. 4, pp. 313-332, Mar. 2003.

[69] A. Fisher, D. White, and G. Rodrigue, "An efficient vector finite element method for nonlinear electromagnetic modeling," *J. Comput. Phys.*, vol. 225, no. 2, pp. 1331-1346, Aug. 2007.

[70] M. Sheik-Bahae and M. P. Hasselbeck, "Third order optical nonlinearities," in *Handbook of Optics, Vol. IV, Fiber Optics and Nonlinear Optics*, edited by M. Bass (McGraw-Hill, New York, 2001) 2nd ed., Chap. 17.

[71] NVIDIA, *CUDA C Programming Guide*, [Online] Sept. 2014, http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[72] A. Cevahir, A. Nukada, and S. Matsuoka, "Fast conjugate gradients with multiple GPUs," in *Proceedings of the International Conference on Computational Science*, 2009, pp. 893-903.

[73] M. Wafai, "Sparse matrix-vector multiplications on graphics processors." M.Sc. thesis, University of Stuttgart, Germany, 2010.

[74] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop, "Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation," in *Proceedings of the International Parallel and Distributed Processing Symposium Workshops*, 2012, pp. 1696-1702.

[75] D. J. Riley, Personal communication on the performance of multi-GPU acceleration of the finite-element time-domain code, 2012.

[76] M. Li, K.-P. Ma, D. M. Hockanson, J. L. Drewniak, T. H. Hubing, and T. P. Van Doren, "Numerical and experimental corroboration of an FDTD thin-slot model for slots near corners of shielding enclosures," *IEEE Trans. Electromagn. Compat.*, vol. 39, no. 3, pp. 225-232, Aug. 1997.

[77] M. Li, J. Nuebel, J. L. Drewniak, R. E. DuBroff, T. H. Hubing, and T. P. Van Doren, "EMI from airflow aperture arrays in shielding enclosures-experiments, FDTD, and MoM modeling," *IEEE Trans. Electromagn. Compat.*, vol. 42, no. 3, pp. 265-275, Aug. 2000.

[78] G. Karypis and V. Kumar, "MeTis: Unstructured graph partitioning and sparse matrix ordering system, Version 5.0" [Online]. Available: http://www.cs.umn.edu/~metis.

[79] E. Montseny, S. Pernet, X. Ferrieres and G. Cohen, "Dissipative terms and local time-stepping improvements in a spatial high order discontinuous Galerkin scheme for the time-domain Maxwell's equations," *J. Comput. Phys.*, vol. 227, no. 14, pp. 6795-6820, Jul. 2008.

[80] S. Piperno, "DGTD methods using modal basis functions and symplectic local time-stepping: application to wave propagation problems," *Eur. J. Comput. Mech.*, vol. 15, no. 6, pp. 643-670, 2006.

[81] Z. Lou and J. M. Jin, "A dual-field domain-decomposition method for the time-domain finite-element analysis of large finite arrays," *J. Comput. Phys.*, vol. 222, no. 1, pp. 408-427, Mar. 2007.

[82] J. S. Hesthaven and T. Warburton, *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications.* New York, NY: Springer, 2008.
[83] X. Li, "Investigation of explicit finite-element time-domain methods and modeling of dispersive media and 3D high-speed circuits," Ph.D. dissertation, Dept. Elect. Eng., Univ. of Illinois at Urbana-Champaign, Urbana, IL, 2012.
[84] J. S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing heterogeneous GPU computing to the computational science community," *Comput. Sci. Eng.*, vol. 13, no. 5, pp. 90-95, Sep.-Oct. 2011.
[85] Texas Advanced Computing Center - Stampede. https://www.tacc.utexas.edu/stampede/.
[86] D.-C. Chang, B.-H. Zeng, and J.-C. Liu, "Modified antipodal Fermi antenna with piecewise-linear approximation and shaped-comb corrugation for ranging applications," *IET Microwaves Antennas Propag*., vol. 4, no. 3, pp. 399-407, Mar. 2010.
[87] J. Zhang, M. Y. Koledintseva, J. L. Drewniak, D. J. Pommerenke, R. E. DuBroff, Z. Yang, W. Cheng, K. N. Rozanov, G. Antonini, and A. Orlandi, "Reconstruction of dispersive dielectric properties for PCB substrates using a genetic algorithm," *IEEE Trans. Electromagn. Compat.*, vol. 50, no. 3, pp. 704-714, Aug. 2008.
[88] J. Wang, B. Z. Wang, Y. X. Guo, L. Ong, and S. Xiao, "A compact slow-wave microstrip branch-line coupler with high performance," *IEEE Microw. Compon. Lett.*, vol. 17, no. 7, pp. 501–503, 2007.
[89] "Nonlinear optical materials," in *Encyclopedia of Materials: Science and Technology*, R. W. Boyd and G. L. Fischer, eds. (2001).
[90] S. Yan and J.-M. Jin, "Theoretical formulation of a time-domain finite element method for nonlinear magnetic problems in three dimensions," *Comput. Phys. Commun.*, under review.
[91] F. Demartini, C. H. Townes, T. K. Gustafso, and P. L. Kelley, "Self-steepening of light pulses," *Phys. Rev.*, vol. 164, no. 2, pp. 312–323, Dec. 1967.
[92] M. A. Foster, A. C. Turner, J. E. Sharping, B. S. Schmidt, M. Lipson, and A. L. Gaeta, "Broad-band optical parametric gain on a silicon photonic chip," *Nature*, vol. 441, no. 7096, pp. 960-963, Jun. 2006.