Progress In Electromagnetics Research, PIER 102, 381–395, 2010

GPU ACCELERATED UNCONDITIONALLY STABLE CRANK-NICOLSON FDTD METHOD FOR THE ANALYSIS OF THREE-DIMENSIONAL MICROWAVE CIRCUITS

K. Xu, Z. H. Fan, D. Z. Ding, and R. S. Chen

Department of Communication Engineering
Nanjing University of Science and Technology
Nanjing 210094, China

Abstract—The programmable graphics processing unit (GPU) is employed to accelerate the unconditionally stable Crank-Nicolson finite-difference time-domain (CN-FDTD) method for the analysis of microwave circuits. In order to efficiently solve the linear system from the CN-FDTD method at each time step, both the sparse matrix vector product (SMVP) and the arithmetic operations on vectors in the bi-conjugate gradient stabilized (Bi-CGSTAB) algorithm are performed with multiple processors of the GPU. Therefore, the GPU based BI-CGSTAB algorithm can significantly speed up the CN-FDTD simulation due to parallel computing capability of modern GPUs. Numerical results demonstrate that this method is very effective and a speedup factor of 10 can be achieved.

1. INTRODUCTION

Fast numerical simulation has particular importance for the analysis or design of microwave circuits used in a wireless communication system at microwave frequency. Many numerical methods have been proposed for this purpose. Among them, the finite-difference time-domain (FDTD) method developed by Yee [1] is used widely. However, this method suffers from a limitation that the maximum time step size is constrained by the minimum spatial resolution defined by the Courant-Friedrich-Levy (CFL) condition. To overcome this stability limit, the unconditionally stable Crank-Nicolson FDTD (CN-FDTD) method for two-dimensional (2-D) and three-dimensional (3-D)

Corresponding author: R. S. Chen (eechenrs@mail.njust.edu.cn).

Maxwell's Equations has been developed [2–4]. The main advantage of this method is that the time step size is no longer restricted by stability. Therefore this method is well suited for the analysis of microwave circuits with fine geometric details. Recently a 3-D CN-FDTD method for frequency-dependent media (FD-CN-FDTD) has been presented by Rouf et al. in [5].

In the processing of the CN-FDTD, a sparse linear system is needed to be solved at each time step. Generally, iterative solvers such as the conjugate gradient (CG) and bi-conjugate gradient stabilized (Bi-CGSTAB) [6, 7] algorithm are preferred to effectively solve the corresponding linear system. But the computation time for a single time step is larger than that of standard FDTD. Therefore, the development of efficient solution methods is crucial for the application of CN-FDTD in the analysis of microwave circuits. Recently the graphics processing unit (GPU) featuring massive parallelism and high memory bandwidth has been used extensively for the acceleration of a lot of scientific applications [8]. In the computational electromagnetic society, relative works have also been reported. Krakiwsky et al. implemented the FDTD on GPU with Opengl as the application programming interface (API) and gave an almost 10 time speedup over the CPU [9]. Inman et al. described their FDTD implementation on the GPU and got an acceleration ratio of 14 in 3-D simulations [10]. Zainud-Deen et al. also proposed a new GPU based implementation of the FDFD using the Brook+ API developed by AMD [11]. Tao et al. accelerated the Graphical electromagnetic computing (GRECO) method by moving all electromagnetic computing code to the GPU [12]. In the context of the method of moments (MoM), Peng and Nie proposed a GPU speedup scheme for filling impedance matrix and the iterative solution [13]. They achieved an acceleration ratio of about 20. These remarkable works inspired us to further investigate the possibility of employing the GPU for accelerating the iterative solution in the CN-FDTD simulation.

For the fast solution of a sparse, linear system from the CN-FDTD, the Bi-CGSTAB algorithm is used because of its faster convergence behavior while compared with the CG method. The Bi-CGSTAB algorithm is composed of two kinds of operations: matrix-vector product (MVP), arithmetic computation on vectors such as the dot-product of two vectors or vector-vector addition. Due to the inherent parallelism of these operations, the Bi-CGSTAB algorithm can be well mapped into the GPU with the help of Compute Unified Device Architecture (CUDA), which is a friendly API for programming parallel applications for all NVIDIA's GPUs [14].

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to the CN-FDTD. Then Section 3 describes the hardware implementation of current CUDA-capable GPU and the programming model of CUDA. The parallelization strategies for realizing the MVP and vector operations on the GPU are also provided in Section 3. At last, Numerical experiments and conclusions are presented in Sections 4 and 5, respectively.

2. BASIC THEORY OF THE CN-FDTD AND THE BI-CGSTAB ALGORITHM

2.1. Formations of the CN-FDTD

According to [4], the Crank-Nicolson scheme solves the discretized Maxwell's equations by a full time step size with one marching procedure, and averages the right-hand-sides of the discretized Maxwell's equations at $n+1$ time step and n time step. In three-dimensional lossless medium, a set of equations with unknown coupled electric field can be obtained by applying the CN algorithm to the Maxwell's equations. The update equations for electric field at $n+1$ time step can be written in the following form

$$\begin{aligned}&\left(1-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial y^2}-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial z^2}\right)E_x^{n+1}(i\cdot j\cdot k)+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial y\partial x}E_y^{n+1}(i\cdot j\cdot k)\\&+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial z\partial x}E_z^{n+1}(i\cdot j\cdot k)=\left(1+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial y^2}+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial z^2}\right)E_x^{n}(i\cdot j\cdot k)\\&+\frac{dt}{\varepsilon}\left(\frac{\partial}{\partial y}H_z^n(i\cdot j\cdot k)-\frac{\partial}{\partial z}H_y^n(i\cdot j\cdot k)\right)-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial y\partial x}E_y^n(i\cdot j\cdot k)\\&-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial z\partial x}E_z^n(i\cdot j\cdot k)\end{aligned}\tag{1a}$$

$$\begin{aligned}&\left(1-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial x^2}-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial z^2}\right)E_y^{n+1}(i\cdot j\cdot k)+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial x\partial y}E_x^{n+1}(i\cdot j\cdot k)\\&+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial z\partial y}E_z^{n+1}(i\cdot j\cdot k)=\left(1+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial x^2}+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial z^2}\right)E_y^{n}(i\cdot j\cdot k)\\&+\frac{dt}{\varepsilon}\left(\frac{\partial}{\partial z}H_x^n(i\cdot j\cdot k)-\frac{\partial}{\partial x}H_z^n(i\cdot j\cdot k)\right)-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial x\partial y}E_x^n(i\cdot j\cdot k)\\&-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial z\partial y}E_z^n(i\cdot j\cdot k)\end{aligned}\tag{1b}$$

$$\left(1-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial x^2}-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial y^2}\right)E_z^{n+1}\left(i\cdot j\cdot k\right)+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial x\partial z}E_x^{n+1}\left(i\cdot j\cdot k\right)$$
$$+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial y\partial z}E_y^{n+1}\left(i\cdot j\cdot k\right)=\left(1+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial x^2}+\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial y^2}\right)E_z^{n}\left(i\cdot j\cdot k\right)$$
$$+\frac{dt}{\varepsilon}\left(\frac{\partial}{\partial x}H_y^{n}\left(i\cdot j\cdot k\right)-\frac{\partial}{\partial y}H_x^{n}\left(i\cdot j\cdot k\right)\right)-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial y\partial z}E_y^{n}\left(i\cdot j\cdot k\right)$$
$$-\frac{dt^2}{4\mu\varepsilon}\frac{\partial^2}{\partial x\partial z}E_x^{n}\left(i\cdot j\cdot k\right)\tag{1c}$$

Then, the update equations for three field components can be rewritten in the form of matrix as follows

$$\mathbf{Ax}=\mathbf{b}\tag{2}$$

where $\mathbf{x}=[\mathbf{E}_x,\mathbf{E}_y,\mathbf{E}_z]^T$. In addition, the Mur's first-order absorbing boundary conditions (ABCs) [15] are used to truncate the outer surfaces in the CN-FDTD method we implemented.

2.2. Serial Bi-CGSTAB Algorithm

The Bi-CGSTAB algorithm is selected to solve the large, sparse and non-symmetric linear system (2) in the CN-FDTD method. An advantage of the Bi-CGSTAB algorithm is that the computation of transpose matrix vector multiplication is avoided at each iterative step. Since the introduction of the Bi-CGSTAB algorithm can be found in [6, 7], for simplicity the basic procedure is presented as follows.

```
The Bi-CGSTAB Algorithm
(1) For an initial guess $\mathbf{x}_0$, $\mathbf{p}_0=\mathbf{r}_0^*=\mathbf{r}_0=\mathbf{b}-\mathbf{Ax}_0$
(2) For $k=1,2,3,\cdots,n$ to convergence
(3) $a_k=\langle\mathbf{r}_k,\mathbf{r}_0^*\rangle/\langle\mathbf{Ap}_k,\mathbf{r}_0^*\rangle$
(4) $\mathbf{s}_k=\mathbf{r}_k-a_k\mathbf{Ap}_k$
(5) $\omega_k=\langle\mathbf{As}_k,\mathbf{s}_k\rangle/\langle\mathbf{As}_k,\mathbf{As}_k\rangle$
(6) $\mathbf{x}_{k+1}=\mathbf{x}_k+a_k\mathbf{p}_k+\omega_k\mathbf{s}_k$
(7) $\mathbf{r}_{k+1}=\mathbf{s}_k-\omega_k\mathbf{As}_k$
(8) $\beta_k=\langle a_k\mathbf{r}_{k+1},\mathbf{r}_0^*\rangle/\langle\omega_k\mathbf{r}_k,\mathbf{r}_0^*\rangle$
(9) $\mathbf{p}_{k+1}=\mathbf{r}_{k+1}+\beta_k(p_k-\omega_k\mathbf{Ap}_k)$
(10) Judge convergence: $\|\mathbf{r}_{k+1}\|/\|\mathbf{r}_0\|<\varepsilon$
```

The solution of linear systems consumes more than 95% of the CPU time for most of CN-FDTD applications in the analysis of microwave circuits. It is meaningful to develop an efficient method for iterative solution so that the simulation time of the CN-FDTD can be greatly reduced. Obviously, The Bi-CGSTAB algorithm can be divided as the following three basic operations: (1) *saxpy*, which performs the calculation of $\mathbf{z} = alpha * \mathbf{x} + \mathbf{y}$, where $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are both real vectors of size n, and *alpha* is a scalar; (2) computing the dot product of two vectors; (3) the sparse matrix vector product (SMVP). Since all these operations offer the parallelism, which can be exploited using parallel computing. In order to speed up the CN-FDTD simulation, the parallelization of these operations on the GPU is implemented by utilizing their parallelism in next section.

3. GPU ACCELERATING THE CN-FDTD: PROGRAMMING MODEL AND PARALLELIZATION STRATEGY

3.1. CUDA Programming Model and GPU Architecture

Before mapping a computing task into the GPU, it is necessary to well understand the hardware architecture and programming environment of the GPU, which are closely related to the parallelization strategies and computational resources available to a programmer. In our current implementation, the CUDA API provided by NVIDIA is used to program the GPU.

CUDA is an extension of C and an associated API for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API (such as the Opengl or DirectX) [8, 14]. On the other hand, it is also a hardware implementation of NVIDIA's GPU. According to [14], the computing module of a CUDA-capable GPU is a set of *multiprocessors,* each of them consisting of 8 *processors*, and each *processor* of the *multiprocessor* supports parallel executing model of the single-instruction multiple-data (SIMD). Furthermore, each *multiprocessor* has a set of 32-bit registers, as well as read-only caches, and a shared memory which can be used to share data between the threads within a thread block. The device memory is located out of chip and plays the role of storing massive data.

One highlight characteristic of CUDA technique is the multithread mechanism which can effectively hide the latency of accessing to the device memory by sustaining thousands of threads at the same time for general computing task. Generally, a highly parallel task characterized by the SIMD feature is divided into a collection of sub-tasks and each sub-task can be handled by one thread. Therefore, thousands or even millions of threads may be produced for a parallel task. These threads are organized as a grid of thread blocks and each block consists of a batch of threads. As shown in Fig. 1, a simple example of the CUDA thread model is given. The grid of thread blocks is then executed on the GPU by assigning blocks for execution on the *multiprocessors*, i.e., each *multiprocessor* processes batches of blocks one batch after the other. As the number of threads in a block is often larger than that of processors in a *multiprocessor*, a time-sliced strategy is applied to schedule threads for parallel execution to make every processor busy. The good introduction of the thread model and hardware implementation of the CUDA technique can be found in [14].

3.2. Parallelization Strategy for the *Saxpy*

In order to clarify the parallelization strategy for the *Saxpy*, an example of the *Saxpy* is shown in Fig. 2, where $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are both real vectors of size 8, and *alpha* is equal to 1.0. By assigning one thread to compute one element of vector $\mathbf{z}$, the calculation of *Saxpy* can be parallelized with the CUDA technique. Thus this parallelization strategy can directly be applied to step (4), (6), (7), (9) in the iteration of the Bi-CGSTAB algorithm.

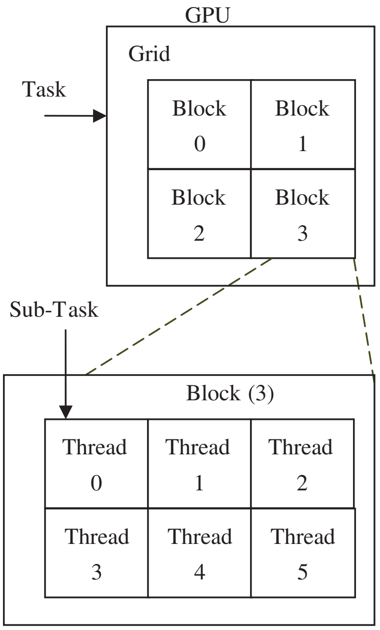


Figure 1. Thread model of the CUDA.

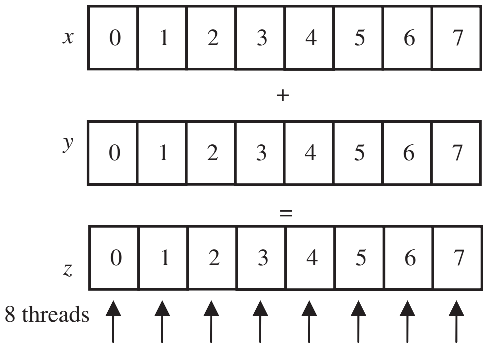


Figure 2. Thread model for calculating $\mathbf{z} = \mathbf{x} + \mathbf{y}$.

3.3. Parallelization Strategy for Computing the Dot Product of Two Vectors

The operation of two vectors' dot product reduces all the elements of two vectors to a single element. This fact leads to a complex strategy for parallelizing the operation of two vectors' dot product. To explain the parallel strategy clearly, we consider the calculation of the *dot(*$\mathbf{x}$,$\mathbf{y}$), where $\mathbf{x}$ and $\mathbf{y}$ are both real vectors of size 8. The process of the *dot*($\mathbf{x}$,$\mathbf{y}$) is divided into two steps in our implementation. The first step is to multiply the vector $\mathbf{x}$ by the vector $\mathbf{y}$ and store the results in an auxiliary vector $\mathbf{z}$ of size 8. Note that this step does the element by element multiplication with the proper elements of two vectors. Obviously, the parallelization strategy used for the *saxpy* is also suitable for the process of the first step. The second step is to sum the auxiliary vector $\mathbf{z}$ for the final result and a multi-pass summation algorithm is employed in this step. As shown in Fig. 3, the first pass has 4 threads, each of which reads two elements of vector $\mathbf{z}$ and writes their sum to a single element. In the second pass, the number of working threads reduces in half and each thread does the same process as the first pass. Then the following pass repeats again until a single thread is survived. Finally the result of the *dot*($\mathbf{x}$,$\mathbf{y}$) is obtained through the single thread. Similarly, the operation of two vectors' dot product within the Bi-CGSTAB algorithm can be accomplished by the GPU in parallel.

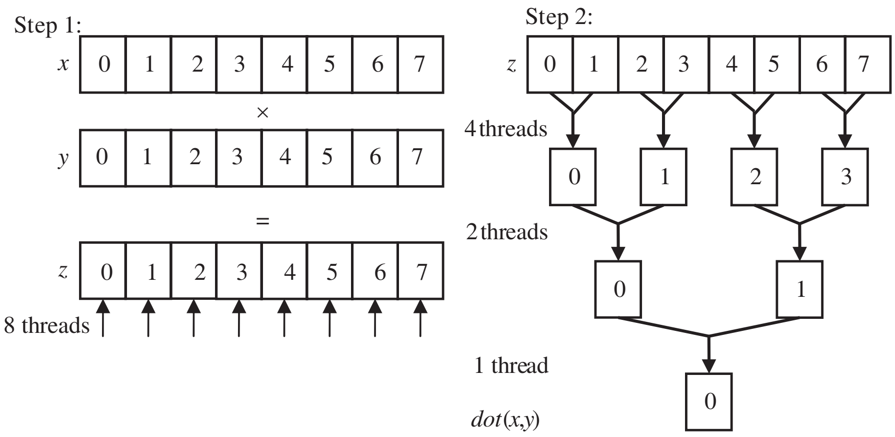


Figure 3. Thread model for calculating the dot-product of vector $\mathbf{x}$ and $\mathbf{y}$.

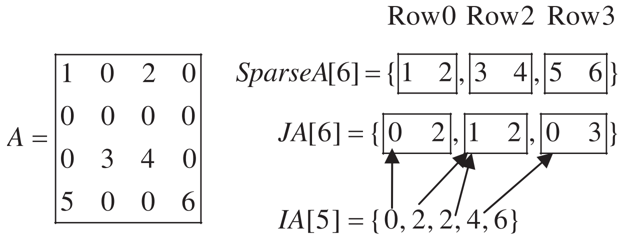

Row0 Row2 Row3
$A = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 5 & 0 & 0 & 6 \end{bmatrix}$
$SparseA[6] = \{1\ 2, 3\ 4, 5\ 6\}$
$JA[6] = \{0\ 2, 1\ 2, 0\ 3\}$
$IA[5] = \{0, 2, 2, 4, 6\}$

Figure 4. CSR representation of a sparse matrix.

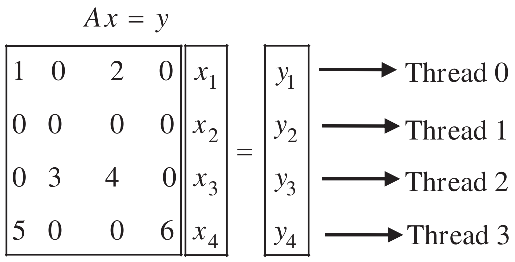


Figure 5. Thread model for calculating the SPMV.

3.4. Parallelization Strategy for the Sparse Matrix Vector Product

It is well known that the most common representation for sparse matrices is the compressed sparse row (CSR) format [16]. This CSR format is suitable for our GPU implementation of the SMVP. As shown in Fig. 4, the CSR representation of a simple sparse matrix **A** is given. A 1-dimensional (1-D) array named *SparseA* contains all the non-zero elements of the sparse matrix, the 1-D array *IA* indicates the position in the *SparseA* where each row starts, and *JA* stores the column index for each non-zero element. Similarly the sparse matrix arising from the CN-FDTD is also represented by these arrays. Considering the serial implementation of the $\mathbf{Ax} = \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are both column vectors of size 4, the ith element of vector $\mathbf{y}$ (y_i) is obtained by computing a dot product of the ith row of $\mathbf{A}$ and vector $\mathbf{x}$, and the element of vector $\mathbf{y}$ is calculated in turn. As shown in Fig. 5, since each element of vector $\mathbf{y}$ is computed independently, we can simply assign one thread to compute one element of vector $\mathbf{y}$. By using this parallelization strategy, the task of the SMVP in the CN-FDTD can be executed by multiple threads in parallel.

The parallel Bi-CGSTAB algorithm on the GPU can be implemented with above parallelization strategies. Hence the time for the iterative solution of the matrix equation can be reduced in the CN-FDTD. At the beginning of solving the linear system corresponding to a time step, the right hand vector **b** has to be transferred into the device memory of the GPU. When the convergence condition for a time step is achieved, the solution vector is copied into the main memory of the CPU. Therefore, our proposed GPU based Bi-CGSTAB algorithm has a low data communication between the CPU and GPU. For convenience, the GPU accelerated CN-FDTD method is called the GPU-CN-FDTD.

4. NUMERICAL RESULTS AND DISCUSSIONS

In this section, a number of numerical results are presented to illustrate the efficiency of the GPU-CN-FDTD method. All the results are derived on an Intel Core 2 E8400 CPU running at 3.0 GHz with 4 GB local memory and a NVIDIA GeForce GTX280 GPU with 1 GB video memory while using single-precision floating-point numbers. The GPU-CN-FDTD method is realized through the use of the CUDA technique in the Microsoft Visual Studio 2005. The conventional CN-FDTD with the Bi-CGSTAB algorithm is executed by the E8400 CPU.

In the CN-FDTD simulation, the time step size *dt* is ten times greater than that of the conventional FDTD. The iterative process is terminated when the relative residual norm is reduced by 10^{-7} for the Bi-CGSTAB algorithm. Additionally the zero vector is taken as an initial approximate solution.

As shown in Fig. 6, a T-junction microstrip filter is simulated [17].

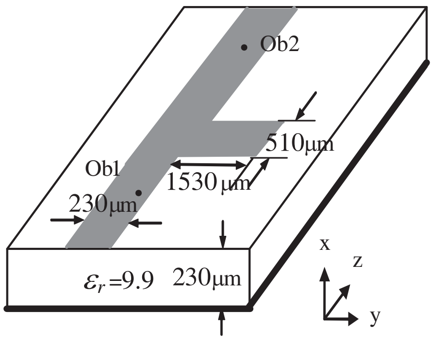


Figure 6. Geometry and dimensions for the T-junction microstrip.

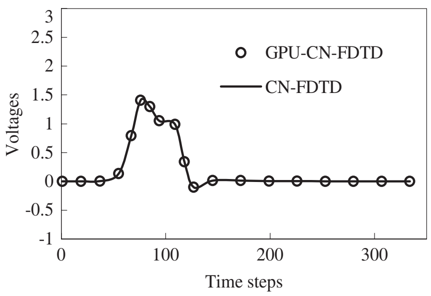


Figure 7. Comparison of time domain waveforms for voltages at observation point 2.

The grid size is chosen to be $dx = 57.5\,\mu\text{m}$, $dy = 38.25\,\mu\text{m}$, $dz = 102\,\mu\text{m}$, leading to a mesh of $16 \times 66 \times 60$. A 15 ps Gaussian pulse is used for source excitation and the size of the time step is set as 1.014 ps. Since the current GPU supports a single-precision floating-point standard which is a subset of IEEE 754, sufficient accuracy can be provided for GPU computing [14]. As shown in Fig. 7, the time domain voltage waveform at the observation point 2 is given. It can be seen that the results of the GPU-CN-FDTD agree well with those of conventional CN-FDTD method.

The second example considered here is the calculation of a microstrip branch line coupler [18] as shown in Fig. 8. The grid size is chosen to be $dx = 0.265$ mm, $dy = 0.4064$ mm, $dz = 0.406$ mm, resulting into a mesh of $15 \times 54 \times 80$. The dielectric constant ε_r is

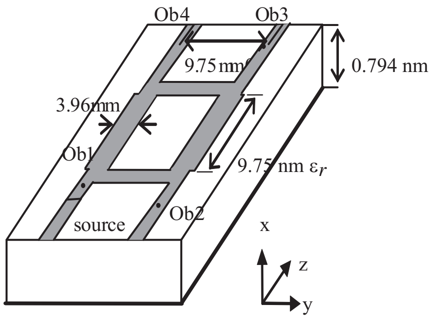


Figure 8. Geometry and dimensions of a microstrip branch line coupler.

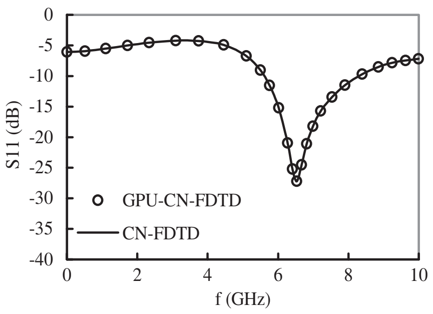


Figure 9. Comparison of S_{11} parameter for the microstrip branch line coupler.

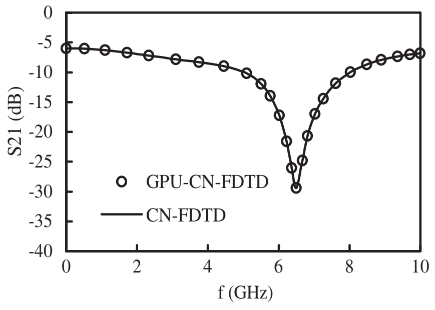
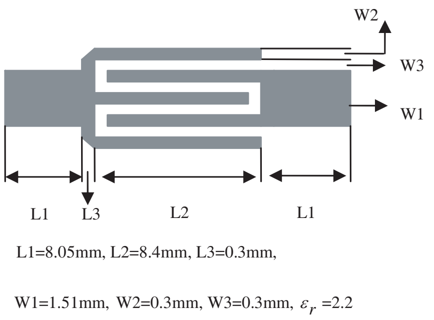


Figure 10. Comparison of S_{21} parameter for the microstrip branch line coupler.



W2
W3
W1
L1
L3
L2
L1
L1=8.05mm, L2=8.4mm, L3=0.3mm,
W1=1.51mm, W2=0.3mm, W3=0.3mm, ε_r =2.2

Figure 11. Geometry and dimensions of an interdigital capacitor.

equal to 2.2, a 25 ps Gaussian pulse is used for source excitation and the time step size is 6.4967 ps. As shown in Figs. 9 and 10, the S_{11} and S_{21} parameters are calculated by the GPU-CN-FDTD and conventional CN-FDTD method, respectively. It can be found that there is a good agreement between them.

As shown in Fig. 11, an interdigital capacitor is simulated and its dimensions are given in [19]. The interdigital capacitor is modeled by a mesh of $16\times38\times220$, where the grid size is chosen to be $dx = 0.125$ mm, $dy = 0.15$ mm, $dz = 0.15$ mm. The dielectric constant ε_r is equal to 2.2. The size of Gaussian pulse and time step is 30 ps, 2.6977 ps, respectively. As shown in Figs. 12 and 13, the S_{11} and S_{21} parameters are computed by the GPU-CN-FDTD and conventional CN-FDTD. It can be seen that there is also a good agreement between the GPU-CN-FDTD and CN-FDTD methods.

Table 1 illustrates the simulation time of both the GPU-CN-FDTD method and the conventional CN-FDTD method with respect to the above circuits. The numbers of time step required in the simulations are 350 for the T-junction filter, 250 for the microstrip branch line coupler, 3000 for the interdigital capacitor, respectively. In addition, the numbers of the electric field components to be

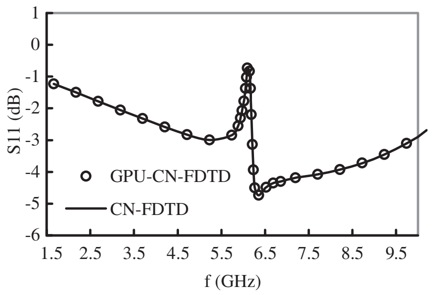


Figure 12. S_{11} parameter comparison for the interdigital capacitor.

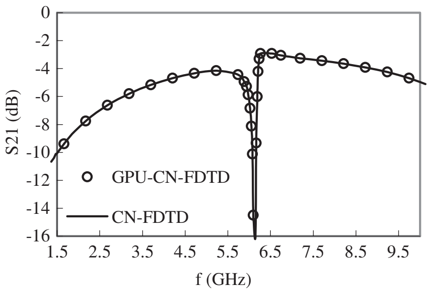


Figure 13. S_{21} parameter comparison for the interdigital capacitor.

Table 1. Total solution time of GPU accelerated CN-FDTD and conventional CN-FDTD method (in second).

	CPU	GPU	Speedup
T-junction microstrip	2076.188	193.828	10.711
Coupler	1367.766	124.469	10.988
Interdigital capacitor	32202.330	2794.344	11.524

solved (unknowns) are 184856 for the T-junction filter, 186777 for the microstrip branch line couple, and 387681 for the interdigital capacitor, respectively. It can be found from Table 1 that the GPU-CN-FDTD is about 10 times faster than the conventional CN-FDTD method. The numerical simulations demonstrate that a GPU can significantly speed up the conventional CN-FDTD method due to the parallel processing ability of the GPU.

In order to show how the number of threads affects the performance of GPU-CN-FDTD method, the SMVP on the GPU is compared with the serial SMVP executed by the CPU since the SMVP consumes 80% of the total time at each iterative step of GPU-CN-FDTD method in our numerical experiments. Considering the calculation of the $\mathbf{Ax} = \mathbf{y}$ for the interdigital capacitor (the size of vector $\mathbf{y}$ is 387681), the number of threads is increased from 1 to 387681. When the number of threads is smaller than 387681, each thread will compute multiple elements of vector $\mathbf{y}$. Fig. 14 shows the impact of number of threads on the speedup factors of the SMVP. It can be seen that the larger the number of threads is, the shorter time is required for the SMVP. When the number of threads is equal to 387681, the maximum speedup factor is achieved. This demonstrates that the optimal strategy for parallelizing the SMVP is to assign one thread for computing one element of vector $\mathbf{y}$. Since the SPMV on the GPU is mainly limited by memory bandwidth [8, 14], most of the execution time is spent to read fresh data from the memory. For minor number of threads, the multiple processors of GPU are not fully used and thus only a modest speedup factor is obtained. Furthermore, the same conclusions can be obtained for parallelizing the *saxpy* and dot product of two vectors on the GPU.

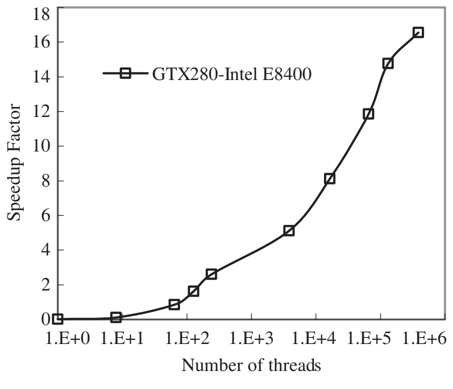


Figure 14. Speedup factor of SMVP for different number of threads.

5. CONCLUSION

In this paper, the GPU is employed as a co-processor of CPU to accelerate the CN-FDTD simulation for the analysis of three-dimensional microwave circuits. The Bi-CGSTAB solver on the GPU is proposed to solve sparse linear systems from the CN-FDTD. With multithread model of the CUDA technique, the sparse matrix vector product and vector operations are parallelized and then executed by the GPU. Numerical results demonstrate that the GPU accelerated CN-FDTD agrees well with the conventional CN-FDTD method and a significant reduction of the solution time can be obtained.

ACKNOWLEDGMENT

We would like to thank the support of Major State Basic Research Development Program of China (973 Program: 2009CB320201), Natural Science Foundation (Grant Nos. 60871013, 60701003, 60701004, 60701005), and Jiangsu Natural Science Foundation (Grant No. BK2008048).

REFERENCES

1. Yee, K. S., "Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media," *IEEE Trans. Antennas Propagat.*, Vol. 14, No. 8, 302–307, Aug. 1966.
2. Sun, G. and C. W. Trueman, "Unconditionally stable Crank-Nicolson scheme for solving the two-dimensional Maxwell's equations," *IEE Electron. Lett.*, Vol. 39, No. 7, 595–597, Apr. 2003.
3. Sun, G. and C. W. Trueman, "Approximate Crank-Nicolson schemes for the 2-D finite-difference time-domain method for TE waves," *IEEE Trans. Antennas Propagat.*, Vol. 52, No. 11, 2963–2972, Nov. 2004.
4. Yang, Y., R. S. Chen, D. X. Wang, and E. K. N. Yung, "Unconditionally stable Crank-Nicolson finite-difference time-domain method for simulation of 3-D microwave circuits," *IEE Microwaves, Antennas & Propagation*, Vol. 1, No. 4, 937–942, Aug. 2007.
5. Rouf, H. K., F. Costen, S. G. Garcia, and S. Fujino, "On the solution of 3-D frequency dependent Crank-Nicolson FDTD scheme," *Journal of Electromagnetic Waves and Applications*, Vol. 23, No. 16, 2163–2175, 2009.

6. Van der Vorst, H. A., "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems," *SIAM J. Sci. Statist. Comput.*, Vol. 13, No. 2, 631–644, Mar. 1992.
7. Zhang, S. L., "GPBi-CG: Generalized product-type methods based on Bi-CG for solving nonsymmetric linear systems," *SIAM J. Sci. Comput.*, Vol. 18, No. 2, 537–551, Mar. 1997.
8. Owens, J. D., M. Houston, et al., "GPU computing," *Proceedings of the IEEE*, Vol. 96, No. 5, 879–899, May 2008.
9. Krakiwsky, S. E., L. E. Turner, and M. M. Okoniewski, "Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU)," *In IEEE MTT-S Int. Microwave Symp. Digest*, 1033–1036, 2004.
10. Inman, M. J. and A. Z. Elsherbeni, "Programming video cards for computational electromagnetics applications," *Antennas Propag. Mag.*, Vol. 47, 71–78, Dec. 2005.
11. Zainud-Deen, S. H., E. El-Deen, et al., "Electromagnetic scattering using gpu-based finite difference frequency domain method," *Progress In Electromagnetics Research B*, Vol. 16, 351–369, 2009.
12. Tao, Y. B., H. Lin, and H. J. Bao, "From CPU to GPU: GPU-based electromagnetic computing (GPUECO)," *Progress In Electromagnetics Research*, PIER 81, 1–19, 2008.
13. Peng, S. X. and Z. P. Nie, "Acceleration of the method of moments calculations by using graphics processing units," *IEEE Trans. Antennas and Propagation*, Vol. 56, No. 7, 2130–2133, Jul. 2008.
14. NVIDIA Corporation, *NVIDIA CUDA Programming Guide*, Version 1.1, Nov. 2007.
15. Mur, G., "Absorbing boundary conditions for the finite-difference approximation of the time-domain electromagmetic-field equations," *IEEE Trans. Electromagnetic Compatibility*, Vol. 23, No. 4, 377–382, Nov. 1981.
16. Gibson, W. C., *The Method of Moments in Electromagnetics*, Chapman & Hall/CRC, 2007.
17. Bracken, J. E., D. K. Sun, and Z. J. Cendes, "S-domain methods for simultaneous time and frequency characterization of electromagnetic devices," *IEEE Trans. Microwave Theory Tech.*, Vol. 46, 1277–1290, Sep. 1998.
18. Sheen, D. M., S. M. Ali, M. D. Abouzahra, and J. A. Kong, "Application of the three-dimensional finite-difference time-domain method to the analysis of planar microstrip circuits,"

IEEE Trans. Microwave Theory Tech., Vol. 38, 849–857, Jul. 1990.

19. Maricevic, Z. A. and T. K. Sarkar, "Analysis and measurements of arbitrarily shaped open microstrip structures," *Progress In Electromagnetics Research*, PIER 15, 253–301, 1997.