Hindawi Publishing Corporation
International Journal of Antennas and Propagation
Volume 2014, Article ID 321081, 8 pages
http://dx.doi.org/10.1155/2014/321081

*Research Article*

# GPU-Accelerated Parallel FDTD on Distributed Heterogeneous Platform

**Ronglin Jiang,[1] Shugang Jiang,[2] Yu Zhang,[2] Ying Xu,[1] Lei Xu,[1] and Dandan Zhang[1]**

[1] *Research and Development Department, Shanghai Supercomputer Center, Shanghai 201203, China*
[2] *School of Electronic Engineering, Xidian University, Xi'an 710071, China*

Correspondence should be addressed to Ronglin Jiang; rljiang@ssc.net.cn

Received 31 October 2013; Revised 27 December 2013; Accepted 10 January 2014; Published 20 February 2014

Academic Editor: Lei Zhao



This paper introduces a (finite difference time domain) FDTD code written in Fortran and CUDA for realistic electromagnetic calculations with parallelization methods of Message Passing Interface (MPI) and Open Multiprocessing (OpenMP). Since both Central Processing Unit (CPU) and Graphics Processing Unit (GPU) resources are utilized, a faster execution speed can be reached compared to a traditional pure GPU code. In our experiments, 64 NVIDIA TESLA K20m GPUs and 64 INTEL XEON E5-2670 CPUs are used to carry out the pure CPU, pure GPU, and CPU + GPU tests. Relative to the pure CPU calculations for the same problems, the speedup ratio achieved by CPU + GPU calculations is around 14. Compared to the pure GPU calculations for the same problems, the CPU + GPU calculations have 7.6%–13.2% performance improvement. Because of the small memory size of GPUs, the FDTD problem size is usually very small. However, this code can enlarge the maximum problem size by 25% without reducing the performance of traditional pure GPU code. Finally, using this code, a microstrip antenna array with $16 \times 18$ elements is calculated and the radiation patterns are compared with the ones of MoM. Results show that there is a well agreement between them.

## 1. Introduction

Electromagnetic researches have penetrated into many aspects of our daily lives, such as wireless signal propagation, mobile communications, and radar technology. Since the electromagnetic wave propagation in a real environment is very complex and usually the characteristics of electromagnetic waves cannot be obtained by theoretical analysis, the numerical solutions of electromagnetic waves have gradually been developed, such as method of moments (MoM), uniform geometrical theory of diffraction (UTD), physical optics (PO), and finite difference time domain (FDTD) method. Each method has its own advantages, for instance, high-frequency methods UTD and PO are suitable for electrical large objects, while the general electrical objects can be treated by the low-frequency methods such as MoM. The FDTD method which will be discussed in this paper was first proposed in 1966 by Yee [1]. The electromagnetic field is discretized by the Yee grid [1] and the Maxwell's equations are converted into finite difference form. Thus, the numerical FDTD can be used to simulate the processes of electromagnetic scattering, radiation of large scale, complex shapes objects, or nonuniform mediums.

Thanks to the rapid development of computer industry, besides traditional CPU, other accelerations like Graphics Processing Unit (GPU, including integrated GPU [2]) and Many Integrated Core Architecture (MIC) have been developed to improve the calculation efficiency. The FDTD method using the finite difference method is very convenient for GPU transplant [3–6]. At early times, single GPU card can achieve a much faster execution speed than INTEL or AMD CPUs [7, 8]. Using NVIDIA FX 5900 by Krakiwsky et al. [7], the speedup is about 7 times faster than Intel P4 CPU. CUDA based code has shown good performance compared with C code operated on Xeon CPU in 3D FDTD simulations. Liuge et al. [9] archived 10 times speedup factors compared with INTEL Xeon E5410 CPU in all their cases by using NVIDIA GTX280 video card. Because of the shortcoming of small memory size of single GPU card, multi-GPU card system was developed for large scale computations [10, 11]. Nagaoka and

Watanabe [11] used 3 nodes (21 NVIDIA TESLA C2070 GPU cards in total) to carry out the FDTD numerical simulations and the parallel efficiency is about 30%. The speedup ratio is 1.5 relative to NEC SX-8R CPU (8 cores). In the GPU experiments made by Kim and Park [12], a theoretical computing efficiency was obtained by spliting GPU kernel functions, in which these functions were arranged in an ingenious way and the MPI communications and FDTD calculations overlapped each other. Xu et al. [13] used 64 NVIDIA TESLA K20m GPU cards in their three-dimensional (3D) FDTD simulation, and the parallel efficiency is 67.5% and the speedup ratio is 3.1 relative to Intel XEON E5-2670 CPU (8 cores). With the development of GPU clusters, GPU computing has a great potentials, based on which more and more GPU-accelerated applications appear.

The FDTD method investigated in this paper includes some module functions like source excitation, near-to-far transformation, preprocessing, and postprocessing. Therefore, this FDTD code can be applied to solve a application problem directly. The code is mainly written in Fortran which is combined with CUDA to make use of GPU resources. The Message Passing Interface (MPI, for large scale calculation) programming and Open Multi-Processing programming (OpenMP, for regulation of CPU and GPU resources) are also very important in our code. Thus, these functions are introduced as follows: Section 2 discusses the FDTD equations and discretization method; the GPU algorithm are illustrated in Section 3; Section 4 discusses the results of GPU and CPU tests; finally, the conclusion and remarks are presented in Section 5.

## 2. Numerical Scheme of FDTD

Maxwell's curl equations are given as follows [14, 15]:

$$\begin{aligned} \nabla \times \mathbf{H} &= \varepsilon \frac{\partial \mathbf{E}}{\partial t} + \sigma \mathbf{E}, \\ \nabla \times \mathbf{E} &= -\mu \frac{\partial \mathbf{H}}{\partial t} - \sigma_m \mathbf{H}, \end{aligned} \tag{1}$$

where $\mathbf{E}$, $\mathbf{H}$ are electric field intensity and magnetic field intensity. The other parameters $\varepsilon$, $\mu$, $\sigma$, and $\sigma_m$ are permittivity, permeability, electrical conductivity, and magnetic permeability, respectively. In vacuum, $\sigma = 0$, $\sigma_m = 0$, and $\varepsilon = \varepsilon_0 = 8.85 \times 10^{-12}$ F/M and $\mu = \mu_0 = 4\pi \times 10^{-7}$ H/M. The FDTD recurrence formula in Cartesian coordinates can be got from the formula above (e.g., take $x$ components of electromagnetic):

$$\begin{aligned} &E_x^{n+1}\left(i+\frac{1}{2}, j, k\right) \\ &\quad = CA\left(i+\frac{1}{2}, j, k\right) \cdot E_x^n\left(i+\frac{1}{2}, j, k\right) \\ &\quad + CB\left(i+\frac{1}{2}, j, k\right) \\ &\quad \times \left( \frac{H_z^{n+(1/2)}\left(i+(1/2), j+(1/2), k\right)}{\Delta y} \right. \\ &\qquad - \frac{H_z^{n+(1/2)}\left(i+(1/2), j-(1/2), k\right)}{\Delta y} \\ &\qquad - \frac{H_y^{n+(1/2)}\left(i+(1/2), j, k+(1/2)\right)}{\Delta z} \\ &\qquad \left. - \frac{H_y^{n+(1/2)}\left(i+(1/2), j, k-(1/2)\right)}{\Delta z} \right), \end{aligned} \tag{2}$$

$$\begin{aligned} &H_x^{n+(1/2)}\left(i, j+\frac{1}{2}, k+\frac{1}{2}\right) \\ &\quad = CP\left(i, j+\frac{1}{2}, k+\frac{1}{2}\right) \cdot H_x^{n-(1/2)}\left(i, j+\frac{1}{2}, k+\frac{1}{2}\right) \\ &\quad - CQ\left(i, j+\frac{1}{2}, k+\frac{1}{2}\right) \\ &\quad \times \left( \frac{E_z^n\left(i, j+1, k+(1/2)\right)}{\Delta y} \right. \\ &\qquad - \frac{E_z^n\left(i, j, k+(1/2)\right)}{\Delta y} \\ &\qquad - \frac{E_y^n\left(i, j+(1/2), k+1\right)}{\Delta z} \\ &\qquad \left. - \frac{E_y^n\left(i, j+(1/2), k\right)}{\Delta z} \right), \end{aligned} \tag{3}$$

where

$$\begin{aligned} CA(m) &= \frac{1-\sigma(m)\Delta t/2\varepsilon(m)}{1+\sigma(m)\Delta t/2\varepsilon(m)}, \\ CB(m) &= \frac{\Delta t/\varepsilon(m)}{1+\sigma(m)\Delta t/2\varepsilon(m)}, \\ CP(m) &= \frac{1-\left(\sigma_m(m)\Delta t\right)/2\mu(m)}{1+\left(\sigma_m(m)\Delta t\right)/2\mu(m)}, \\ CQ(m) &= \frac{\Delta t/\mu(m)}{1+\left(\sigma_m(m)\Delta t\right)/2\mu(m)}, \end{aligned} \tag{4}$$

where $\Delta x$, $\Delta y$, $\Delta z$ are space steps and $\Delta t$ is time step. $\sigma(m)$ is the value of $\sigma$ at the positon $m$, for example, $m = (i+1/2, j, k)$ in formula (2). Other variables $\sigma_m(m)$, $\varepsilon(m)$, and $\mu(m)$ have the same meaning. From formulae (2) and (3), we know that the electric or magnetic field at any point is regarded with the four nearest magnetic or electric fields around.

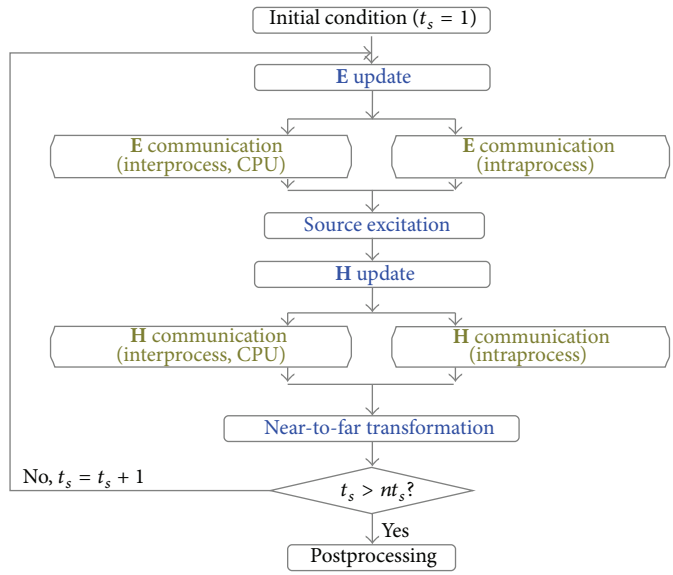


Figure 1: The flowchart of cudaFDTD. The modules written in blue mean the main calculations and that in brown mean data communications. Note that the interprocess is done by only CPU; other function modules are treated by CPU and GPU simultaneously.

## 3. FDTD Parallelization Strategy

*3.1. Calculation Flow.* This section describes the strategy of using both CPU and GPU resources. For the convenience of expressing, the new FDTD code after GPU transplant is referred to as cudaFDTD hereafter. cudaFDTD includes several functional modules as follows (for detailed formulae we refer the readers to the books by Taflove and Hagness [15] and Ge and Yan [16]).

(1) **E** and **H** updates. As described above, cudaFDTD uses the Yee cell and leapfrog scheme to update electric field and magnetic field in turn. The UPML layer [17] is placed near the boundaries. In our cudaFDTD code, the thickness of the UPML layer is set to be 5 grid points which is suitable for most situations. Of course, this value can be adjusted for some extreme cases. There is little influence coming from the boundaries and the verification of the correctness of cudaFDTD is presented in Section 4.

(2) Source excitation. cudaFDTD has the source excitation, which creates perturbations in the computational domain for scattered or radiant simulations. The form of source excitation is flexible.

(3) Near-to-far transformation. According to the Huygens' principle, the scattered or radiant field at an observation point outside the computational domain can be derived from the already existed **E** and **H** in the domain. The **E** and **H** values are recorded during the **E** and **H** updates. After all the updating processes have been accomplished, the far fields can be derived in postprocessing procedure.

The functional modules depicted above can be calculated by CPU and GPU simultaneously. The strategy of distributing the computing tasks for CPU and GPU is presented in Section 3.3. The completed flowchart containing the above functional modules is shown in Figure 1.

*3.2. The Process and Thread in cudaFDTD.* The number of processes booting by cudaFDTD depends on the number of GPU cards in most of cases (the number of GPU cards greater than that of CPU cores). Each process contains several threads. One thread is distributed to control the CUDA stream; the rest of them are for CPU calculations (using OpenMP). If one node of a supercomputer offers $N_c$ CPU cores and $N_g$ GPU cards, the avaliable process number and thread number of each process are given by the following fomulae:

$$N_p = \min\left(N_c, N_g\right), \tag{5}$$

$$N_t = \text{int}\left(\frac{N_c}{N_p}\right), \tag{6}$$

where the function "min" in (5) means minimum function, and function "int" in (6) rounds the result to an integer. For instance, if one node offers $N_g = 2$ GPU cards and $N_c = 16$ CPU cores, then 2 processes are started at this node. Each process forks to $N_t = 8$ threads for 8 CPU cores and one of the 8 threads (or say one of the 8 CPU cores) is distributed to control one CUDA stream, as shown by Figure 2. CPU and GPU can work simultaneously.

*3.3. Data Partition.* The data partition consists of two aspects: (1) interprocess partition and (2) intraprocess partition. Interprocess partition distributes the total computational domain to different processes. Every subdomain has its own CPU and GPU devices and the number of subdomain is decided by $N_p$. The data exchage is realized by MPI communications. The intraprocess partition means the data

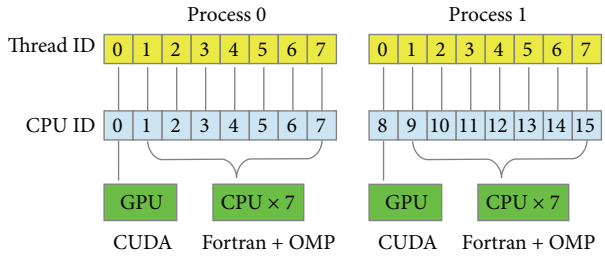


Figure 2: An example to show the processes and threads in cudaFDTD: 2 processes are started and each process controls 8 threads.

distribution in one process. Because every subdomain uses CPU and GPU simultaneously, thus the intraprocess partition splits the subdomain to two pieces for CPU and GPU calculations. The boundary data is passed by cudaMemcpy between CPU and GPU.

As we know, the GFLOPS (giga floating-point operations per second) between CPU and GPU is very different. A ratio has to be determined for load balance of CPU and GPU. In cudaFDTD, we use the parameter cuRatio to control the amount of calculation for GPU. However, the cuRatio is not a simple divisional result of theoretical GFLOPS of CPU and GPU devices but the actual GFLOPS which is counted in the code. Figure 3 gives the partition method in intraprocess partition. The data of this subdomain is divided into upper and lower parts for the purpose of spatial locality. The expression of cuRatio is

$$\text{cuRatio} = \frac{nzg}{nzc + nzg} = \frac{nzg}{nz}, \tag{7}$$

where $nx$, $ny$, and $nz$ are the dimension sizes of $x$, $y$, and $z$ directions. So, the block size calculated by CPU is $nx \times ny \times nzc$ and the rest space $nx \times ny \times nzg$ is left for GPU. The interprocess partition and the intraprocess partition are applicative for all function modules like update, source excitation, and near-to-far transformation.

Since the data partition has two different types, the data communication also has two types: (1) interprocess communication and (2) intraprocess communication. In interprocess communication the boundary data is passed thought nonblocking MPI functions like MPI_ISEND and MPI_IRECV, whereas the data transportation in intraprocess communication is transmitted by the CUDA function cudaMemcpy between CPU and GPU device. For both communication types, the operations are similar (see Figure 3(b)): (1) gathering the bounday data of all variables; (2) sending this data by MPI_ISEND or cudaMemcpy; (3) receiving data by MPI_IRECV or cudaMemcpy; (4) distributing the boundary data to overlap the ghost data.

*3.4. Optimization Methods Used in cudaFDTD.* To maximize the efficiency of the CPU and the GPU computing, we adopted a number of technical methods to improve the efficiency of our procedures. We have the following points for the optimization of Fortran code.

(1) Reduce unnecessary arithmetic operation. For example, defining intermediate variables can reduce duplication of calculations; besides, division and modulo operations should be avoided.

(2) Specify CPU instruction set when compiling code. For example, Intel compiler uses default SSE2 instruction set, and it is recommended to use the option -xHost which can automatically use the highest level of instruction set. The improvement of program performance is substantial.

(3) Artificial vectorization (only for Intel compiler). If one loop is very complex, the compiler might deem that such a loop is not suitable for vectorization. However, if the loop can actually be vectorized, we can use the following clauses to force the compiler to do vectorization (for instance, in Fortran code): !DEC$ IVDEP or !DEC$ VECTOR ALWAYS.

We have the following optimization methods for CUDA code.

(1) Reduce unnecessary arithmetic operation. This is the same as CPU code.

(2) Memory coalescing. Coalesced memory access provides beneficial performance [18]. The latest NVIDIA GPU (compute capability 2.0 or later) fetches data from memory in groups of 32 4-byte floats [19]. For instance, if a thread-block needs to fetch 16 floats from the GPU memory, at most 16 fetches are needed in the worst case. If the 16 floats are adjacent in memory, then a single fetch operation can grab all required 16 floats. Therefore, we already do our best to use the data in adjacent memory in order to merge memory accesses.

(3) Constant memory. GPU constant memory is a small-readable memory, and it has the following advantages compared to other memories: (1) single read can broadcast its value to other nearby thread and (2) constant memory has cache and the sequential read of the same constant is very fast. Some of the constant coefficients of the material are put into the constant memory in cudaFDTD.

The other optimization techniques, like the usage of shared memory [20], will be developed in the next version. The performances with and without optimizations are shown by Table 1. In this test, the CPU and GPU devices are Intel XEON E5-2670 and NVIDIA TESLA K20m in Shanghai Supercomputer Center (SSC). The problem size is $512 \times 256 \times 256$.

## 4. cudaFDTD Tests

Most of our tests are compiled and executed on the GPU cluster in the Center for High Performance Computing of Shanghai Jiaotong University. Each node in the GPU cluster is configured with 2-way INTEL XEON E5-2670 CPUs and 2 NVIDIA TESLA K20m GPUs. A total of 50 nodes were connected by Mellanox FDR InfiniBand. cudaFDTD is a mixed

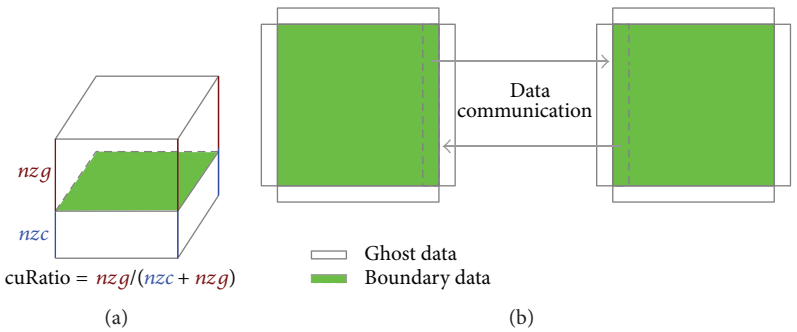


Figure 3: (a) Intraprocess partition. (b) The data exchange communications.

Table 1: The performances before and after optimization.

| | Loop time of E5-2670 single core (s) | Loop time of K20m single card (s) |
|---|---|---|
| Before optimization | 2.2740 | 0.1320 |
| After optimization | 1.3587 | 0.1021 |
| Performance improvement | 40% | 23% |

programming code of Fortran and CUDA using double-precision float for calculations for which the Fortran compiler is Intel ifort 13.1.1 and CUDA 5.0. Since Intel Fortran compiler cannot directly call GPU kernel functions, our cudaFDTD program is necessarily to call C language functions first and then GPU kernel function in the C language functions.

*4.1. Verification for the Numerical Accuracy.* Electric dipole source are used to verify the accuracy of cudaFDTD code [16]. The vertical electric dipole is located in the center of the computational domain, and we observe the changes of $E_z$ value with time at an observational point $N\Delta x$ from the dipole. The computational box is $nx \times ny \times nz = 64 \times 64 \times 64$ and the Yee cell is $\Delta x \times \Delta y \times \Delta z = 5\,\text{cm} \times 5\,\text{cm} \times 5\,\text{cm}$. The time step is $\Delta t = 83.3\,\text{ps}$. The electric field $E_z$ generated by electric dipole is added directly to $E_z(nx/2, ny/2, nz/2)$:

$$E_z\left(\frac{nx}{2}, \frac{ny}{2}, \frac{nz}{2}\right) = -2 \times 10^{-10} \frac{\varepsilon_0 (t - 3T)\,\Delta x \Delta y \Delta z}{T^2 \Delta t} e^{-((t-3T)/T)^2}, \quad (8)$$

where $T = 2\,\text{ns}$ and $\varepsilon_0 = 8.8541878 \times 10^{-12}\,\text{F/m}$. The thickness of UPML is $5\Delta x$. The theoretical expression for $E_z(nx/2 + N, ny/2, nz/2)$ which is $N\Delta x$ from the center $(nx/2, ny/2, nz/2)$ is given by

$$E_z\left(\frac{nx}{2} + N, \frac{ny}{2}, \frac{nz}{2}\right) = \frac{10^{-17}}{r} e^{-T_0^2} \left(4\frac{T_0^2}{T^2} - 2\frac{cT_0}{rT} - \frac{2}{T^2} + \frac{c^2}{r^2}\right), \quad (9)$$

where $T_0 = (t - r/c - 3T)/T$, $r = N\Delta x$. $c$ is the speed of light. We choose $N = 10$ in this verification and Figure 4 presents the difference between the simulation result and theoretical

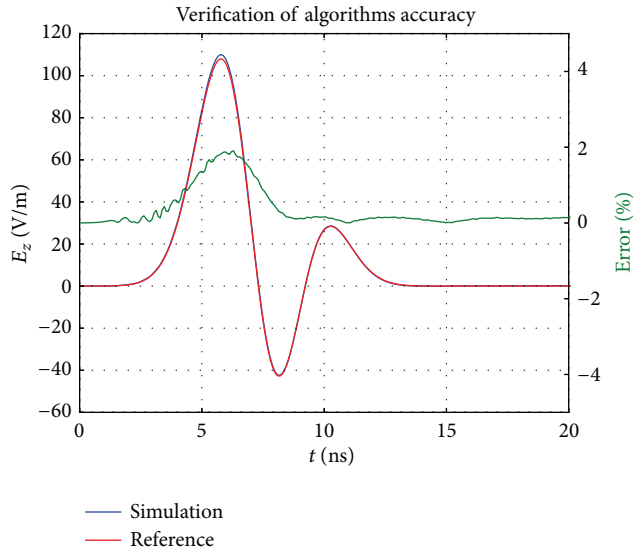


Figure 4: The accuracy verification of cudaFDTD. The green line is relative error using the right $y$ axis.

Table 2: Statistics of the time consumption of cudaFDTD.

| Function module | Proportion |
|---|---|
| **E** update | 48.2% |
| Inter- and intraprocess communications of **E** | 5.0% |
| Collection and distribution boundary data of **E** | 2.2% |
| Source excitation | 0.6% |
| **H** update | 35.9% |
| Inter- and intraprocess communications of **H** | 3.6% |
| Collection and distribution boundary data of **H** | 3.8% |
| near-to-far transformation | 0.7% |

results. For the scale of $64 \times 64 \times 64$, the maximum relative error in about 2% which is reasonable and accurate enough to ensure the correctness of our cudaFDTD.

In order to obtain the detailed statistics of the time consumption, a larger scale test with the size $2048 \times 1024 \times 1024$ is performed. The time consumption of all function modules is given in Table 2. In this test, we used 32 nodes.

*4.2. Investigation of cuRatio.* A new parameter cuRatio is introduced in cudaFDTD to adjust the amount of calculation

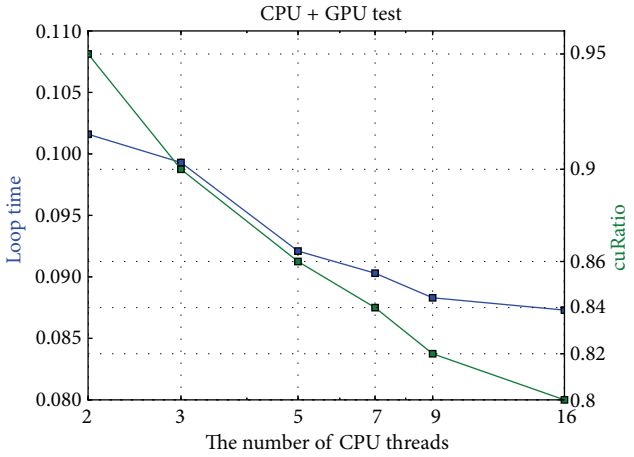


Figure 5: The variation of loop time and cuRatio value over threads number. The problem size is 512 × 256 × 256. The blue curve means loop time (left $y$ axis) and the green curve presents the value of cuRatio (right $y$ axis).

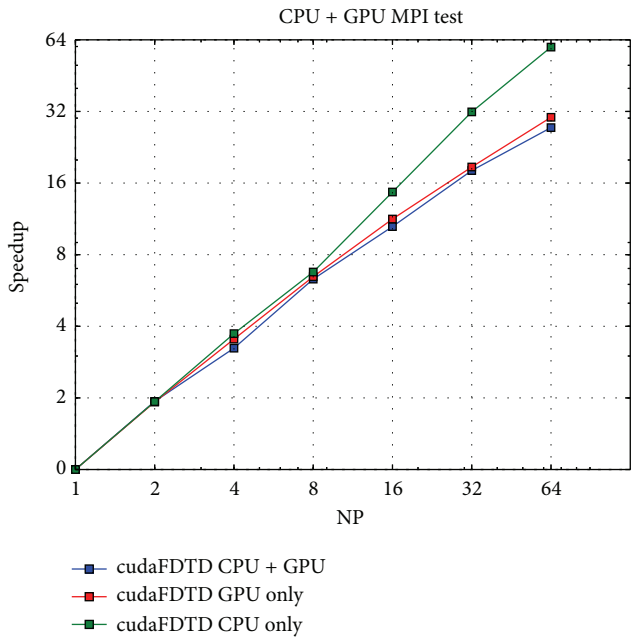


Figure 6: Speedup test of cudaFDTD. The unit of $x$ axis is NP (the number of processes) which has different means for different speedup curve: 1 NP = 1 CPU core for pure CPU curve, 1 NP = 1 GPU card for pure GPU curve, and 1 NP = 7 CPU cores + 1 GPU card for CPU + GPU curve. The problem size is 512 × 256 × 256.

for GPU. If the problem size is 100 × 100 × 100 and cuRatio is 0.8, the part for GPU is 100 × 100 × 80. It is also an important key to guarantee the load balance. Figure 5 shows the variation of loop time and cuRatio value over threads number. The test case is the same as described in Section 4.1. We used one node in this case and the problem size is 512 × 256×256. As depicted in Figure 2, there is a thread for CUDA stream and the other ones for CPU calculations. Thus, the values 2, 3, 5, 7, 9, 16 of $x$ axis in Figure 5 correspond to 1, 2, 4, 6, 8, 15 for CPU calculations (it is noted that there is only one process in one node, so it can use threads number from 2 to 16). The minimum loop time is 0.084 s when the number of threads is 16 and cuRatio = 0.80. Relative to the pure GPU test (the loop time is 0.1028 s; see the next section), the computational speed is increased by 15%. Moreover, since the cuRatio value is 0.80, the NVIDIA TESLA K20m GPU afforded 80% of total calculation. The actual problem size can be increased by 25% to further improve the CPU and GPU resource utilization.

Table 3: Loop time information used in Figure 6.

| NP | Pure CPU | Pure GPU | CPU + GPU (cuRatio) |
|---|---|---|---|
| 1 | 1.3541 | 0.1028 | 0.0903 (0.840) |
| 2 | 0.7020 | 0.0535 | 0.0467 (0.850) |
| 4 | 0.3634 | 0.0291 | 0.0279 (0.860) |
| 8 | 0.2168 | 0.0159 | 0.0143 (0.860) |
| 16 | 0.0923 | 0.0091 | 0.0086 (0.860) |
| 32 | 0.0425 | 0.0055 | 0.0050 (0.840) |
| 64 | 0.0227 | 0.0034 | 0.0033 (0.810) |

*4.3. Speedup Test.* In speedup test, we use the same test case as described in Section 4.1, but the problem scale is 512 × 256 × 256 which uses 4.3 GB memory. The speedup curves of pure CPU, pure GPU, and CPU + GPU are plotted in Figure 6. We note that the unit of $x$ axis is NP (the number of processes) which has different means for different speedup curves: 1 NP = 1 CPU core for pure CPU curve, 1 NP = 1 GPU card for pure GPU curve, and 1 NP = 7 CPU cores + 1 GPU card for CPU + GPU curve. Because one CPU core is used to control the CUDA stream, we use 7 CPU cores in CPU + GPU test (it is noted that there are two processes in one node, so each process can use 8 CPU cores at most). Compared to one process, the parallel efficiencies ($t_1/(n\times t_n)$, $t_1$ means the loop time calculated by 1 NP; $t_n$ means the loop time calculated by $n$ NP.) of pure CPU, pure GPU, and CPU + GPU are 93.2%, 47.2%, and 42.8%, respectively. The small problem size (512 × 256 × 256) and complex communication are responsible for the low parallel efficiencies of pure GPU. The time spent in communication takes the larger proportion in 64 processes. As for the CPU + GPU test, the perfect load balance is difficult to reach which further polls down the parallel efficiency. Although the parallel efficiency is unsatisfactory, the loop time is very short (see Table 3) which gives a high speedup ratio.

Compared to CPU calculation, the speedup ratios of GPU and CPU + GPU are given in Tables 4 and 5. The meaning of NP is described above. The difference between Tables 4 and 5 is that the MPI communication is involved in 64 processes tests, while the other one has no MPI communication. As shown in these tables, pure GPU calculation has a speedup ratio of 13.2 and the ratio in CPU + GPU test is higher than 13.2 in both tables. Compared to the pure GPU calculations for the same problems, the CPU + GPU calculations have 7.6%–13.6% performance improvement. Since the perfect

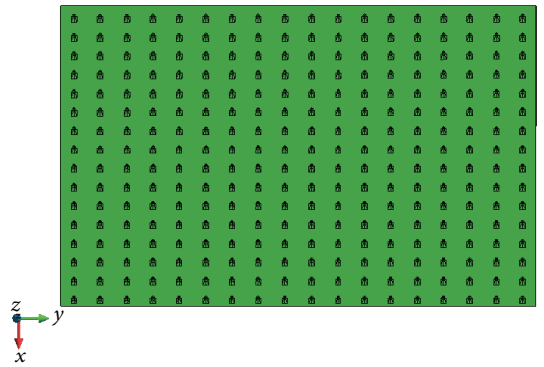


Figure 7: 16 × 18 mircostrip antenna array.

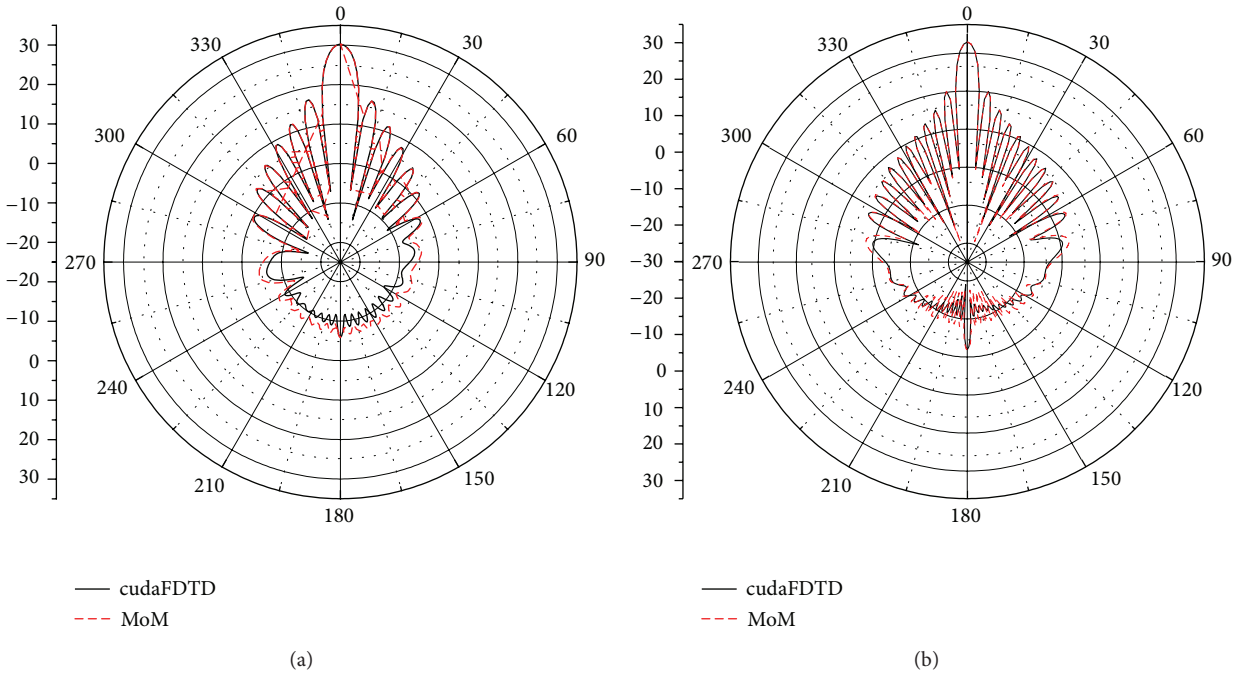


Figure 8: The radiation patterns of *xoz* (a) and *yoz* planes (b) of 16 × 18 mircostrip antenna array.

Table 4: Speedup ratio (1 NP, problem size 512 × 256 × 256).

| Device | Loop time | Speedup |
|---|---|---|
| Pure CPU | 1.3541 | 1 |
| Pure GPU | 0.1028 | 13.2 |
| CPU + GPU | 0.0903 (cuRatio = 0.84) | 15.0 |

Table 5: Speedup ratio (64 NP, problem size 2048 × 1024 × 1024).

| Device | Loop time | Speedup |
|---|---|---|
| Pure CPU | 1.5276 | 1 |
| Pure GPU | 0.1156 | 13.2 |
| CPU + GPU | 0.1079 (cuRatio = 0.90) | 14.2 |

load balance is difficult to achieve, the speedup of 64 processes is lower than that of single process.

*4.4. Radiation of Mircostrip Array.* In this section, we present an application test of 16 × 18 mircostrip antenna array. The initial model is illustrated in Figure 7. The frequency is 9 GHz; the size of the array is 0.2453 m × 0.3908 m × 0.0013 m (grid size is $dx = dy = dz = 0.00033$ m). The total computational grid is 800 × 1200 × 100. Using 24 NP (1 NP = 7 CPU cores + 1 GPU card) to iterate 10000 steps, the radiation patterns of this array are shown in Figure 8. The result is almost identical with MoM method.

## 5. Discussion and Summary

This paper introduces a new FDTD code which uses both CPU and GPU resources to deal with the realistic electromagnetic problems. The loads of CPU and GPU are balanced by the parameter cuRatio. We used 64 NVIDIA TESLA K20m GPUs and 64 INTEL XEON E5-2670 CPUs to carry out

numerical tests. Compared to the pure GPU calculations for the same problems, the CPU + GPU calculations have 7.6%–13.2% performance improvement and the maximum problem size can be enlarged by 25%. Accroding to (5) and (6), this FDTD code is also available for different CPU and GPU configurations of other supercomputers. Moreover, the cudaFDTD is also compatible with the clusters without GPU devices in which the CUDA functions will be disabled. Several function modules like source excitation, near-to-far transformation, preprocessing, and postprocessing have been implemented in cudaFDTD; therefore this code can be applied directly to calculate the application problem as shown in Section 4.4. However, we already noticed that our code has some shortages which will be overcome in the future.

(1) The MPI parallel efficiency is not good enough (refer to Section 4.3). The complicated communication is one of the reasons, but there are some approaches to overlap this overhead, for instance, the nonblocking MPI communication.

(2) The OpenMP parallel efficiency is bad. For example, in Section 4.2, the cuRatio is 0.80 and 0.95 when using 15 CPU cores and 1 CPU core (note that one CPU core is used for CUDA stream). The speedup from single core to 15 cores is only a factor of 4.

## Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

## Acknowledgments

This work is supported by the National High Technology Research and Development Program (863 project) under Grant no. 2012AA01A308. The computations were done by using the GPGPU clusters of Center for High Performance Computing of Shanghai Jiaotong University. This work was also made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: http://www.sharcnet.ca/) and Compute/Calcul Canada.

## References

[1] K. S. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Transactions on Antennas and Propagation*, vol. 14, pp. 302–307, 1966.

[2] R. Ilgner and D. Davidson, "A comparison of the fdtd algorithm implemented on an integrated gpu versus a gpu configured as a co-processor," in *Proceedings of the International Conference on Electromagnetics in Advanced Applications (ICEAA '12)*, pp. 1046–1049, 2012.

[3] D. K. Price, J. R. Humphrey, and E. J. Kelmelis, "GPU-based accelerated 2D and 3D FDTD solvers," in *Physics and Simulation of Optoelectronic Devices XV*, vol. 6468 of *Proceedings of the SPIE*, January 2007.

[4] N. Takada, T. Shimobaba, N. Masuda, and T. Ito, "High-speed FDTD simulation algorithm for GPU with compute unified device architecture," in *Proceedings of the IEEE International Symposium on Antennas and Propagation and USNC/URSI National Radio Science Meeting (APSURSI '09)*, pp. 1–4, June 2009.

[5] K.-H. Kim, K. Kim, and Q.-H. Park, "Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model," *Computer Physics Communications*, vol. 182, no. 6, pp. 1201–1207, 2011.

[6] M. F. Hadi and S. A. Esmaeili, "Cuda fortran acceleration for the finite-difference time-domain method," *Computer Physics Communications*, vol. 184, pp. 1395–1400, 2013.

[7] S. E. Krakiwsky, L. E. Turner, and M. M. Okoniewski, "Acceleration of Finite-Difference Time-Domain (FDTD) using Graphics Processor Units (GPU)," in *Proceedings of the IEEE MITT-S International Microwave Symposium Digest*, vol. 2, pp. 1033–1036, June 2004.

[8] S. Adams, J. Payne, and R. Boppana, "Finite difference time domain (FDTD) simulations using graphics processors," in *Proceedings of the DoD High Performance Computing Modernization Program Users Group Conference*, pp. 334–338, June 2007.

[9] D. Liuge, L. Kang, and K. Fanmin, "Parallel 3D finite difference time domain simulations on graphics processors with CUDA," in *Proceedings of the International Conference on Computational Intelligence and Software Engineering (CiSE '09)*, December 2009.

[10] T. Nagaoka and S. Watanabe, "GPU-based 3D-FDTD computation for electromagnetic field dosimetry," in *Proceedings of the IEEE (AFRICON '11)*, pp. 1–6, Livingstone, Zambia, September 2011.

[11] T. Nagaoka and S. Watanabe, "Accelerating three-dimensional fdtd calculations on gpu clusters for electromagnetic field simulation," in *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC '12)*, pp. 5691–5694.

[12] K. H. Kim and Q. H. Park, "Overlapping computation and communication of three-dimensional FDTD on a GPU cluster," *Computer Physics Communications*, vol. 183, pp. 2364–2369, 2012.

[13] L. Xu, Y. Xu, R. L. Jiang, and D. D. Zhang, "Implementation and optimization of three-dimensional UPML-FDTD algorithm on GPU cluster," *Computer Engineering & Science*, vol. 35, p. 160, 2013.

[14] J. D. Jackson, *Classical Electrodynamics*, John Wiley & Sons, 3rd edition, 1998.

[15] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, Artech House, 3rd edition, 2005.

[16] D. B. Ge and Y. B. Yan, *Finite-Difference Time-Domain Method For Electromagnetic Waves*, Xidian University Press, 3rd edition, 2011.

[17] J.-P. Berenger, "A perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 114, no. 2, pp. 185–200, 1994.

[18] M. Livesey, J. Stack, F. Costen, T. Nanri, N. Nakashima, and S. Fujino, "Impact of gpu memory access patterns on fdtd," in *Proceedings of the IEEE Antennas and Propagation Society International Symposium (APSURSI '12)*, pp. 1–2, 2012.

[19] Nvdia, *CUDA C Programming Guide 13*, 2013.

[20] D. De Donno, A. Esposito, L. Tarricone, and L. Catarinucci, "Introduction to GPU computing and CUOA programming: a case study on FOTO," *IEEE Antennas and Propagation Magazine*, vol. 52, no. 3, pp. 116–122, 2010.