Acceleration of Solving Maxwell’s Equations Using Cluster of GPUs

E. Arianyan, S. A. Motamedi, M. Hekmatpanah and I. Arianyan

Abstract—Finite difference time domain (FDTD) is a numerical method for solving differential equations like Maxwell’s equations. Normally, simulation time of these equations is very long and there has been a great effort to reduce it. The most recent and useful way to reduce the simulation time of these equations is through using GPUs. Graphical processing units (GPUs) are powerful hardware which have parallel architectures and are suitable for running parallel applications. In this paper we evaluate three different configurations for implementing FDTD on GPU: one single GPU system, cluster of GPUs on the same system, and cluster of GPUs on different systems. We present the problems of these implementations and how to solve them. We apply some methods to solve data divergence and data conflict problems which lead to 1.6-times increase in speed. Moreover, we devise a new overlap algorithm to run FDTD computations on discrete memories which results in 5-times increase in speed. Furthermore, we divide data in the right direction of memory which causes 1.6-times increase in speed. The speed of running FDTD, on our most powerful cluster system based on GPU, has increased by a factor of 40 compared to CPU cluster.

Index Terms—CUDA, Electric field, FDTD, GPU, Magnetic field, PDE.

1 INTRODUCTION

Solving Maxwell’s equations is a routine procedure in electromagnetic simulators.

Time to market of new applications in this area depends on the simulation time of these equations which can result in faster capturing of the market among other competitive companies. Graphical processing unit (GPU) is a parallel hardware capable of executing enormous computations on its parallel, many-core structure. GPU first appeared on the market for only graphics rendering applications. The emergence of new APIs and high level programming languages like compute unified device architecture (CUDA) simplified the process of harnessing the horsepower of GPU. It is now used in versatile applications like scientific and engineering applications.

The parallel architecture of GPU is appropriate for implementing single instruction multiple data (SIMD) type of applications in which the same instruction is carried out simultaneously for a massive amount of data. Another important point in executing applications on GPU is the amount of parallelism in the application itself. More precisely, if the nature of application is not parallel in a way that the functions should be executed serially, we may achieve very little speed improvement running our algorithm on GPU.

GPUs are evolving at an astonishing rate. They are currently capable of delivering over 1 TFLOPS of single precision performance and over 300 GFLOPS of double precision while executing up to 240 simultaneous threads in one low-cost package [1]. As a consequence of this huge computational power, GPUs have become powerful tools for high performance computing, achieving 2-100 times the speed of their x86 counterparts in various applications [1].

Maxwell’s equations are used in computations of the electromagnetic simulators [2]. These computations require solving of partial differential equations (PDEs) which are vastly used in scientific computations and engineering applications. Since these equations cannot be solved analytically, numerical methods like FDTD are used to solve them [3]. Great efforts have been put into solving of these equations which have huge computations as fast as possible. Since the computations of the electric and magnetic fields are dependant, they should be updated concurrently on parallel structures [4]. Thus, by implementing them on parallel-structured GPUs, we can gain large speedup.

Many researchers have presented valuable works on implementing FDTD algorithm on GPUs. Ong et al. proposed a scalable GPU cluster solution for the acceleration of FDTD for large-scale simulations [5]. To illustrate the speed performance of the cluster, they presented the simulation results of a cubic resonator with PEC boundaries. They reported that FDTD computations on Accelerware’s G80 GPU cluster are 25 to 29 times faster than CPU implementation.

In another work, Ong et al. presented a basic method for parallelizing the FDTD algorithm and explore some applications that have been enabled by this technology [2]. They reported up to 34 speedup for implementing FDTD on GPU instead of CPU.

Inman et al. described the practical addition of a
CPML absorbing boundary to a three dimensional GPU accelerated FDTD code [6]. They presented the results of simulations with dielectric and conducting objects in the computational domain. They reported 6 times speedup in their charts.

Takada et al. proposed a high-speed FDTD algorithm for GPU [7]. Their algorithm includes two important techniques: coalesced global memory access on a GPU board and the improved cache block algorithm for GPU and achieves an approximately 20fold improvement in computational speed compared with a conventional CPU.

Liuge et al. analyzed parallel FDTD method and CUDA architecture and presented a GPU based implementation of three-dimensional FDTD which is solved by two-dimensional grid of threads and extra shared memory is used in their application for optimal memory accessing [8]. They reported tens of times speedup with a GT200 GPU as coprocessor compared with traditional PC computation.

Vveysel et al. presented an implementation of FDTD method using CUDA [9]. They presented a thread-to-cell mapping algorithm. They reported that their code processes about 450 million cells per second on the average.

In this paper based on our previous works in [1] and [10], we evaluate three configurations for implementing FDTD computations: one single GPU, a cluster of GPUs on different machines, and a cluster of GPUs on the same machine. We state the problems we faced during implementation of FDTD on these platforms and suggest some solutions to solve these drawbacks so that the speed of running our algorithms on these platforms improves.

We start our work with redefining FDTD equations to ones that are suitable for discrete memories computations. Moreover, we clarify the notations we use when we map a three dimensional data space onto a one dimensional memory space. Next, we find methods to accelerate the execution of FDTD algorithm regardless of the hardware it is being executed on. Confirming that the algorithm is well tuned, we suggest methods by the help of which the speed of executing FDTD algorithm on our hardware increases. We use CUDA technology to implement FDTD on GPU.

The remainder of this paper is organized as follows. In section 2, we briefly introduce Maxwell’s equations and FDTD algorithm. In section 3, we discuss using CUDA for programming on GPUs. In section 4, we evaluate implementation of FDTD on a single GPU. In section 5, we present implementation of FDTD on a cluster of GPUs. In section 6, we explain our experimental results. Finally, we conclude the paper in section 7.

2 Maxwell’s Equations and FDTD

2.1 Maxwell’s Equations

The goal of solving Maxwell’s equations with FDTD method is both making the routinely unsolvable equation of Maxwell solvable, and making the equations implementable on parallel structures like GPUs. The Maxwell’s equations are represented with (1) and (2).

\[
\nabla \times E(i, j, k, t) = -\mu \frac{\partial H(i, j, k, t)}{\partial t} \tag{1}
\]

\[
\nabla \times H(i, j, k, t) = +\epsilon_0 \sigma_r \frac{\partial E(i, j, k, t)}{\partial t} + \sigma E(i, j, k, t) \tag{2}
\]

Where \( E \) represents electrical field, \( H \) represents magnetic field, \( \epsilon \) represents permittivity, \( \mu \) represents permeability, and \( \sigma \) represents conductivity. For brevity, we show only the magnetic field in \( x \) dimension, \( H_x \), and electric field in \( z \) direction, \( E_z \), by (3) and (4) respectively, which are obtained by extracting the equations in Cartesian adapt.

\[
\frac{\partial H_x(i, j, k, t)}{\partial t} = \frac{1}{\mu_0 \mu_r} \left( \frac{\partial E_y(i, j, k, t)}{\partial \xi} - \frac{\partial E_z(i, j, k, t)}{\partial \eta} \right) \tag{3}
\]

\[
\frac{\partial E_z(i, j, k, t)}{\partial t} = \frac{1}{\epsilon_0 \sigma_r} \left( \frac{\partial H_y(i, j, k, t)}{\partial \xi} - \frac{\partial H_z(i, j, k, t)}{\partial \eta} \right) - \frac{\sigma}{\epsilon_0 \sigma_r} E_z(i, j, k, t) \tag{4}
\]

These equations accompanying the ones for \( H_y \), \( H_z \), \( E_x \), and \( E_y \) should be solved for a variety of materials having different relative permeability and permittivity, and also for different sources of electric or magnetic fields.

2.1 FDTD Method

When the variation of independent parameter \( \Delta x \) is small, we can estimate the derivative of function \( f(x) \) with (5).

\[
\frac{df(x)}{dx} \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{5}
\]

So, based on the approximation in (5), we can change the \( \frac{\partial}{\partial \xi}, \frac{\partial}{\partial \eta}, \frac{\partial}{\partial \zeta} \) to simple subtraction operators and rewrite (3) and (4) as (6) and (7).

\[
\frac{H_x(t + \Delta t) - H_x(t)}{\Delta t} = \frac{1}{\mu_0 \mu_r} \left( \frac{H_y(z + \Delta z) - H_y(z)}{\Delta z} - \frac{H_z(y + \Delta y) - H_z(y)}{\Delta y} \right) \tag{6}
\]

\[
\frac{H_x(t + \Delta t) - H_x(t)}{\Delta t} = \frac{1}{\epsilon_0 \sigma_r} \left( \frac{H_y(x + \Delta x) - H_y(x)}{\Delta x} - \frac{H_x(y + \Delta y) - H_x(y)}{\Delta y} - \frac{\sigma}{\epsilon_0 \sigma_r} E_z(x, y, z, t) \right) \tag{7}
\]

We should divide our problem space into small areas and repeat the computations of components of magnetic and electric fields for each time step and for all of these areas.

Since we want to execute our computations on GPU, we should assign a space for each of these three dimensional areas in memory space of GPU, which is done by mapping any three dimensional function like \( f(i, j, k) \) to a one dimensional function \( f(n) \) in memory, using a for loop like one that is depicted in algorithm 1.

As a result, we change the FDTD equations by replac-
ing variable n instead of (i,j,k) in our formulas. Hence, if we reformulate the partial differential equations to (8), (9), (10), (11), (12), and (13), then we can rewrite the Hx and Ez equations like (14) and (15). Fig. 1 depicts the mapping strategy. It shows that we map the three dimensional input data onto linear memory space in the order of x, y, and z directions respectively.

**Algorithm 1.** Mapping input data to memory space

```c
for(k=0; k<nz; k++){
    for(j=0; j<ny; j++){
        for(i=0; i<nx; i++){
            n=(k*nx*ny)+(j*nx)+i;
            f[n]="data";
        }
    }
}
```

\[
\frac{\partial E}{\partial x} = \frac{(E[n+1] - E[n])}{\Delta x}
\]  
(8)

\[
\frac{\partial H}{\partial x} = \frac{(H[n+1] - H[n])}{\Delta x}
\]  
(9)

\[
\frac{\partial E}{\partial y} = \frac{(E[n + nx] - E[n])}{\Delta y}
\]  
(10)

\[
\frac{\partial H}{\partial y} = \frac{(H[n + nx] - H[n])}{\Delta y}
\]  
(11)

\[
\frac{\partial E}{\partial z} = \frac{(E[n + nx*ny] - E[n])}{\Delta z}
\]  
(12)

\[
\frac{\partial H}{\partial z} = \frac{(H[n + nx*ny] - H[n])}{\Delta z}
\]  
(13)

\[
Hx[n]\text{new} = \frac{\Delta t}{\mu_0 \mu_r} [Ey[n + nx*ny]\text{old} - Ey[n]\text{old}] - \frac{\Delta t}{\mu_0 \mu_r} [Ez[n + nx]\text{old} - Ez[n]\text{old}] + Hx[n]\text{old}
\]  
(14)

\[
Ez[n]\text{new} = \frac{\Delta t}{\varepsilon_0 \sigma} [Hx[n]\text{old} - Hx[n - 1]\text{old}] - \frac{\Delta t}{\varepsilon_0 \sigma} [Hx[n]\text{old} - Hx[n - nx]\text{old}] + \frac{\Delta t}{\varepsilon_0 \sigma} [Hx[n]\text{old} - Hx[n - nx]\text{old}] + (1 - \frac{\Delta t}{\varepsilon_0 \sigma})Hx[n]\text{old}
\]  
(15)

For wave absorbing layer space with CPML boundaries, the FDTD equations to update Hx and Ez are like (16) through (21).

\[
psihxy[n]\text{new} = (bxy[j]*psihxy[n]\text{old}) - cxy[j]*(Ez[n + nx]\text{old} - Ez[n]\text{old})
\]  
(16)

\[
psiez[n]\text{new} = (bey[j]*psiez[n]\text{old} + cex[i]*((Hy[n]\text{old} - Hy[n - 1]\text{old}) - (Hy[n]\text{old} - Hy[n - nx]\text{old})) + (1 - \alpha\Delta t/\varepsilon_0\sigma) * (psiez[n]\text{new} + psiez[n]\text{old})
\]  
(17)

\[
Hx[n]\text{new} = \frac{\Delta t}{\mu_0 \mu_r} [Ey[n + nx*ny]\text{old} - Ey[n]\text{old}] - \frac{\Delta t}{\mu_0 \mu_r} [Ez[n + nx]\text{old} - Ez[n]\text{old}] - \frac{\Delta t}{\mu_0 \mu_r} [Ez[n]\text{old} - \Delta x]\times Ey[n]\text{old} + Hx[n]\text{old}
\]  
(18)

\[
Ez[n]\text{new} = \frac{\Delta t}{\varepsilon_0 \sigma} [Hx[n]\text{old} - Hx[n - 1]\text{old}] \times kex[i] - \frac{\Delta t}{\varepsilon_0 \sigma} [Hx[n]\text{old} - Hx[n - nx]\text{old}] \times key[j] + (1 - \alpha\Delta t/\varepsilon_0\sigma) * (psiez[n]\text{new} + psiez[n]\text{old})
\]  
(19)

\[
Hx[n]\text{new} = \frac{\Delta t}{\mu_0 \mu_r} [Ez[n]\text{old} - \Delta y]\times Ey[n]\text{old} + Hx[n]\text{old}
\]  
(20)

\[
Ez[n]\text{new} = \frac{\Delta t}{\varepsilon_0 \sigma} [Hx[n]\text{old} - Hx[n - 1]\text{old}] \times kex[i] - \frac{\Delta t}{\varepsilon_0 \sigma} [Hx[n]\text{old} - Hx[n - nx]\text{old}] \times key[j] + (1 - \alpha\Delta t/\varepsilon_0\sigma) * (psiez[n]\text{new} + psiez[n]\text{old})
\]  
(21)

3 **CUDA**

In November 2006, NVIDIA company introduced CUDA, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture – that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

CUDA comes with a software environment that al-
allows developers to use C as a high-level programming language. CUDA's parallel programming model maintains a low learning curve for programmers familiar with standard programming languages such as C [11].

At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions.

GPU is capable of running many threads in parallel by the help of kernels defined in CUDA program that work on array of large data elements [1]. There are some limitations to these kernels, two of which - that is, allocating memory, and memory transfer, are of great importance [10]. They come from the barrier on the length of kernels and the sum of local memory they use.

A compiled CUDA program can execute on any number of processor cores, and only the runtime system needs to know the physical processor count.

4 FDTD IMPLEMENTATION ON A GPU

4.1 Data Conflict Problems and the Solution

Fig. 2 shows that according to the Maxwell’s equations, in order to update one component of magnetic field, we need the values of six components of its neighboring electric field and in order to update one component of electric field, we need the values of six components of its neighboring magnetic field.

Since computations in GPU are based on threads which execute concurrently, it is probable that some threads finish their work before other threads and change the value of the memory which is the input to another thread. Fig. 3 shows this data conflict. Thus, some threads violate the input values of other threads and this process can lead to erroneous computations.

To solve this problem, we should make the threads independent of each other, but unfortunately the updating equations depend on each other. So, we repeat the updating computations in short differential times rather than separate times, so that the time difference between updating magnetic and electric fields become negligible.

Moreover, we execute the kernel which updates magnetic fields after completion of the kernel which updates electric fields. When both kernels are finished, we update the magnetic and electric fields values for time duration $\Delta t$. Fig. 4 shows this process.

4.2 Data Divergence Problem and the Solution

A thread that is responsible for computing magnetic and electric fields, should consider three issues. First, it should understand the $\varepsilon$ and $\mu$ values of the space it is updating its fields. Second, it should detect whether this component of the space is in the CPML space or not, and if so use the CPML equations. Third, it should see whether there are any other field sources inside this component of the space, and if so add or subtract their values from the current field.

Hence, we should add conditional statements in our kernels to consider these issues. On the other hand, if there are conditional commands in kernels, the amount of computations become different for each thread. Also, the kernels should stop and test the condition in each iteration. Thus, the threads will diverge. In this state there is a divergence problem in GPU computations which decreases the speed of computations. So, the commands that are delivered to GPU processors should have the same computational loads and memory calls as possible.

In order to solve the divergence problem, we change the updating equations and also define some new parameters and allocate some spaces for them in memory. Instead of using conditional statements and different equations for different environments, which leads to divergence, we use the CPML equations as general updating equations in all environments. In each part of the space that is not inside the CPML boundaries, we choose the coefficients of the equations in a way that the terms that are related to CPML boundaries will be omitted automatically.

For instance, in equations of updating Ex the cex[i], bex[i], cey[j], and bey[j] vectors are defined so that they
have their original values in CPML areas and have value of zero out of this space. Accordingly, the pseizx and psezx values become zero and disappear from the equations. Furthermore, the key[i] and key[j] coefficients are defined to have their original values in CPML areas and one otherwise, so that their effects vanish in the computations.

Likewise, we utilize the same method to neutralize the effect of areas where there is no field source in general equations. More precisely, we define three dimensional arrays which have their original values in source areas and zero otherwise. Also, we take advantage of the same method in areas where have different \( \varepsilon_i \) and \( \mu_i \) coefficients and define parameters that have values corresponding to those areas. As a result, the kernels do not wait to check the condition of if statements and rapidly execute the multiplying process.

5 FDTD IMPLEMENTATION ON A CLUSTER OF GPUs

5.1 Implementing Computations on Discrete Memories

As mentioned earlier, we should divide the three dimensional space into small areas and iterate computations for each one and in each time step. It is true that based on the accuracy of computations the number of these areas varies, but we always need a large memory space for them. Moreover, considering the additional arrays that we add to solve the data divergence problem, we realize the large amount of memory needed to implement the FDTD algorithm. Consequently, we are obliged to use cluster systems which have large memory spaces.

As mentioned earlier, each update of electric or magnetic fields in FDTD equations require transfer of neighboring data of the other field between discrete memories. As a result, the algorithm which transfers these huge data should be efficient so that minimum transmissions take part between memories of clusters. In the following subsections we propose three methods to implement FDTD algorithm on discrete systems.

5.2 A Cluster with a Master Node Solution

To update a component of the magnetic field, we require the values of its neighboring electric fields. Similarly to update the value of an electric field component, we need the values of its neighboring magnetic fields. These neighboring components are placed on different memories in the cluster, so we need to transfer them between cluster nodes. One way to update components of memory which their data are placed on the border of memories is by the help of a master node. In this method the master receives the bordering data from two parties, executes the updating computations, and sends the result to the memory of each node. Fig. 5 shows a cluster of 5 systems in which one of them acts as a master node.

Fig. 6 depicts the process of updating bordering data of magnetic field by the help of a master node. First the data placed on the border are copied to the master node and it executes the updating procedure. When update is finished, the results are copied back to the right system. The white color segments in figure 6 represent the data that are not yet updated and the darker segments represents the data that are updated.

Other updates that their input data are locally ready for computation, do not depend on the master node. As a result, they are individually executed on the local systems. This process should be executed for all of \( H_x, H_y, H_z, E_x, E_y, \) and \( E_z \). Each of these updates needs transmission of 8 elements in each step of update (transfer of 6 elements before update and transfer of 2 elements after update).

5.3 Improving Implementation Using One Directional Derivative

We propose a new method for updating bordering data using one directional derivative to improve the execution time of FDTD algorithm. In this method, we use (22) to define the derivative formula of updating electric fields based on forward points and the derivative formula of updating magnetic fields based on backward points. It is worth noting that we cannot define the derivative formula of updating magnetic and electric fields based on the same direction. Because, by so doing the changes cannot transmit from side to side.

Taking advantage of this method, we need \( H[n] \) and \( H[n+1] \) to update \( E[n+1] \), but they are placed on two discrete memories. Consequently, we cannot update \( E[n+1] \) in neither of the nodes. Fig. 7 shows that in order to update electric field, first we should transfer two components of magnetic field, which are placed on two discrete memories, and one component of electric field to the memory of master node. When computations are completed, we should bring the results back to PC2’s memory.

This process results in 4 transfers in every update cycle for each field’s components and should be executed for both electric and magnetic fields. Hence, this method diminishes the amount of transfer by a factor of 2 for every cycle of update compared to the previous method.
In order to improve the performance of executing FDTD algorithm on clusters, we propose a new technique which we call it overlap algorithm. In this method we overlap the bordering data on bordering memories so that the bordering data are present on both neighborhood memories. As a result there is no need for an extra master node that was used in two previous methods and the updates are all executed on the nodes themselves.

The other major advantage of this technique is the reduction in the amount of data exchanged between discrete memories. Fig. 8 shows that to update H[n] on PC1 we need the value of E[n] and E[n+1], but we have only the value E[n] on PC1. However, thanks to our overlap algorithm, we have the value of both E[n] and E[n+1] on PC2’s memory. Thus, we update H[n] on PC2 rather than PC1 and then transfer the result back to PC1.

Consequently, we need only 2 transfers for each update of field’s component. As a result, the amount of transfer is reduced by a factor of two for every cycle of update compared to the previous technique. Fig. 9 shows how the three dimensional data are overlapped between memories.

![Fig. 6. Updating bordering data of magnetic field by the help of a master node](image)

\[
\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x}
\]

(22)

### 5.4 Improving Implementation Using Overlap Algorithm

5.4 Improving Implementation Using Overlap Algorithm

In order to improve the performance of executing FDTD algorithm on clusters, we propose a new technique which we call it overlap algorithm. In this method we overlap the bordering data on bordering memories so that the bordering data are present on both neighborhood memories. As a result there is no need for an extra master node that was used in two previous methods and the updates are all executed on the nodes themselves.

The other major advantage of this technique is the reduction in the amount of data exchanged between discrete memories. Fig. 8 shows that to update H[n] on PC1 we need the value of E[n] and E[n+1], but we have only the value E[n] on PC1. However, thanks to our overlap algorithm, we have the value of both E[n] and E[n+1] on PC2’s memory. Thus, we update H[n] on PC2 rather than PC1 and then transfer the result back to PC1.

Consequently, we need only 2 transfers for each update of field’s component. As a result, the amount of transfer is reduced by a factor of two for every cycle of update compared to the previous technique. Fig. 9 shows how the three dimensional data are overlapped between memories.

![Fig. 6. Updating bordering data of magnetic field by the help of a master node](image)

\[
\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x}
\]

(22)

### 5.5 Proper Three Dimensional Data Distribution among GPU Memories

5.5 Proper Three Dimensional Data Distribution among GPU Memories

The most important memory in GPU is its global memory which we use for executing our program. Data division order on this memory is vital, because the executing time of our algorithm depends on it. More precisely, if we call data from discrete locations of GPU memory or copy the results to discrete locations, we will radically decrease the speed of computations. Also, we know that implementing FDTD computations on discrete memories of GPU clusters need constant transfer of bordering data between them through PCI-EXPRESS interfaces. Moreover, to have a coalesced global memory access pattern, threads should access data in sequence. Hence, we should manage data on GPU memories so that they arrange continuously besides each other.

Suppose that the electric and magnetic fields data arrays are divided through the x direction. Fig. 10 shows that data of the x direction are discontinuously arranged in memory. For instance, the data of x1, x2, and x3 pages are discretely distributed in memory. Consequently, this scenario takes excessive computation time and reduces the speed of execution.

In another possibility, consider that electric and magnetic fields data arrays are divided through the y direction. Fig. 11 shows that even though the data in the y direction are closer to each other, but there is still a discontinuity in data which diminishes the speed of computa-
Now consider dividing data through the z direction. Fig. 12 depicts that bordering data are now continuously located besides each other. As a result, all the data on this group can be copied or called from memory at the same time and with a much higher speed. Hence, we use the latter scenario in our algorithm and map our three dimensional data onto one dimensional memory using a loop like one depicted in the algorithm 1.

6 EXPERIMENTS

The configurations of implementation systems are listed in Table 1.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>System 1</th>
<th>System 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CPU Memory</td>
<td>32GB</td>
<td>2 GB</td>
</tr>
<tr>
<td>CPU Frequency</td>
<td>2.67GHz</td>
<td>2611MHz</td>
</tr>
<tr>
<td>GPU</td>
<td>TESLA C1060</td>
<td>NVIDIA GeForce 8400 GS</td>
</tr>
<tr>
<td>Total GPU Memory</td>
<td>16GB</td>
<td>256MB</td>
</tr>
<tr>
<td>Number of GPUs</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>CUDA Version</td>
<td>CUDA 3.1-win 7 64bit</td>
<td>CUDA 3.1-win XP 32bit</td>
</tr>
<tr>
<td>GPU Frequency</td>
<td>1.3 GHz</td>
<td>450 MHz</td>
</tr>
<tr>
<td>Number of Processors Per GPU</td>
<td>240</td>
<td>16</td>
</tr>
<tr>
<td>CPU-GPU Interface Port</td>
<td>PCI-EXPRESS 1.5 Gbps</td>
<td>PCI-EXPRESS 1Gbps</td>
</tr>
<tr>
<td>MPI Software</td>
<td>Mpic2-1.3a2-win-x86-64.msi</td>
<td>Mpic2-1.0.5p2-win32-ia32.msi</td>
</tr>
<tr>
<td>O.S</td>
<td>Windows7 Ultimate-64bit</td>
<td>Windows XP 32bit</td>
</tr>
</tbody>
</table>

6.1 Executing Computations on a Single GPU

We start our experiments with the changes we made in the FDTD algorithm which resulted in solving data conflicts and data divergence problems. Fig. 13 shows the execution time of running our algorithm on system 2 versus the number of update repetitions. It is clear from the figure that before applying changes to the algorithm the GPU time is more than the CPU time. However, after applying changes to GPU code, we achieve a 1.6 times speedup when executing computations on GPU compared to CPU.
6.2 Executing Computations on Discrete Clusters

6.1.1 A Cluster with a master node, Using One Directional Derivative and Overlap Algorithm

In this experiment we compare the execution time of running FDTD on a CPU cluster with a master node, on the same cluster using one directional derivative, and on the same cluster using overlap algorithm without any master node. We use system 2 configuration whose characteristics are depicted in Table 1. Fig. 14 shows the execution time versus the number of update repetitions. Executing computations using one directional derivative algorithm running on a cluster of CPUs of system 2 is 1.7 times faster than executing computations on a cluster of CPUs with a simple master node method. Also, executing computations using overlap algorithm running on a cluster of CPUs of system 2 is approximately 5 times faster than executing computations on a cluster of CPUs with a master node method and 3 times faster than one directional derivative method.

![Fig. 13. Comparison of FDTD execution time on CPU and GPU before and after solving data conflicts and divergence](image)

![Fig. 14. Comparing execution time of master, one directional derivative, and overlap methods](image)

6.1.2 Dividing data in the right direction

In another experiment we test the effect of considering the direction of dividing data on discrete memories of our cluster. Fig. 15 shows the execution time versus the number of update repetitions for a cluster of CPUs of system 2 which takes advantage of overlap algorithm. The implementation which considers the right direction of dividing data on GPU memories of the cluster is approximately 1.6 times faster than the implementation that does not consider it.

6.3 Executing Computations on a Cluster of GPUs Placed on One System

Now that we have evaluated the effects of all hardware and software corrections in our algorithm, we want to compare the power of a cluster of GPUs and a cluster of CPUs consisted of hardware of system 1 which their configurations are depicted in Table 1. This system consists of 4 GPUs which construct a powerful GPU cluster and 2 CPUs each having 4 cores. Fig. 16 shows the execution time versus the number of update repetitions. Execution of the algorithm considering all the points previously discussed using overlap algorithm and considering the right direction of data division between memories on a cluster of GPUs placed on one system is approximately 40 times faster than executing the algorithm on a cluster of CPUs placed on one system using overlap algorithm and considering the right direction of data division.

![Fig. 15. Effect of considering the right direction of dividing data between cluster’s node](image)

![Fig. 16. Executing Computations on a Cluster of GPUs Placed on One System](image)
Fig. 16. Comparing implementation time of FDTD algorithm on a cluster of CPU and a cluster of GPU.

7 Conclusion

In this paper, we evaluated three systems for implementing electromagnetic simulations of Maxwell’s equations using PDEs. As described in the paper, these equations cannot be solved analytically and need numerical methods like FDTD.

We reformulated the FDTD in order to be suitable for implementing on GPU’s parallel architecture. We suggested methods to solve the data conflict and data divergence problems in implementation of FDTD method on GPU memories. Applying these corrections, we got 1.6 times speedup running our algorithm on GPU instead of CPU.

Next, we implemented our revised algorithm on a cluster of CPUs and suggested a system with a master node, a cluster with a master node which takes advantage of one directional derivative, and a cluster without master node which takes advantage of overlap algorithm. We respectively achieved 1.7 and 5 times speedup in second and third configurations compared to the first one.

The next improvement in execution time of FDTD on a cluster was considering the right direction of cutting bordering data which resulted in 1.6 times speedup.

Finally, we achieved 40 times speedup running our revised algorithm on a cluster of GPUs placed on system 1 compared to a cluster of CPUs placed in this system.

Acknowledgment

The authors gratefully acknowledge the High Performance Computing Lab of Amirkabir university for their support.

References


Ehsan Arianyan is with the Electrical & Electronics Engineering Department, University of AMIRKABIR, High Performance Computing Lab, Tehran, Iran E-mail: ehsan_arianyan@aut.ac.ir. Ehsan Arianyan is now PhD student in Amirkabir University of technology. He is graduated from ELMOSANAT university of TEHRAN in BSc of Electronics and from Amirkabir University of technology in MSc of Electronics. His interests are currently cluster computing, virtualization, and decision algorithms.

Seyed Ahmad Motamedi is with the Electrical & Electronics Engineering Department, University of AMIR KABIR, High Performance Computing Lab, Tehran, Iran.

Mehran Hekmatpanah graduated from the Electrical & Electronics Engineering Department, University of AMIRKABIR, High Performance Computing Lab, Tehran, Iran. His interests are currently cluster computing.

Iman Arianyan is with the Electrical & Electronics Engineering Department, University of AMIRKABIR, High Performance Computing Lab, Tehran, Iran. Iman Arianyan is now PhD student in Amirkabir University of technology. He is graduated from Amirkabir university in BSc of Electronics and from Amirkabir University of technology in MSc of Communication. His interests are currently cluster computing, virtualization, and decision algorithms.