GPU-Accelerated FDTD Solver for Electromagnetic Differential Equations

MohammadReza HoseinyFarahabady$^{1[0000-0002-7851-9377]}$ and Albert Y. Zomaya$^{1[0000-0002-3090-1059]}$

The University of Sydney, School of Computer Science, Center for Distributed and High Performance Computing, Sydney, NSW, Australia
`{reza.hoseiny,albert.zomaya}@sydney.edu.au`

Abstract. Computational electromagnetics plays a crucial role across diverse domains, notably in fields such as antenna design and radar signature prediction, owing to the omnipresence of electromagnetic phenomena. Numerical methods have replaced traditional experimental approaches, expediting design iterations and scenario characterization. The emergence of GPU accelerators offers an efficient implementation of numerical methods that can significantly enhance the computational capabilities of partial differential equations (PDE) solvers with specific boundary-value conditions. This paper explores parallelization strategies for implementing a Finite-Difference Time-Domain (FDTD) solver on GPUs, leveraging shared memory and optimizing memory access patterns to achieve performance gains. One notable innovation presented in this research involves utilizing strategies such as exploiting temporal locality and avoiding misaligned global memory accesses to enhance data processing efficiency. Additionally, we break down the computation process into multiple kernels, each focusing on computing different electromagnetic (EM) field components, to enhance shared memory utilization and GPU cache efficiency. We implement crucial design optimizations to exploit GPU's parallel processing capabilities fully. These include maintaining consistent block sizes, analyzing optimal configurations for field-updating kernels, and optimizing memory access patterns for CUDA threads within warps. Our experimental analysis verifies the effectiveness of these strategies, resulting in improvements in both reducing execution time and enhancing the GPU's effective memory bandwidth. Throughput evaluation demonstrates performance gains, with our CUDA implementation achieving up to 17 times higher throughput than CPU-based methods. Speedup gains and throughput comparisons illustrate the scalability and efficiency of our approach, showcasing its potential for developing large-scale electromagnetic simulations on GPUs.

Keywords: Numerical Computational Electromagnetics · GPU Accelerators · Finite-Difference Time-Domain Solver · Partial Differential Equations · Geometric Discretization

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63751-3_24

1 Introduction

Electromagnetics, the study of electrical and magnetic fields and their interaction, has been a cornerstone technology since the twentieth century and continues to be vital in the twenty-first. Maxwell's equations are at the heart of modern electromagnetic engineering, typically solved using computational electromagnetics (CEM) [19]. CEM has undergone significant advancement in the last decade, enabling highly accurate predictions for various electromagnetic phenomena such as wave scattering analysis, radar target scattering and the precise design of antennas and microwave devices, and analysis of electromagnetic interference (EMI) and electromagnetic compatibility (EMC) issues in electronic devices [11]. Commonly used CEM methods fall into two categories: those based on *differential equation* (DE) methods and those based on *integral equation* (IE) methods, leveraging Maxwell's equations and appropriate boundary conditions. IE methods typically offer approximations using finite sums, while DE methods employ finite differences. Previously, numerical EM analysis primarily occurred in the frequency domain due to its suitability for obtaining analytical solutions and limited experimental hardware. However, recent advancements in computational resources have led to a shift towards more advanced *time-domain* CEM models, mainly focusing on DE time-domain approaches like the *finite-difference time-domain* (FDTD) method [13, 8]. The FDTD method solves problems in time while providing frequency-domain responses via Fourier transform and is applicable across a wide range of fields [4, 5, 15].

Implementing the Finite-Difference Time-Domain (FDTD) algorithm on Graphics Processing Units (GPUs) offers a promising avenue for accelerating numerical simulations [20]. GPUs, originally designed for graphics rendering, excel at parallel computation, making them well-suited for tasks like CEM models solver that involve heavy computational loads. By leveraging CUDA (Compute Unified Device Architecture), NVIDIA's parallel computing programming model, developers can harness the massive parallelism of various NVIDIA GPUs (e.g., Tesla architecture [12]) to significantly speed up FDTD computations. The implementation typically involves partitioning the space domain into smaller cells assigned to individual GPU threads. Each thread performs calculations for a specific portion of the domain in parallel with other threads. By carefully optimizing memory access patterns, developers can exploit the GPU's architecture to achieve high throughput and efficiency. One key advantage of GPU-accelerated FDTD computations is its ability to handle larger and more complex geometries with finer spatial resolution in a reasonable time frame. Such scalability is particularly beneficial for large-scale applications such as antenna design, electromagnetic compatibility analysis, and photonics research, where intricate geometries and high-fidelity computations are standard requirements. As a result, GPU-based FDTD implementations offer researchers and engineers a powerful tool for exploring and analyzing electromagnetic phenomena with unprecedented speed and accuracy.

This paper introduces an innovative approach for GPU-accelerated FDTD implementation, focusing on leveraging shared memory, optimizing memory access patterns, and minimizing divergence paths. We ensure streamlined execution by maintaining consistent block sizes and employing coalesced memory access to maximize floating-point operations per second. Additionally, we introduce key strategies such as exploiting temporal locality and breaking down computation into multiple kernels. Through rigorous experimentation, we identify optimal configurations for updating electromagnetic fields and determine the number of cells assigned to each GPU thread. This optimization effectively balances execution time reduction and GPU memory bandwidth enhancement. Furthermore, we compare the performance of our GPU implementation with a CPU-based approach using the Meep software package. Our extensive experimental analysis across various simulation sizes and configurations demonstrates substantial speedup gains and throughput improvements compared to CPU methods.

The remainder of this paper is structured as follows: Section 2 provides an overview of the background concepts relevant to our study, including Finite-Difference Time-Domain (FDTD) methods and GPU acceleration techniques. Following this, Section 3 details the methodology adopted in our study, encompassing the design and implementation of our GPU-accelerated FDTD solver. In Section 4, we outline the experimental setup used to evaluate the performance of our implementation, discussing hardware, software, parameter configurations, benchmark scenarios, and the results of our experiments. Finally, Section 5 concludes the paper, highlighting avenues for future research.

2 FDTD Framework for Numerical CEM

The FDTD method, introduced by Yee in 1966, discretizes Maxwell's equations in both space and time, enabling their solution within the time domain. Electric and magnetic field components are positioned at discrete spatial points, progressing through discrete time steps by approximating derivatives to model field evolution. In FDTD, fields are sampled at discrete time intervals, with electric and magnetic components sampled at distinct intervals, offset by $\Delta t/2$. The FDTD algorithm starts with Maxwell's time-domain equations, which are discretized using second-order accurate central difference formulas. The 3D geometry is divided into cells, forming a grid with rectangular Yee cells for stepped surface approximation. Field components within Yee cells are positioned with electric vectors at edge centers and magnetic vectors at face centers, representing Faraday's and Ampere's laws, respectively. Four electric field vectors surround each magnetic field vector, and vice versa, depicting law simulations. Field sampling in FDTD occurs at discrete time intervals, with electric components sampled at integer time intervals and magnetic components at half-integer intervals, offset by $\Delta t/2$. This necessitates spatial and temporal indices to differentiate components.

The foundation of constructing an FDTD-based CEM algorithm lies in Maxwell's time-domain equations. These differential equations describe the field behavior over time. The equations are as follows [7]:

$$\begin{aligned}\nabla \cdot \mathbf{D} &= \rho_e \\ \nabla \cdot \mathbf{B} &= \rho_m = 0 \\ \nabla \times \mathbf{E} &= -\frac{\partial \mathbf{B}}{\partial t} - \mathbf{M} \\ \nabla \times \mathbf{H} &= \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}\end{aligned}$$

Here, $\mathbf{E}$ represents the electric field strength vector (in Volts per meter), $\mathbf{D}$ represents the electric displacement vector (in Coulombs per square meter), $\mathbf{H}$ represents the magnetic field strength vector (in Amperes per meter), $\mathbf{B}$ represents the magnetic flux density vector (in Webers per square meter), $\mathbf{J}$ represents the electric current density vector (in Amperes per square meter), $\mathbf{M}$ represents the magnetic current density vector (in Volts per square meter), ρ_e represents the electric charge density (in Coulombs per cubic meter), and ρ_m represents the magnetic charge density (in Webers per cubic meter), equal to zero everywhere. Additionally, constitutive relations complement Maxwell's equations to characterize the material media [7]. Constitutive relations for linear, isotropic, and non-dispersive materials can be expressed as $\mathbf{D} = \epsilon\mathbf{E}$ and $\mathbf{B} = \mu\mathbf{H}$, where ϵ and μ represent the permittivity (in Farad/meter) and the permeability (in Henry/meter) of the material.

When deriving Finite-Difference Time-Domain (FDTD) equations, we can focus on the curl equations as the divergence equations can be fulfilled by the developed FDTD updating equations [7, 19]. The electric current density $\mathbf{J}$ comprises the sum of the conduction current density $\mathbf{J}_c = \sigma_e\mathbf{E}$ and the impressed current density $\mathbf{J}_i$ such that $\mathbf{J} = \mathbf{J}_c + \mathbf{J}_i$. For the magnetic current density $\mathbf{M}$, we have $\mathbf{M} = \mathbf{M}_c + \mathbf{M}_i$, where $\mathbf{M}_c = \sigma_m\mathbf{H}$. Here, σ_e represents the electric conductivity in Siemens per meter, and σ_m represents the magnetic conductivity in Ohms per meter. By decomposing the current densities into conduction and impressed components and employing the constitutive relations, we can rewrite Maxwell's curl equations as:

$$\begin{aligned}\epsilon\frac{\partial \mathbf{E}}{\partial t} &= \nabla \times \mathbf{H} - \sigma_e\mathbf{E} - \mathbf{J}_i, \\ \mu\frac{\partial \mathbf{H}}{\partial t} &= -\nabla \times \mathbf{E} - \sigma_m\mathbf{H} - \mathbf{M}_i.\end{aligned}$$

This formulation treats only the electromagnetic fields $\mathbf{E}$ and $\mathbf{H}$. All four constitutive parameters ε, μ, σ_e, and σ_m are the input parameters so that any linear isotropic material can be specified. Treatment of electric and magnetic sources is included through the impressed currents. Each vector equation can be further decomposed into three scalar equations for three-dimensional space, *i.e.,* $\mathbf{E} = (\mathbf{E}_x, \mathbf{E}_y, \mathbf{E}_z)$ and $\mathbf{H} = (\mathbf{H}_x, \mathbf{H}_y, \mathbf{H}_z)$.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63751-3_24

Central Difference Approximation Schemes

The central difference formula approximates derivatives through finite differences and is a foundational technique within numerical analysis, especially concerning the resolution of differential equations. These formulas estimate a function's derivative at a specific point by assessing the function at neighboring points. Specifically, for a first-order derivative, the central difference formula calculates the disparity between function values at symmetrically positioned points around the point of interest and divides by the spacing between these points. This method offers several benefits, including straightforward implementation and relatively high accuracy compared to alternative finite difference approaches. Nonetheless, it's crucial to recognize that the selection of grid spacing and the order of the difference formula can significantly influence the accuracy and stability of the numerical solution.

The central difference formula to approximate the derivative $f'(x)$ involves averaging the forward and backward difference formulas. This technique yields:

$$f'(x) = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} - \frac{(\Delta x)^2}{6} f''(x) + O(\Delta x^2)$$

Therefore, the central difference formula for $f'(x)$ exhibits second-order accuracy. This level of accuracy implies that the dominant term in the error introduced by a second-order accurate formula is proportional to the square of the sampling period. For example, halving the sampling period reduces the error by a factor of four. Consequently, a second-order accurate formula like the central difference formula provides greater precision than a first-order accurate formula.

Updating Procedures in FDTD Method

The FDTD method, introduced by Yee in 1966 [21], discretizes Maxwell's equations in both space and time, enabling their solution in the time domain. It places electric and magnetic field components at discrete spatial points within a grid, advancing through discrete time steps to simulate field evolution. In FDTD, fields are sampled at discrete time instants, with electric and magnetic components sampled at different intervals offset by $\Delta t/2$. While the algorithm calculates the fields at discrete time points, electric and magnetic components are not sampled simultaneously. Electric field components are sampled at integer time steps $(0, \Delta t, 2\Delta t, \ldots)$, while magnetic field components are sampled at half-integer time steps $(\frac{1}{2}\Delta t, \left(1+\frac{1}{2}\right)\Delta t, \ldots)$, introducing an offset of $\Delta t/2$ between them. Figure 1 showcases a single Yee cell within the grid utilized in the FDTD method.

As outlined previously, the FDTD updating process approximates derivatives using the central difference formula. Here, field components are referenced by spatial and temporal indices, denoted with superscript notation. For example, $\mathbf{E}_z^n(i,j,k)$ represents the z component of an electric field vector sampled at time instant $n\Delta t$, positioned at $((i-1)\Delta_x, (j-1)\Delta_y, (k-0.5)\Delta_z)$. Similarly, $\mathbf{H}_y^{n+\frac{1}{2}}(i,j,k)$ represents the y component of a magnetic field vector positioned

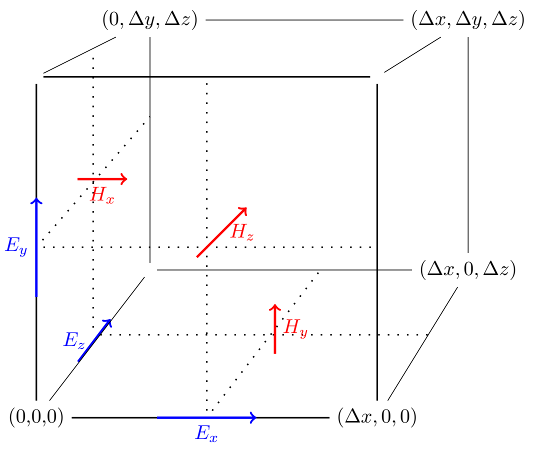

$(0, \Delta y, \Delta z)$
$(\Delta x, \Delta y, \Delta z)$
H_x
E_y
H_z
$(\Delta x, 0, \Delta z)$
E_z
H_y
(0,0,0)
$(\Delta x, 0, 0)$
E_x

Fig. 1: Arrangement of electric and magnetic field vector components within a single Yee cell (Inspired by the diagrams in [19, 22]).

at $((i-0.5)\Delta_x, (j-1)\Delta_y, (k-0.5)\Delta_z)$ and sampled at time instant $\left(n+\frac{1}{2}\right)\Delta t$. Remarkably, for a computational domain with the number of cells (M_x, M_y, M_z), the total spatial problem size is $M = M_x \times M_y \times M_z$. Hence, The storage space required for this is approximately $24\times M$ bytes for 32-bit precision and $48\times M$ bytes for 64-bit precision [1].

3 Our Proposed Implementation of the FDTD Method

Recent advancements in GPU accelerators promise efficient numerical method implementations, significantly boosting the computational efficiency of EM solvers. Ongoing progress in GPU technology, with increased computational power, memory bandwidth, and specialized hardware features like tensor cores, continues to advance numerical simulations. This section emphasizes designing and implementing a GPU-accelerated FDTD solver, showcasing its effectiveness in solving EM equations with enhanced computational speed. This paper primarily focuses on NVIDIA's GPU platform [17] and the associated CUDA programming model [16, 6]. While a similar approach can be developed for GPUs supporting the OpenCL standard, we center our discussion on NVIDIA's technology. In the following section, we provide readers with a concise overview of modern GPU processor architecture, including its memory hierarchy and the programming model utilized to map thread blocks onto the GPU streaming processors effectively. This foundational knowledge is essential for readers to grasp the design considerations before implementing the FDTD method on modern GPUs.

GPU Architecture and CUDA Programming Model

A GPU comprises Streaming Multiprocessors (SMs) containing processing cores and shared low-latency memory. SMs execute kernel functions in parallel with variable core counts. Global DRAM memory is partitioned and managed by an interconnection network for read/write requests. The speed difference between shared and global memory motivates minimizing data transfers for efficiency. Thread blocks can use local/shared memory as a manual cache, compensating for the lack of automatic caching in the older models of NVIDIA GPUs. Accessing shared memory is faster than global memory, mitigating latency with numerous threads emphasizing arithmetic operations.

CUDA, developed by NVIDIA, allows C/C++ code to run directly on GPUs, exploiting parallel processing. Programs consist of host code on the CPU and device code on the GPU, organized into kernel functions executed by multiple threads in parallel. CUDA manages parallelism and memory, with barrier instructions ensuring synchronized operations among threads. CUDA streams allow asynchronous execution of commands on NVIDIA GPUs, enabling concurrent operations and facilitating overlap of computation with data transfers [16]. Conditional branching significantly impacts stream performance, particularly in kernels with numerous threads intended to hide global memory access latency, which can range from 400 to 600 cycles [2].

Parallelization Strategies of FDTD Implementation

The field-updating kernel's parallelization in CUDA is feasible since the update equation is independently applied at each Yee's cell at every time step. In traditional CPU-based computations, the sequential nature of processing limits the speed and scalability of simulations involving large-scale electromagnetic problems. However, the parallelization of the field-updating kernel in CUDA is highly advantageous due to the inherent independence of the update equation at each Yee cell and time step. This independence allows for efficient exploitation of parallel processing capabilities offered by NVIDIA GPUs. CUDA enables the execution of thousands of threads in parallel on GPU cores, thereby significantly accelerating the computation process.

CUDA harnesses the massive parallelism of GPU architectures to update multiple Yee cells simultaneously at each time step by distributing the workload across multiple threads. This strategy significantly reduces the computational time needed for complex electromagnetic simulations. The CUDA programming model also offers developers fine-grained control over thread management and memory allocation. This level of control guarantees efficient parallelization of the field-updating kernel, leading to substantial performance enhancements compared to traditional CPU-based methods.

To optimize performance on GPUs, it is crucial to reduce global memory access latency. This entails following specific architectural guidelines, such as leveraging shared memory to minimize latency. We employ strategies like exploiting temporal locality and avoiding misaligned global memory accesses to reduce additional memory transactions. Additionally, minimizing divergence paths, where threads within a warp take different control flow paths, ensures efficient data processing. Furthermore, breaking down the computation process into multiple kernels, each focusing on computing different components of the EM field, can enhance shared memory utilization and GPU cache efficiency, which is the approach we have adopted in this work.

As highlighted below, we have implemented several crucial design decisions and optimizations to exploit the GPU's parallel processing capabilities fully.

- For executing the field-updating kernels, we maintain a consistent number of blocks per grid. This approach ensures that the workload is evenly distributed across the GPU cores, preventing the overloading of individual GPU cores and minimizing idle resources. This approach also simplifies memory management and synchronization, as each block operates independently without dependencies on other blocks. This enables seamless coordination between threads within the same block, facilitating efficient execution of the field-updating kernels.
- We conduct a thorough experimental analysis across various block sizes to ascertain the optimal configurations for all kernels that update the electric and magnetic fields. This evaluation involves systematically varying the number of threads per block and the total number of blocks per grid to explore the performance impact on different GPU architectures and computational workloads. Our experimental methodology includes profiling each kernel's memory access patterns and computational intensity to gain insights into their performance characteristics under varying block configurations. This analysis helps us understand how different block sizes affect memory bandwidth usage, register pressure, and arithmetic throughput, enabling us to uncover the most efficient block configurations that balance computational efficiency, memory bandwidth utilization, and resource utilization on the GPU.
- The most effective performance is consistently achieved with a block size of $B_1 \times B_2$, where B_1 is set to a relatively large value, such as greater than 256, on our GeForce 4070 Ti GPU. Meanwhile, B_2 remains relatively small, for example, less than 8. This configuration optimizes the utilization of the GPU's resources while minimizing potential overhead. A large value for B_1 allows for a high degree of parallelism within each block, enabling efficient utilization of the GPU's streaming multiprocessors (SMs) and maximizing the number of threads running concurrently. On the other hand, keeping B_2 small helps mitigate potential memory contention and resource conflicts within each block. With a smaller B_2, the threads within a block have access to a more localized and efficient shared memory space, reducing the likelihood of memory access conflicts and improving overall memory access latency.
- Such configurations consistently yield the shortest execution times for field-updating kernels, thereby maximizing the number of floating-point opera-

tions per second (FLOPS) for each kernel. By meticulously optimizing the block size parameters, we ensure that the GPU's computational resources are fully utilized, enabling the kernels to achieve peak performance. The reduction in execution times directly translates to a higher throughput of FLOPS, as more operations can be completed within a given time frame. This enhanced efficiency allows for faster simulation runs and enables us to tackle larger and more complex electromagnetic problems within a reasonable timeframe.

– In our CUDA implementation, data arrays are typically accessed from global memory, while integer constants are accessed from constant memory. To optimize the usage of global memory, we employ coalesced access strategy. Coalesced access involves ensuring that consecutive threads within a thread block access consecutive memory addresses when reading or writing data. This approach allows for the efficient transfer of whole data sets in a single transaction, maximizing memory bandwidth utilization and reducing memory access latency. By arranging threads to access contiguous memory locations, we can streamline memory transfers and minimize the number of transactions required to load or store data in shared memory. This, in turn, enhances memory access efficiency and the overall performance of memory-bound kernels.
– In our CUDA implementation, we adopt a one-dimensional (1D) indexing scheme to define block and thread numbers, aiming to optimize memory access patterns and enhance overall performance. By organizing blocks and threads in a 1D manner, we facilitate memory coalescing and streamline access operations, both for reading data from and writing data to global memory. The decision to use 1D indexing is rooted in its effectiveness in maximizing memory coalescing. With 1D indexing, consecutive threads within a block access contiguous memory locations, enabling efficient data transfer in a single transaction and minimizing memory access latency. Furthermore, *column-major* order ensures that data elements along the same column are stored contiguously in memory, aligning with the memory access patterns of CUDA kernels. This organization is particularly advantageous during computations involving a time-consuming partial FDTD solver, where efficient memory access is essential for performance optimization.
– The strategy we have adopted involves assigning multiple Yee's cells to each thread, enabling direct access from shared memory for neighboring values within each warp. This approach minimizes memory access overhead by reducing the frequency of global memory accesses per thread. By allowing inner threads of a warp block to read from shared memory instead of global memory, our strategy eliminates the need for additional global memory reads within the warp. Moreover, when data is arranged in global memory, our strategy leverages the on-chip caching mechanism in the latest NVIDIA GPUs, further optimizing memory access and reducing latency. To determine the most effective performance, we conducted extensive experimental analysis across various sizes of cells per thread computation. For example, in our in-house system, we have identified that the optimal value for the

cell size per thread computation is 8. This optimal configuration maximizes the efficiency of thread utilization and memory access, resulting in improved performance and overall computational throughput.

4 Results

This section presents the results of the electromagnetic differential equation solutions analysis conducted using the developed GPU-accelerated FDTD method. We begin by showcasing the experimental results designed to validate the implemented FDTD method through two simple scenarios. Subsequently, we present the performance evaluation results, comparing them to the outcomes of other FDTD implementations running on the CPU.

Platform Setup: The numerical simulations described in this study were conducted on a desktop computer featuring an Intel Core i7-13700 CPU, boasting 24 cores and 64GB of memory, alongside a single NVIDIA GTX 4070 Ti Graphics card equipped with *8GB* of global memory. The desktop operates on Ubuntu 22.04, and the kernels were developed using the CUDA Toolkit v12.2.

Accuracy Assessment

To validate the numerical solver, we compared our GPU-implemented FDTD method and other FDTD implementations primarily running on the CPU. This evaluation was performed using two distinct scenarios, as outlined below.

Scenario 1: In our evaluation, we simulated the reflection coefficient of a frequency-selective surface composed of a dipole array at normal incidence. Using a 3D FDTD method within free space, the dipole array was arranged in the XY plane with a size of 8 mm and evenly spaced intervals of 30 mm in all X, Y, and Z directions. The FDTD simulation was executed twice: once on the CPU using the `Meep` software package [18] – an open-source FDTD implementation for electromagnetism simulation on CPU many-core processors – and once on the GPU using our proposed implementation. Both simulations employed mesh sizes of 0.1 mm in each direction. To ensure accurate results, we applied boundary conditions at the regional boundary. Additionally, we set the time step to $\Delta t = 0.004$ ns to maintain the stability and convergence of the FDTD method. This approach enabled us to assess the performance and effectiveness of our implementation precisely. We calculated the 250-time step E and H fields in each direction under identical parameter settings by comparing the results obtained from both the Meep implementation and our proposed method. The simulation results revealed consistency between the two methods, with a maximum error of 0.8% in computational differences.

Scenario 2: We further conducted a second experimental evaluation to calculate the Radar Cross Section (RCS) of a Perfect Electric Conductor (PEC) sphere, which serves as a fundamental benchmark for electromagnetic simulation methods due to its well-understood analytical solution. Using our proposed implementation of the FDTD method, we simulated the electromagnetic scattering from a sphere to compare the computed RCS values against the analytically derived RCS values. The simulation setup consisted of a cubic domain with a PEC sphere at its center, spanning 25 cells in radius within a calculation domain of $600 \times 600 \times 600$ cells. The domain was filled with free-space, and the temporal resolution was set to $\Delta t = 0.004$ ns. The sphere, with a radius of $r = 0.2\lambda$, was subjected to an incident electric field at 0 degrees in the x direction and had PEC boundaries with Perfect Magnetic Conductor (PMC) boundary conditions. Throughout the simulation of 300 steps, we computed the absorption efficiency of the sphere and meticulously compared it with analytical results. We observed a degree of agreement between the two methods exceeding 99.4%.

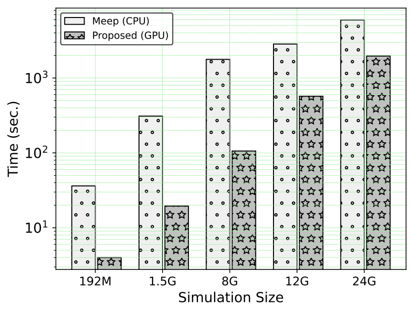
Efficiency Evaluation of GPU-implemented FDTD Method

In this section, we compare the performance of our proposed GPU massive parallel implementation of the FDTD method with the parallel vectorized CPU implementation using the `Meep` software package for the two scenarios mentioned above. While there are many other GPU implementations of the FDTD method in the literature, such as those referenced in [14, 9, 20, 10, 3], the source code for various other GPU implementations is not publicly available. Hence, direct comparisons of performance evaluations between our method and these implementations are not feasible in this study.

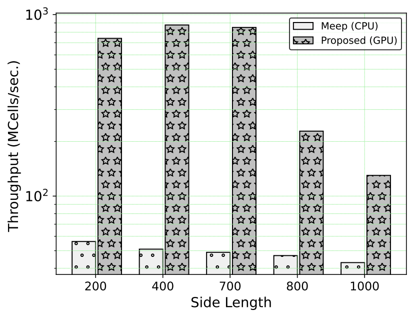
Speedup Gain: Figure 2 (a) plots the speedup of our implementation compared to the `Meep` implementation on CPU cores. The GPU version's speedup demonstrates our approach's potential and the consistency of such speedup as the simulation size increases. It is evident that GPU implementation is more efficient when more extensive simulations are accommodated in GPU global memory. However, when the simulation space exceeds the capacity of the GPU global memory (*i.e.,* more than 8GB in our setup), the speedup declines. Such a decline is attributed to the challenges in hiding the context switching with a small number of active warps, compounded by the mandatory data transfer between the main memory in the host and the GPU global memory per each simulation step.

Furthermore, maximizing the concurrent execution of a large number of threads on the GPU proves to be more efficient, given that the primary bottleneck for FDTD computation lies in the availability of data in the global/shared memory of the GPU. Additionally, the cache memory of the GPU plays a non-significant role in influencing speedup performance. For smaller simulation sizes, even where data fits within the SM caches, the speedup can reach values near 8x, as for smaller simulation sizes; the time costs of this memory management become more significant with respect to the whole execution time as the communication time of data movement between host and device is dominated. Similarly, as the simulation size increases beyond the capacity of GPU global memory, the occurrence of communication time between host and device becomes more significant again. This results in heightened communication time between host and device memory, as well as the global and shared memory within the GPU, consequently leading to severe degradation in performance. Consequently, the speedup diminishes significantly to as low as 2x when the simulation size surpasses the GPU's global memory capacity by one order of magnitude.

Throughput: Throughput refers to the number of finite difference cells updated per second. Figure 2 (b) plots the average performance throughput (in Millions of Cells per Second) and the throughput gain of our implementation compared to the CPU-based `Meep` implementation as the side length of the cubic domain increases from 200 to 1000. This calculation excludes the wasted communication time transferring data from the host's main memory to GPU global memory. Results show that the throughput of the CPU-based `Meep` solver of the FDTD code remains constant at approximately 50 Mega Cells/s, while our CUDA implementation ranges from 130 to 877 Mega Cells/s, maintaining around 100 when the simulation size exceeds the capacity of the GPU's global memory by one order of magnitude. Furthermore, our method demonstrates a 17x higher throughput performance than the CPU-based method when the entire simulation data fits into the GPU global memory. The consistent speedup and throughput gain trend observed in Scenario 2 experiments across varying simulation sizes aligns with earlier reported results. Therefore, we omit the repetition of these findings.



(a) Speedup Gain



(b) Solver Throughput (MegaCells/sec.)

Fig. 2: Comparative analysis of (a) speedup gain, and (b) throughput with increasing simulation size: our GPU implementation versus `Meep` (CPU) method. Evaluation conducted with varying model sizes and a consistent time step in FDTD simulations for Scenario 1.

5 Conclusions

This paper illuminates the promising potential of employing GPU-accelerated FDTD methods to implement large-scale PDE solvers. By harnessing advanced features of the CUDA framework, such as CUDA streams, we have developed a GPU-accelerated FDTD solver and extensively evaluated its performance across NVIDIA's GPUs. Comparative analyses against the parallelized CPU solver have revealed substantial performance advantages. Notably, our GPU solver achieved throughput rates of up to 877 Mega Cells per second, marking up to a 17-fold improvement over the open-source Meep CPU solver on standard desktops. This affordability renders cutting-edge GPU technology accessible to a wide range of individuals utilizing commodity workstations in partial differential equations solver. While GPUs inherently offer superior computational performance compared to traditional CPUs, owing to their heightened floating-point capabilities and memory bandwidth, future investigations will explore alternative FDTD implementations tailored for diverse contexts alongside their scalability on multi-GPU cluster platforms.

Acknowledgments. Professor Zomaya would like to acknowledge the support of the Australian Research Council Discovery Project (DP200103494). Dr. M. Reza Hoseiny-Farahabady acknowledges the continued support of The Center for Distributed and High-Performance Computing at The University of Sydney for giving access to advanced high-performance computing and cloud facilities, digital platforms, and necessary tools.

References

1. Andersson, U.: Time-Domain Methods for the Maxwell Equations. Doctoral dissertation, Royal Institute of Technology, Stockholm (2001)
2. Ansorge, R.: Programming in Parallel with CUDA: A Practical Guide. Cambridge University Press (2022)
3. Baumeister, P.F., Hater, T., Kraus, J., Pleiter, D., Wahl, P.: A performance model for gpu-accelerated fdtd applications. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC). pp. 185–193 (2015). https://doi.org/10.1109/HiPC.2015.24
4. Carcione, J.M., Valle, S., Lenzi, G.: Gpr modelling by the fourier method: improvement of the algorithm. Geophysical Prospecting **47**(6), 1015–1029 (1999)
5. Cassidy, N.J., Millington, T.M.: The application of finite-difference time-domain modelling for the assessment of gpr in magnetically lossy materials. Journal of Applied Geophysics **67**(4), 296–308 (2009)
6. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Morgan Kaufmann (November 2012)
7. Elsherbeni, A., Demir, V.: The Finite-difference Time-domain for Electromagnetics: With MATLAB Simulations. ACES series on computational electromagnetics and engineering, Institution of Engineering and Technology (2016)
8. Feng, D.S., Dai, Q.W.: Gpr numerical simulation of full wave field based on upml boundary condition of adi-fdtd. NDT & E International **44**(6), 495–504 (2011)

9. Francés, J., Bleda, S., Neipp, C., Márquez, A., Pascual, I., Beléndez, A.: Performance analysis of the fdtd method applied to holographic volume gratings: Multi-core cpu versus gpu computing. Computer Physics Communications **184**(3), 469–479 (2013)
10. Francés, J., Otero, B., Bleda, S., Gallego, S., Neipp, C., Márquez, A., Beléndez, A.: Multi-gpu and multi-cpu accelerated fdtd scheme for vibroacoustic applications. Computer Physics Communications **191**, 43–51 (2015)
11. Giannakis, I., Giannopoulos, A., Warren, C.: A realistic fdtd numerical modeling framework of ground penetrating radar for landmine detection. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing **9**(1), 37–51 (2016)
12. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: Nvidia tesla: a unified graphics and computing architecture. IEEE Micro **28**(2), 39–55 (2008)
13. Liu, J., Shen, L.: Numerical simulation of subsurface radar for detecting buried pipes. IEEE Transactions on Geoscience and Remote Sensing **29**(5), 795–798 (1991)
14. Livesey, M., Stack, Jr, J.F., Costen, F., Nanri, T., Nakashima, N., Fujino, S.: Development of a cuda implementation of the 3d fdtd method. IEEE Antennas and Propagation Magazine **54**(5), 186–195 (2012)
15. Lopez, J., Carnicero, D., Ferrando, N., Escolano, J.: Parallelization of the finite-difference time-domain method for room acoustics modelling based on cuda. Mathematical and Computer Modelling **57**(7), 1822–1831 (2013)
16. NVIDIA Corporation: NVIDIA CUDA Toolkit Documentation, CUDA Streams: Asynchronous Concurrent Execution (2023), accessed on Nov. 2023
17. NVIDIA Corporation: NVIDIA's GPU Platform (2023), accessed on Nov. 2023
18. Oskooi, A., Roundy, D., Ibanescu, M., Bermel, P.A., Joannopoulos, J.: Meep: A flexible free-software package for electromagnetic simulations by the fdtd method. Computer Physics Communications **181**(3), 687–702 (2010)
19. Taflove, A.: Computational Electrodynamics: The Finite-Difference Time-Domain Method. Artech House, Norwood, MA, USA, 3rd edn. (2005)
20. Warren, C., Giannopoulos, A., Gray, A., et al.: A cuda-based gpu engine for gprmax: Open source fdtd electromagnetic simulation software. Computer Physics Communications **237**, 208–218 (2019)
21. Yee, K.S.: Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. IEEE Trans. Antennas Propagat. **14**(3), 302–307 (March 1966)
22. Yu, W., Yang, X., Liu, Y., Mittra, R., Muto, A.: Advanced FDTD Method: Parallelization, Acceleration, and Engineering Applications. Artech House, Norwood, MA (2011)